MAPS

NUMMER 41 • NAJAAR 2010

NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

Voorzitter
Taco Hoekwater
ntg-president@ntg.nl

Secretaris
Willi Egger
ntg-secretary@ntg.nl

Penningmeester
Ferdy Hanssen
ntg-treasurer@ntg.nl

Bestuursleden
Frans Absil
fgj.absil@nlda.nl

Frans Goddijn
frans@goddijn.com

Hans Hagen
pragma@wxs.nl

Postadres
Nederlandstalige TEX Gebruikersgroep
Maasstraat 2
5836 BB Sambeek

ING bankrekening
1306238
t.n.v. NTG, Arnhem
BIC-code: INGBNL2A
IBAN-code: NL53INGB0001306238

E-mail bestuur
ntg@ntg.nl

E-mail MAPS redactie
maps@ntg.nl

WWW
www.ntg.nl

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 document-opmaak 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 0.64.0 draaiend onder Linux 2.6. De gebruikte fonts zijn Linux Libertine, het niet-proportionele font Inconsolata, schreefloze 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 letterzet-mogelijkheden van TEX. Voorbeelden zijn LATEX van Leslie Lamport, $\mathcal{AMS}$-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Inhoudsopgave

Redactioneel

Laat ik beginnen met iedereen een

Gelukkig en productief 2011

toe te wensen namens de Maps redactie en het NTG bestuur.

Tegen de tijd dat je dit leest zou het namelijk wel eens voorbij half januari kunnen zijn. Met andere woorden: deze najaars-Maps is te laat. Ik zou graag willen kunnen zeggen dat het aan weerbarstige auteurs lag, of dat de redactie onderbezet is, of dat de ConTEXt beta die ik gebruik zorgde voor onbetrouwbare uitvoer waardoor ik op Hans moest wachten voor een nieuwe beta, of aan vertraging bij TNT post vanwege de stakingen, of dat de drukker het veel te druk had vanwege de eindejaars-rush, maar helaas. Al voornoemde zaken zijn waar, maar de hoofdreden is dat ik het veel te druk heb gehad met andere zaken om me op de Maps te kunnen concentreren. En dat is niet voor het eerst, dus hierbij mijn welgemeende excuses voor de veroorzaakte vertraging.

En als ik me dan toch aan het verontschuldigen ben, laat ik dan ook maar meteen schuld bekennen over de vreselijk lelijke inhoudsopgave op de achterkant van Maps 40. Om eerlijk te zijn weet ik nog steeds niet wat er fout is gegaan daar, maar ik beloof dat de inhoudsopgave van deze Maps met argusogen is bekeken.

Nu ik toch over bekijken ben begonnen: Siep Kroonenberg heeft eind van het jaar de stylesheet van de NTG website onder handen genomen. Wijzelf zijn erg tevreden over de nieuwe frisse uitstraling van de NTG website, maar ga vooral even kijken op http://www.ntg.nl om het resultaat met eigen ogen te zien.

En dan deze Maps zelf. Er is weer een kleine layout aanpassing; de titels en eventuele subtitels van de artikelen zijn niet meer schreefloos, maar in hetzelfde font als de eigenlijke tekst (Linux Libertine). Met deze verandering ziet de aanvang van een artikel er nu wat rustiger uit. En wie weet, in de volgende Maps veranderen wellicht de kopjes en subkopjes ook nog wel.

Die volgende Maps wordt overigens waarschijnlijk niet meer gedrukt in offset, maar geprint via een goede Nederlandse *printing on demand* service die Hans Hagen heeft ontdekt. We zouden dat liever niet doen, maar door het langzaam maar gestaag zakken van het ledenaantal van de NTG zakt uiteraard ook de oplage van de Maps, en het gebruik van een echte drukker begint een onredelijke zware druk op de begroting te worden.

En er is ook een lichtpuntje aan de omschakeling naar printen: voortaan is het dan niet meer belangrijk of de artikelen in kleur helemaal vooraan of helemaal achteraan staan. Bij printwerk wordt er namelijk per pagina afgerekend voor kleur of zwart-wit, in plaats van per katern zoals bij drukwerk.

Een laatste nieuwtje: Hans Hagen en ik hebben op http://www.boekplan.nl een site, waar diverse boeken via printing on demand worden aangeboden. Op diezelfde site is een pagina aangemaakt waar jullie oudere Maps uitgaven kunnen bestellen (uiteraard zolang de voorraad strekt). Het gebruik van de site is eenvoudig en de weg wijst zich vanzelf. Dus als je een bepaalde Maps niet meer hebt, of zo vaak gelezen hebt dat hij vervangen zou moeten worden, kijk dan eens op de boekplan site. Alle opbrengsten uit de NTG catalogus van boekplan gaan rechtstreeks naar de NTG, dus je sponsort er ook nog eens de vereniging mee.

Met al die nieuwtjes is er niet veel ruimte meer over op deze pagina, dus voor deze ene keer ga ik niet proberen een samenvatting te geven van de inhoud van Maps 41. De deadline van Maps 42 is 1 April 2011, en tot die tijd:

Veel leesplezier toegewenst,

Taco Hoekwater

EuroBachoTEX 2011: CALL FOR PAPERS

Æsthetics and effectiveness of the message, cultural contexts. The BachoTEX conference in 2011, already XIXth of the series, will be held from April 29th until May 3rd 2011. It will also be the official European TEX conference, hence we will reference it by the already traditional name EuroBachoTEX with the distinguishing "2011". BachoTEX conferences are being organized yearly since 1993 by GUST, the Polish TEX Users Group.

As the lead theme we propose the *æsthetics of publications* from the perspective of the effectiveness of the message. Encouraged are also references to the cultural contexts – not everywhere and everything is liked by all.

We await TEX, Metapost, ConTEXt, LATEX, and friends related presentations revolving around those issues, but will be more than happy if programmers and designers of typographic systems, typographers and other users of such systems want to share their thoughts and experience.

In addition, we await papers on support that typography systems can offer to the disabled, e.g., in connection with sign languages or generating speech (e.g., from TEX's mathematical notation). Perhaps somebody looked into issues with non-alphabetic notations such as tabulature (http://en.wikipedia.org/wiki/Tablature).

Also, please note the "Call for TEX Pearls" below.

Workshops and tutorials. Especially welcome are proposals for TEX-related tutorials or introductions. If you have suggestions for tutorials or workshops by others than yourself or about specific topics, please let us know.

Poster sessions. All participants will be given the opportunity to present their TEX and typographic results in the form of posters. We will provide exhibition space. Perhaps new ideas or solutions will emerge?

Call for TEX Pearls. We are continuing the tradition of "The Pearls of TEX Programming". Here is, briefly, what is wanted:
- short TEX, Metafont or Metapost macro(s), not necessarily useful,
- the solution not obvious at the first glance,
- easy to explain: necessarily 10 minutes at most.

If you have something that fits the bill, please consider submitting a proposal. If you know of somebody's work that does the same, please let us know, and we will contact that person. The email address is: pearls at gust dot org dot pl. Previously collected pearls can be found at http://www.gust.org.pl/projects/pearls.

The TEX Clinic. We hope that more advanced TEXies will help out the TEX Clinic team led by Joanna Ryćko (http://www.gust.org.pl/projects/klinika).

Deadlines and addresses. The deadline for abstracts and other proposals is March 28th 2011. The deadline for final papers to appear in the conference materials is April 11th.

Contributions should be send by email to the Programme Committee: prog-ebt2011 at gust dot org dot pl. The PC is chaired by Bogusław Jackowski (b underscore jackowski at gust dot org dot pl).

tlcontrib.metatex.org

Abstract
TLContrib is a distribution and associated website that hosts contributed, supplementary packages for TEX Live. The packages on TLContrib are those not distributed inside TEX Live proper for one or several of the following reasons: because the are not free software according to the FSF guidelines, because they contain an executable update, because they are not available on CTAN, or because they represent an intermediate release for testing.
Anything related to TEX that can not be on TEX Live but can still legally be distributed over the Internet can have its place on TLContrib.

Keywords
TEX Live, TLContrib, distribution, contribution, packages

Introduction

Many of you are familiar with TEX Live as an easy way to install TEX. This distribution provides a comprehensive TEX system with binaries for most flavors of Unix, including GNU/Linux, and also Windows. It includes all the major TEX-related programs, macro packages, and fonts that are free software, including support for many languages around the world. The current version is TEX Live 2010.

TEX Live is distributed on DVD by most of the local TEX user groups, but it also allows for continuous package updates over the Internet using the `tlmgr` program.

TEX Live is a wonderful tool, but there are a few considerations to be aware of:

- it only contains FSF-defined 'free' software packages
- it uses CTAN as its primary source for packages
- it does not make interim executable updates
- it is not a suitable medium for package test releases

Each of these limitations has a perfectly reasonable cause:

- The TEX Live maintainers agree (at least for the purposes of working on TEX Live) with the principles and philosophy of the free software movement. Therefore they follow the FSF guidelines on licensing.
- It is good for the TEX community if CTAN is as complete as possible. That gives users one place to look, for instance. Also, it makes it more likely for separate distributions like TEX Live and MiKTeX to be consistent with each other. By using CTAN as the primary package source, TEX Live promotes the use of CTAN.

  A secondary reason for the use of CTAN is that creating a large distribution like TEX Live takes a lot of work, and the number of volunteers is limited. Having a single place to check for new package updates is a lot easier, because this process can be automated to a large extent. Using many separate sources would make this task much more complicated.
- TEX Live ships binaries for 19 different computer platforms, and something like 300 binaries need to be compiled for each of those. Coordinating the task of preparing these binaries is a major effort.
- Because TEX Live is not just a network installation, but also shipped on DVD, it is important that the included packages and binaries are as stable as possible. After all, there is no guarantee that the DVD users will *ever* update their system after the initial installation.

Nevertheless, the limitations of TEX Live mean that there is room for extension. This is the reason for the existence of TLContrib.[1]

On TLContrib, anything that is freely distributable is acceptable, so packages that are not on CTAN are also fine, and TLContrib can and will contain updates to executables (just not necessarily for all platforms).

This is possible because the two major limitations of TEX Live do not exist in TLContrib. Firstly, TLContrib is a network-only distribution without the limitations introduced by the physical medium. Secondly, the problem of lack of human resources is solved by offloading the burden of creating and maintaining packages to the actual package maintainers.

Before going on to explain how to use TLContrib, it is important to note the following:

- TLContrib is *not* a full TEX Live repository: it is a completement and contains only its own packages. This means TLContrib can only be used as a secondary repository on top of an existing TEX Live installation.
- TLContrib is not maintained by the TEX Live team:

the responsibility for the actual packages lies with the package maintainers themselves, and the server maintenance is handled by yours truly.

There is no competition between TLContrib and TEX Live, but as one of the goals of TLContrib is to ease the workload of the TEX Live team, it would not make much sense for them to be the actual maintainers. For this reason there is a separate mailing list dedicated to TLContrib.[2] Please address your questions related to packages obtained from TLContrib there, and not on the regular TEX Live list.

Using TLContrib as a distribution

First things first: before attempting to use TLContrib, make sure that you have the latest (network) update of TEX Live 2010, and in particular that you run the latest `tlmgr`. During the development of TLContrib, a small number of incompatibilities have been found in the `tlmgr` as distributed on the DVD that have since been fixed. Furthermore, the current version of TLContrib only works with TEX Live 2010 and not for any earlier versions of TEX Live.

And a warning: Executable packages are not necessarily available for all platforms on TLContrib. Unfortunately, it appears that the current TEX Live update manager is not smart enough to correctly detect versioning in dependencies. In practice, this means that you should not update packages that depend on executable package updates *unless* the actual executable package update is also available on TLContrib for your platform.

In order to use TLContrib as an extra repository in the TEX Live 2010 package manager (`tlmgr`), there are two options, depending on whether you prefer to use the command line version or the GUI version of the TEX Live 2010 package manager.

Graphical interface usage

In the GUI version of the package manager, select the menu item Load other repository ... from within the tlmgr menu. Set the value to

```
http://tlcontrib.metatex.org/2010
```

There is currently no way to save this setting.

Besides not being able to save the TLContrib setting, when using the graphical user interface it is not always easy to see whether executable package updates are available. For this reason you should consider using the command line version of the package manager for use with TLContrib, even if you are accustomed to using the GUI interface.

Command line usage

The simplest approach is to just start `tlmgr` from the command line with an extra option:

```
$ tlmgr --repository                          \
  http://tlcontrib.metatex.org/2010
```

If you plan to use TLContrib regularly, it makes sense to define a shell alias to save you some typing (the next trick is courtesy of Will Robertson).

Define an alias to make things easier to remember; put this into your `.bash_profile` or equivalent (this has to be on a single line):

```
alias tlc="tlmgr --repository
           http://tlcontrib.metatex.org/2010"
```

You can now view what is available in the TLContrib repository with standard `tlmgr` commands such as

```
$ tlc list
```

to see what is currently available for installation. Packages can be updated to their pre-release versions by typing, say,

```
$ tlc update siunitx
```

and if an update performed in this way 'goes bad' and you'd like to revert to the official release, execute

```
$ tlmgr install fontspec --reinstall
```

and things will be back to normal.

Using TLContrib for distribution

The rest of this article describes further details important for a package maintainer aiming to use TLContrib for distribution.

Before you decide to add a package to TLContrib, please bear this in mind:

- It is not the intention of TLContrib to replace either TEX Live or CTAN: if a package is not blocked from TEX Live for one of the reasons mentioned earlier, and can be made available on TEX Live or CTAN, then it should not be part of TLContrib at all.

In order to be able to upload packages to TLContrib, you have to be a registered user. You can register as a user via the TLContrib website, and, not by coincidence, this is also the place where you create new packages and package releases.

After registration is complete, you can log in to TLContrib by following the member section link.

If you do upload a package to TLContrib, please also

subscribe to the TLContrib mailing list, because any questions about your package a likely to be made there.

Package creation example

This quick start guide uses an update of the
`context-lettrine`
package as an example of how to create a package. In the following, you need to replace `context-lettrine` by the actual package name that *you* are updating, of course.

Besides being logged in to TLContrib, the first thing you need to do is to create your updated package source. In this case, the easiest way is to start from the current TEX Live version, so first you have to fetch the current `context-lettrine` archive(s) from the network distribution of TEX Live. The base URL is: http://www.ctan.org/tex-archive/systems/texlive/tlnet/archive.

In fact, for this example, there are two archives to be downloaded:

```
context-lettrine.tar.xz
context-lettrine.doc.tar.xz
```

For some TEX Live packages there is even a third archive file named `<package>.source.tar.xz`. This is because the distribution system of both TEX Live and TLContrib splits the contribution into run-time files, documentation files, and source files. Users can ask the installer not to install the last two file types to save on disk space and network traffic.

You have to create a single local archive file with the combined and updated content of the two downloaded archives. After extracting both `tar.xz` files in the same directory, you will have a tree structure that looks like this:

```
doc/
  context/
    third/
      lettrine/
        lettrine-doc.pdf
        lettrine-doc.tex
        W.pdf
tex/
  context/
    interface/
      third/
        lettrine.xml
    third/
      lettrine/
        t-lettrine.tex
tlpkg/
  tlpobj/
    context-lettrine.doc.tlpobj
    context-lettrine.tlpobj
```

First, delete the whole `tlpkg` sub-tree. The `tlpobj` files contain meta-data specific to each particular revision of a package, and the information in the downloaded version of these files will henceforth be no longer applicable. New versions of the `tlpobj` files will be generated automatically by TLContrib's distribution creation tool.

You may now update the other files in the tree, and create the archive file (the acceptable formats are `tar.gz`, `tar.xz`, and `zip`). Please read the next section named '**About package sources**' carefully before finalizing the archive.

The TLContrib `context-lettrine` package will use the newly created archive as source for the package, so make doubly sure you use the right files. The use of existing TEX Live package archive(s) to start with is just so you get an idea of what goes where: sometimes TEX Live packages contain more files and symbolic links than you initially expect. You can build the source package completely from scratch if you want to, but it is easy to forget files if you don't check.

Incidentally, while the base name of the local archive file does not matter, you have to make sure that the extension is `.tar.gz`, `.tar.xz`, or `zip`, otherwise the upload will fail.

Now go to `http://tlcontrib.metatex.org`, log in, and click new package. As the new package is an update to TEX Live, make sure you select that option, and the proper package name from the drop-down (`context-lettrine`).

In the next screen, most of the needed input will be automatically filled in for you, based on the current TEX Live revision of the package.

Edit the rest of the input form to have a proper version and set the source to File upload. Its value has to be set to the new archive that was created earlier. Adjust the Release state drop-down so it is set to public. It is also wise to check the license field, for it does not always import correctly due to database mismatches.

Then press submit new revision, verify the upload, and submit again to finalize the new package.

Assuming all went well, all that is needed for now is to wait until the hour has passed: your package should be available from the TLContrib repository after that.

The TLContrib distribution system works asynchronously: the front-end data that you as a package maintainer can create and modify is exported to the user-side TLContrib repository by a cron job that runs independent of the actual website. Currently this cron job runs hourly, on the hour.

About package sources

Please note: Currently only the `tar.gz`, `tar.xz`, and `zip` archive formats are supported in the File upload

and HTTP URL methods, and there are further strict requirements on the archive itself:

For a non-executable package, it should contain a complete TDS[3] sub-tree. In TEX Live, normally all macro files go under `texmf-dist`, and, in that case, this directory level can be skipped in the archive (it will be added automatically by the TLContrib publication system). Be advised that, in general, uploading a CTAN zipped folder will *not* work, because CTAN packages are almost never in TDS format.

For an executable package, you can also use the TDS layout (with the binaries in `bin/$ARCH/`), but if you only have files inside the binaries folder, you can skip the directory structure completely: in this case, the TLContrib publication system will automatically add the needed structure.

Make sure that your archive contains only files that belong to your package, and especially that it does not accidentally overwrite files owned by other packages.

Also, check twice that the archive contains only files that belong in the TDS: Delete backup files, and remove any special files that may have been added by the operating system (MacOSX especially has a very bad habit of adding sub-directories for its Finder that really do not belong in the package).

It is not always simple to guess what should go into a TEX Live update package. If you are building such an updated package, it is always wise to start from the existing TEX Live sources.

TLContrib accepts no responsibility for package contents: the system does run some sanity checks, but ultimately, you as maintainer are responsible for creating a correctly functioning package. Badly behaving or non-working packages will be removed on executive decision by the TLContrib maintainer(s) without prior notice.

Package creation in detail

When you create a new package, a short wizard will help present itself to help you set up the package type. There are two types of packages: those that are updates of existing TEX Live packages, and those that are standalone. The wizard screen presents you the choice between these two types, and a dropdown listing TEX Live packages. The list of existing TEX Live packages is updated daily. Once this decision is made, it becomes fixed forever: the Id field of a package cannot be edited afterwards.

The Id field is the internal identifier of the package. Id-s should consist of a single 'word' with a length of at least two characters that only contains alphanumerics, dashes, and underscores. It can optionally followed by a platform identifier, which is then separated from the first part by a single dot.

Also note that when Release state becomes public (as explained below), it will no longer be possible to edit that particular release of the package. All further edits will force the creation of a new release, with a new revision id, and needing new sources.

Yet another note: If you intend to create an executable package, you have to be really sure you know what you are doing. Creating portable binaries for any platform is far from trivial. Paraphrasing Norbert Preining from the TLContrib mailing list:

> "If you have NO experience with compiling, preparing binaries for various platforms, distributing, etc., JUST DO NOT GO THERE!"

Macro packages are much easier; for those you only need a good understanding of how the TDS works.

Package editing

After the initial New package wizard screen, or after pressing Edit in the your package list for pre-existing packages, you will be presented with a fairly large edit screen.

During the initial TLContrib package creation process, if the package is updating an existing TEX Live package, certain fields will have been filled in automatically from the TEX Live package database. Otherwise you will have to fill in everything yourself.

Title
: This is the human-readable name of your package

Description
: This is a description in a few sentences of what the package does.

Package type
: Even though the drop-down is long, really there are only two choices in the drop-down: A package is either a Macro package, or a Executable package. The distinction is important because the required package source structure is different for each of the two types, as explained below.

TLMGR directives
: A list of TLMGR directives like e.g. addMap or addFormat. A better interface is planned, but, for the moment, you have to make sure you know what you are doing. Have a look at the existing TEX Live package database (`texlive.tlpdb`) for examples.

  You only have to specify the directives, do not add execute at the start.

TL dependencies
: Package Id-s of other TEX Live packages on which this package depends, one per line. Unless you know exactly what is needed, it is probably best to leave this field blank, but in any case:

You only have to specify the package Id-s, do not add depend at the start. If your package depends on a executable package, for example luatex, write the Id as luatex.ARCH. Doing so will make `tlmgr` automatically select the appropriate executable package for the user's platform.

TL postactions
: A list of TLMGR post-install actions like e.g. shortcut or fileassoc. A better interface is also planned, but, for the moment, you have to make sure you know what you are doing here as well. Have a look at the existing TEX Live package database (`texlive.tlpdb`) for examples.

  You only have to specify the actions, do not add postaction at the start.

License
: Pick one from the two drop-downs, and set the radio button accordingly. If you need to use Other free license or Other non-free license, please drop me an email. I am sure the list is incomplete. In this context, Free means: according to the Debian Free Software Guidelines.

Log message
: This field is just for release notes: it will not be exported to the TLContrib repository. The SVN URL and GIT URL methods will automatically refill in this field with the remote revision and log message. For other source methods, you can fill in whatever seems appropriate.

Release state
: Only packages that are public are exported, but this also has side-effects. Once the Release state is public, it is no longer possible to edit a package release on the spot. Submitting the form in that case will always create a new release.

  On edits, you will see some extra information: Synch state and rev. The first is the current status of a package release with respect to the published repository, the second is the revision number that has been assigned to this release.

Version
: This is the user-visible version field.

Source
: Here things get interesting. There are five ways to put the source of a package release into the database, as explained in the next sections.

  As previous revision
  : If you are editing an already existing package, then it is possible to re-use the uploaded source from the revision you are editing as the source for the new revision that will be created.

  File upload
  : Upload of a local archive file via CGI. Be warned that if there are other errors in your form, you will have to re-select the local file after fixing those other errors. Contrary to the other fields, local file selection is not persistent across form submits.

  HTTP URL
  : This asks the system to do a `wget` of an archive file on a specific URL, which could be either HTTP or FTP. If you need remote log-in information to access the file, please encode the user name and password in the URL, exactly as you would do when using `wget` on the command line.

  SVN URL
  : This asks the system to do a `svn checkout` on a specific URL. In this case, you may also need SVN Username and SVN Password. Also, some repositories may need anonymous as user name for anonymous access. The top-level checkout folder will be stripped away before creating the package. This is so that you can e.g. give http://foundry.supelec.fr/svn/metapost/trunk/texmf/ as URL without getting an extra directory level.

  GIT URL
  : This asks the system to do a `git clone` on a specific URL. It is very similar to SVN URL, just using a different versioning system. In this case, you may also need GIT Branch.

Please verify package contents
: The first time the edit form is loaded, this will only display a message, but after the initial submit (assuming everything else went well), it will display the full list of files that will become the source of your package.

  Please check this list carefully! TLContrib does run some tests on the package contents and will refuse to accept package sources that are horribly wrong, but it does not check the actual contents of any of the files, and of course it can not test for every possible problem.

Package transfer

It is possible for the maintainer of a package to transfer the package to another user completely. To do so, follow the Share link in your package list. See the help text in that form for details.

Package sharing

It is also possible for the maintainer of a package to share the package maintenance with other users.

To set up package sharing for a package you maintain, follow the Share link in your package list. See the help text in that form for details.

When someone else has shared a package with you, then you will see new entries in your package list. These will have the user id of the actual package

maintainer added after the Date field. You can edit such packages (and thus create new revisions), but the new revisions will become property of the actual package maintainer.

In other words: a package can only have one actual maintainer, and that maintainer is responsible for all revisions of the package. However, the maintainer can allow other users to help with the actual creation of new revisions.

Package deletion

In the list of your packages and in the view screen of one of your package releases, there are two links that delete items:

Del / Delete revision
: This link deletes a single revision of a package.

Delete package (in view screen only)
: This link removes a whole package completely, including all revisions of it.

Both links show a confirmation screen first.

Remote revision creation (advanced usage)

Once a package has been created (there must at least one revision record present already), and under the conditions that it has a source method of HTTP URL, SVN URL, or GIT URL, it is possible to submit a new revision by fetching a special URL from a remote location or script. Using this method, there is no need to be logged in at all.

The URL template looks like this:

```
http://tlcontrib.metatex.org
  /cgi-bin/package.cgi/action=notify/key=<key>
  /check=<md5>?version=<version>
```

Please note that `version` is preceded by a question mark, but everything else is separated by slashes, and, of course, the actual URL should be a single line, without any spaces. All three fields are required.

The three special fields have to be filled in like this:

`<key>`
: This is the package Id of the package.

  Let's use `luatex.i386-linux` as example value for `<key>`.

`<md5>`
: This is a constructed check-sum, created as follows: it is the hexadecimal representation of the md5 check-sum of the string created by combining your userid, your password, and the new version string, separated by slashes.

  For example, let's assume that your userid is `taco` and your password is `test`, and that the new release that you are trying to create has version `0.64.0`.

  On a Unix command line, the check-sum can be calculated like this:

```
$ echo taco/test/0.64.0 | md5sum
c704f499e086e0d54fca36fb0abc973e  -
```

  The value of `<md5>` is therefore `c704f499e086e0d54fca36fb0abc973e`.

`<version>`
: This is the version field of the new release.

  Note: if this contains spaces or other characters that cannot be used in URLs, then you either have to escape the version string in the URL, or use POST instead of GET. In any case, do not escape the version while calculating the check-sum string.

  There is no need to do any URL escaping here, so the value of `<version>` will be `0.64.0`

Using the example variables given above, the final URL that would have to be accessed is (again without line breaks or spaces):

```
http://tlcontrib.metatex.org/cgi-bin/package.cgi
  /action=notify/key=luatex.i386-linux
  /check=c704f499e086e0d54fca36fb0abc973e
  ?version=0.64.0
```

Accessing this URL will cause TLContrib to fetch the HTTP or SVN or GIT URL source in the package's top-level revision (regardless of what its publication state is), and create a new revision based on the fetched file(s) and the supplied version string. All other fields will remain exactly the same as in the original top-level revision.

This new package revision will appear in the web interface just like any other revision, there is nothing special about it other than what is already mentioned.

Final remark

TLContrib is a fairly new project, and some improvements are definitely possible, especially in the edit forms on the website. But I hope that even in the current state, it will be a useful addition to the whole TEX Live experience.

Notes

1. The website for TLContrib is http://tlcontrib.metatex.org/
2. The mailman page for the mailing list is http://www.ntg.nl/cgi-bin/mailman/listinfo/tlcontrib
3. See http://www.tug.org/tds/tds.html for a detailed description of the current TEX Directory Structure specification.

Taco Hoekwater
tlcontrib@metatex.org

Nieuws van CTAN

Een uittreksel uit de recente bijdragen in het CTAN archief

Abstract
Dit artikel beschrijft een aantal recente bijdragen uit het CTAN archief (en andere bronnen op het Internet). De selectie is gebaseerd op wat ik zelf interessant vind en wat ik denk dat voor veel anderen interessant is. Het is dus een persoonlijke keuze. Het heeft niet de bedoeling om een volledig overzicht te geven.

Keywords
TEX, LaTEX, packages, CTAN, bibliografie, biber, biblatex.

Inleiding

In juli 2010 is een nieuwe TEXLive 2010 distributie uitgekomen. Ik zit momenteel in Bolivia, waar ik geen TEXLive DVD's krijg. Ik heb hem (in de vorm van een MacTEX installer voor mijn MacBook) gedownload over een 256kbps internetverbinding. Als ik me goed herinner deed hij er meer dan 24 uur over. Gelukkig is de download herstartbaar waarbij hij verder gaat op het punt waar hij gebleven is, anders zou het niet gelukt zijn.

De TEXLive installaties hebben tegenwoordig een programma om pakketten automatisch te updaten, net als MiKTEX dat heeft. Helaas was het niet mogelijk om automatisch te updaten van TEXLive 2009 naar TEXLive 2010. Hopelijk komt dit ook een keer in een toekomstige versie. Als je namelijk je installatie regelmatig laat updaten heb je natuurlijk bij de volgende versie een groot gedeelte al geïnstalleerd.

In dit licht is het eigenlijk niet zo interessant om een lijst van nieuwe of bijgewerkte pakketten te geven. Daarom beperk ik me maar tot het beschrijven van interessante ontwikkelingen. De meeste van de hieronder beschreven pakketten zijn onderdeel van de TEXLive en MiKTEX distributies.

Biblatex

Biblatex is een pakket dat de functionaliteit van BibTEX voor het grootste gedeelte implementeert in LaTEX. Het maken van BibTEX stijlen is erg lastig omdat BibTEX een programmeertaaltje gebruikt dat gebaseerd is op operaties op een stack, een techniek die de meeste mensen niet beheersen, en die bovendien gemakkelijk tot fouten leidt. Het implementeren van deze stijlen in LaTEX zou gemakkelijker moeten zijn. Bij het gebruik van biblatex wordt BibTEX nog steeds gebruikt maar alleen voor het sorteren en het genereren van labels. Voor de rest, het formatteren van de bibliografische items worden LaTEX macro's gebruikt. Hierdoor zou het voor meer mensen mogelijk moeten zijn om bibliografische stijlen te ontwikkelen. Misschien is het ontwikkelen van een compleet nieuwe stijl niet voor iedereen weggelegd, het aanpassen van een stijl zal in ieder geval een stuk makkelijker zijn. Ook alle commando's voor citaties kunnen gemakkelijk worden aangepast.

Biblatex is niet compatibel met een groot aantal pakketten die iets met bibliografieën doen, zoals babelbib, bibtopic, bibunits, chapterbib, multibib en meer. Maar de functionaliteit van de hier genoemde pakketten is in biblatex aanwezig. In ieder geval ondersteunt biblatex opgesplitste bibliografieën (bijvoorbeeld op topic), meerdere bibliografieën in een document, bibliografieën per hoofdstuk, sectie en dergelijke.

Ook is biblatex niet compatibel met het pakket ucs, zodat gebruikers van een aantal talen met een niet-latijns schrift helaas dit niet kunnen gebruiken, tenzij ze gebruik maken van XeTEX. Overigens is het gebruik van BibTEX met dit soort schriften ook problematisch. Verder is biblatex wel gelocaliseerd (dat wil zeggen dat het verschillende talen ondersteunt) en kan het gebruik maken van het babel pakket.

Jurabib en natbib zijn ook incompatibel met biblatex en de functionaliteit van deze pakketten is slechts gedeeltelijk aanwezig in biblatex zelf (maar zie ook verderop).

Hierbij een voorbeeld waarbij ik eerst een document met natbib en de klassieke BibTEX oplossing geef en daarna de oplossing met biblatex. De BibTEX entry is:

```
@Book{Date2003,
 author = {Date, C. J.},
 title = {An Introduction to Database Systems},
 publisher =
      {Addison-Wesley Publishing Company Inc.},
 year = {2003},
 address = {Reading, Massachusetts},
```

```
 edition = {8}
}
```

Het input-document, gevolgd door de output:

```
\documentclass{article}
\pagestyle{empty}
\setlength{\textwidth}{220pt}
\usepackage[round]{natbib}
\begin{document}
This is a very short article.
It cites the classical databases book
\cite{Date2003}.

\bibliographystyle{plainnat}
\bibliography{bibfile}
\end{document}
```

This is a very short article. It cites the classical databases book Date (2003).

References

C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 8 edition, 2003.

En dan nu de biblatex-versie:

```
\documentclass{article}
\pagestyle{empty}
\setlength{\textwidth}{220pt}
\usepackage[natbib, style=authoryear]{biblatex}
\bibliography{bibfile}
\begin{document}
This is a very short article.
It cites the classical databases book
\cite{Date2003}.

\printbibliography
\end{document}
```

This is a very short article. It cites the classical databases book Date, 2003.

References

Date, C. J. (2003). *An Introduction to Database Systems*. 8th ed. Reading, Massachusetts: Addison-Wesley Publishing Company Inc.

Er zijn een paar verschillen te constateren:

- Ten eerste: in de LaTeX-input staat het `\bibliography`-commando in de preamble en niet meer op de plaats waar de bibliografie komt. Op die plaats staat het `\printbibliography`-commando.
- Er is geen `\bibliographystyle` meer. Inplaats daarvan worden er opties van het biblatex-pakket gebruikt.
- In de citatie ontbreken de haakjes om het jaartal. Dit is op te lossen door in plaats van `\cite` het commando `\textcite` te gebruiken.
- Het jaartal in de bibliografie staat op een andere plaats.

Het omzetten van `\cite` in `\textcite` is vervelend. Misschien zijn er nog opties te vinden om de layout meer zoals in natbib te krijgen. Het aantal opties in biblatex is echter gigantisch en dat zal dus nog wel wat extra zoekwerk kosten. Hier is de versie met `\textcite`:

```
\documentclass{article}
\pagestyle{empty}
\setlength{\textwidth}{220pt}
\usepackage[natbib, style=authoryear]{biblatex}
\bibliography{bibfile}
\begin{document}
This is a very short article.
It cites the classical databases book
\textcite{Date2003}.

\printbibliography
\end{document}
```

This is a very short article. It cites the classical databases book Date (2003).

References

Date, C. J. (2003). *An Introduction to Database Systems*. 8th ed. Reading, Massachusetts: Addison-Wesley Publishing Company Inc.

Biblatex heeft voor het functioneren eTeX nodig maar dit is tegenwoordig meestal de standaard TeX machine die gebruikt wordt.

Biblatex is nu op versie 1.0, de vorige versies waren allemaal 0.x, dus we mogen aannemen dat het een zekere volwassenheid heeft bereikt. Het is geschreven door Philipp Lehman, bekend van de handleiding voor het installeren van LaTeX fonts. De handleiding van biblatex is zeer uitgebreid: ongeveer 200 pagina's. Het zou wel handig zijn als er een mini-handleiding zou zijn voor mensen die alleen maar een bibliografie wil-

len gebruiken en niet zelf bibliografische stijlen willen ontwikkelen.

Er zijn ook een aantal uitbreidingspakketten voor biblatex. Deze worden niet met \usepackage gebruikt maar door middel van opties in biblatex.

biblatex-dw
Biblatex-dw is een pakket geschreven door Dominik Wassenhoven, in eerste instantie voor eigen gebruik. Het implementeert de citatie-stijl die gebruikelijk is in de humaniora. In feite zijn er twee stijlen in verwerkt:

Een stijl speciaal voor verwijzingen in voetnoten (footnote-dw). Hierbij wordt de referentie als voetnoot gezet met dezelfde informatie die ook in de bibliografie staat.

```
\usepackage[style=footnote-dw]{biblatex}
```

[1]C. J. Date: An Introduction to Database Systems, 8th ed., Reading, Massachusetts 2003.

Auteur-titel-stijl (authortitle-dw).

```
\usepackage[style=authortitle-dw]{biblatex}
```

Zie het voorbeeld hierna.

This is a very short article. It cites the classical databases book Date: An Introduction to Database Systems.

References

Date, C. J.: An Introduction to Database Systems, 8th ed., Reading, Massachusetts 2003.

Beide stijlen zijn ook mogelijk met het basispakket biblatex, maar biblatex-dw heeft veel meer opties.

biblatex-chem
Biblatex-chem bevat 4 stijlen voor chemici, namelijk

- chem-acs: *American Chemical Society*
- chem-angew: *Angewandte Chemie*
- chem-biochem: *Biochemistry*
- chem-rsc: *Royal Society of Chemistry*

Maar ook andere chemische publicaties vallen hieronder. Het pakket is geschreven door Joseph Wright.

biblatex-nature
Dit pakket implementeert de citatiestijl voor het tijdschrift *Nature*. Hierbij worden superscripts gebruikt voor de citatie, echter zonder dat er een voetnoot bijhoort. Hiervoor moet wel het commando \autocite gebruikt worden in plaats van \cite, anders wordt het nummer tussen vierkante haken gezet. Het pakket is ook geschreven door Joseph Wright.

```
\documentclass{article}
\pagestyle{empty}
\setlength{\textwidth}{220pt}
\usepackage[natbib=true, style=nature]{biblatex}
\bibliography{bibfile}
\begin{document}
This is a very short article.
It cites the classical databases book
\autocite{Date2003}.

\printbibliography
\end{document}
```

This is a very short article. It cites the classical databases book[1].

References

1. Date, C. J. *An Introduction to Database Systems* 8th ed. (Addison-Wesley Publishing Company Inc., 2003).

biblatex-science
Het begint een beetje saai te worden maar voor het tijdschrift *Science* is er het pakket biblatex-science, eveneens door Joseph Wright. Citaties worden als nummers met ronde haakjes afgedrukt.

biblatex-apa
Biblex-apa implementeert de citatie- en bibliografiestijl voor de APA (American Psychological Association). De implementator, Philip Kime, schrijft dat het nogal een klus was omdat de layoutregels van de APA zo'n 60 pagina's beslaan en het hem niet gelukt is om ze allemaal te implementeren. Er zijn ook een groot aantal localisaties aanwezig voor degenen in andere landen die een vergelijkbare stijl moeten gebruiken. Alleen werkt deze stijl op dit moment niet met de nieuwste versie van biblatex.

biblatex-chicago
Biblatex-chicago bevat stijlen volgens het beroemde '*Chicago Manual of Style*'. Deze kent twee stijlen: een auteur-datum-stijl voor gebruik in de natuurwetenschappen en een voetnoot-stijl voor gebruik in de humaniora. Het pakket is geschreven door David Fussner. Het verkeert nog in het bètastadium maar is al erg uit-

gebreid. De handleiding beslaat 95 pagina's.

Bij dit pakket is het mogelijk om in plaats van de

```
\usepackage[biblatex]
```

met de nodige opties het pakket zelf aan te roepen door middel van:

```
\usepackage[authordate]{biblatex-chicago}
of:
\usepackage[notes]{biblatex-chicago}
```

Op deze manier worden meer opties automatisch gezet dan met de aanroep van het pakket biblatex.

De optie 'authordate' heeft nog het meeste weg van wat we in het eerste voorbeeld met natbib en de traditionele BibTEX route hebben geproduceerd. Er is zelfs een optie 'natbib' die meegegeven kan worden. De optie 'notes' lijkt nog het meest op de stijl footnote-dw die we eerder gezien hebben.

Er zijn ook localisaties voor Duits en Frans.

```
\documentclass{article}
\pagestyle{empty}
\setlength{\textwidth}{220pt}
\usepackage[natbib,authordate]{biblatex-chicago}
\bibliography{bibfile}
\begin{document}
This is a very short article.
It cites the classical databases book
\textcite{Date2003}.

\printbibliography
\end{document}
```

This is a very short article. It cites the classical databases book Date (2003).

References

Date, C. J. 2003. *An introduction to database systems*. 8th ed. Reading, Massachusetts: Addison-Wesley Publishing Company Inc.

---

biblatex-historian

Deze stijl, geschreven door Sander Glibof, is een aangepaste versie van de chicago stijl voor historici, omdat deze vaak voetnoten gebruiken om te refereren, niet alleen naar boeken en artikelen maar ook naar herdrukken, correspondentie, archieven, ongepubliceerde manuscripten en dergelijke. Het wordt voornamelijk gebruikt met voetnoten als referenties maar er zijn ook speciale commando's om referenties in voetnoten te gebruiken.

De stijl wordt beschreven in het boek van Kate L. Turabian, '*A Manual for Writers of Research Papers, Theses, and Dissertations: Chicago Style for Students and Researchers*, 7th ed. (Chicago and London: University of Chicago Press, 2007)'. De stijl is erg uitgebreid, de handleiding heeft 84 pagina's waarvan een groot deel bestaat uit de bespreking van de verschillende soorten documenten waarnaar gerefereerd kan worden.

biblatex-philosophy

Deze stijl van Ivan Valbusa is voor referenties in het vakgebed filosofie. Helaas is de handleiding in het Italiaans met slechts een korte README tekst in het Engels. Er zijn 3 stijlen: klassiek, modern en verbose. Citaties zijn door middel van voetnoten. De eerste twee stijlen zijn gebaseerd op auteur-jaar stijl (de voetnoot bevat auteur en jaar). Bij 'verbose' worden alle bibliografische gegevens opgenomen in de voetnoot. Met de optie 'backref' worden in de bibliografie bij een document terugverwijzingen geplaatst naar de plaatsen in de tekst waar het document geciteerd wordt. Het pakket wordt gebruikt door middel van een van de volgende commando's:

```
\usepackage[style=philosophy-classic]{biblatex}
\usepackage[style=philosophy-modern]{biblatex}
\usepackage[style=philosophy-verbose]{biblatex}
```

Het bijzondere van dit pakket is dat er voorzieningen zijn om vertalingen van werken of herdrukken tegelijk met het origineel op te nemen in de referenties. Er zijn voorzieningen voor Engels en Italiaans waarbij het gemakkelijk is om andere talen toe te voegen.

biblatex-mla

Het pakket biblatex-mla van James Clawson ondersteunt citaties volgens de richtlijnen van de Modern Language Association (MLA). Het bijzondere van deze stijl is dat alleen auteur en paginanummers gerefereerd worden, tenzij er van een auteur meerdere werken geciteerd worden. In dat geval wordt ook de titel erbij gevoegd.

Biblatex-mla is echter nog niet aangepast aan de nieuwste versie van biblatex. Het is ook geen onderdeel van TEXLive, maar wel van MiKTEX. Het is te vinden op `http://konx.net/biblatex-mla/` en op CTAN in `macros/latex/contrib/biblatex-contrib/biblatex-mla.zip`.

biblatex-jura

Een stijl van Ben E. Hard voor de Duitse juridische stijl van citeren volgens voorschriften van de uitgever Nomos. Ook dit pakket is geen onderdeel van TEXLive, maar wel van MiKTEX. De documentatie bestaat slechts uit een README file. Hieruit is wel duidelijk dat dit geen vervanging is van het uitgebreide jurabib pakket voor het normale BibTEX gebruik. Het pakket wordt gebruikt door middel van het commando:

```
\usepackage[style=biblatex-jura]{biblatex}
```

Opmerkelijk dat de stijl hier 'biblatex-jura' heet, terwijl de andere stijlen het gedeelte 'biblatex-' weglaten. Citaties worden met `\footcite` gedaan inplaats van met `\cite`.

Bibfile wijzigingen

Sommige opties van biblatex of een van de stijlen werken pas optimaal als de bibfile aangepast wordt.

Programma

Biber

Biber is een alternatief voor BibTEX voor gebruik bij biblatex. Het is geschreven in Perl en kan op machines waarop Perl aanwezig is redelijk gemakkelijk geïnstalleerd worden. Op de site `http://biblatex-biber.sourceforge.net/` zijn bovendien executables te downloaden voor Linux, Mac OS X en Windows. Het is niet opgenomen in TEXLive en MiKTEX. Bij gebruik van biber in plaats van BibTEX moet de extra optie 'backend=biber' meegegeven worden aan het biblatex-pakket. De voordelen van biber zijn dat het Unicode ondersteunt, geen kunstmatige limieten heeft zoals BibTEX, en dat het opgesplitste en meervoudige bibliografieën in één keer kan verwerken.

Het programma is geschreven door François Charette and Philip Kime.

CTAN: `biblio/biber/`

Piet van Oostrum
http://www.pietvanoostrum.com
piet@vanoostrum.org

Up to ConTEXt MkVI

Introduction

No, this is not a typo: MkVI is the name of upcoming functionality but with an experimental character. It is also a playground. Therefore this is not the final story.

Defining macros

When you define macros in TEX, you use the # to indicate variables. So, you code can end up with the following:

```
\def\MyTest#1#2#3#4%
  {\dontleavehmode
   \dostepwiserecurse{#1}{#2}{#3}
     {\ifnum\recurselevel>#1 \space,\fi
      \recurselevel: #4\space}%
   .\par}
```

This macro is called with 4 arguments:

```
\MyTest{3}{8}{1}{Hi}
```

However, using numbers as variable identifiers might not have your preference. It makes perfect sense if you keep in mind that TEX supports delimited arguments using arbitrary characters. But in practice, and especially in ConTEXt we use only a few well defined variants. This is why you can also imagine:

```
\def\MyTest#first#last#step#text%
  {\dontleavehmode
   \dostepwiserecurse{#first}{#last}{#step}
     {\ifnum\recurselevel>#first \space,\fi
      \recurselevel: #text}%
   .\par}
```

In order for this to work, you need to give your file the suffix `mkvi` or you need to put a directive on the first line:

```
% macros=mkvi
```

You can of course use delimited arguments as well, given that the delimiters are not letters.

```
\def\TestOne[#1]%
  {this is: #1}

\def\TestTwo#some%
  {this is: #some}

\def\TestThree[#whatever][#more]%
  {this is: #more and #whatever}

\def\TestFour[#one]#two%
  {\def\TestFive[#alpha][#one]%
     {#one, #two, #alpha}}
```

You can also use the following variant which is already present for a while but not that much advertised. This method ignores all spaces in definitions so if you need one, you have to use `\space`.

```
\starttexdefinition TestSix #oeps

  here: #oeps

\stoptexdefinition
```

These commands work as expected:

```
\startlines
  \TestOne  [one]
  \TestTwo  {one}
  \TestThree[one][two]
  \TestFour [one]{two}
  \TestFive [one][two]
  \TestSix  {one}
\stoplines
```

this is: one
this is: one
this is: two and one
two, two, one
here: one

You can use buffers to collect definitions. In that case you can force preprocessing of the buffer with `\mkvibuffer[name]`.

Implementation

This functionality is not hard coded in the LuaTEX engine as this is not needed at all. We just preprocess the file before it gets loaded and this is something that is relatively easy to implement. Already early in the development of LuaTEX we have decided that instead of hard coding solutions, opening up makes more sense. One of the first mechanisms that were opened up was

file IO. This means that when a file is opened, you can decide to intercept lines and process them before passing them to the traditional built in input parser. The user can be completely unaware of this. In fact, as LuaTEX only accepts UTF-8, preprocessing will likely happen already when other input encodings are used. The following helper functions are available:

```
local result = resolvers.macros.preprocessed(str)
```

This function returns a string with all named parameters replaced.

```
resolvers.macros.convertfile(oldname,newname)
```

This function converts a file into a new one.

```
local result =
    resolvers.macros.processmkvi(str,filename)
```

This function converts the string but only if the suffix of the filename is `mkvi` or when the first line of the string is a comment line containing `macros=mkvi`. Otherwise the original string is returned. The filename is optional.

A few details

Imagine that you want to do this:

```
\def\test#1{before#1after}
```

When we use names this could look like:

```
\def\test#inbetween{before#inbetweenafter}
```

and that is not going to work out well. We could be more liberal with spaces, like

```
\def\test #inbetween {before #inbetween after}
```

but then getting spaces in the output before or after variables would get more complex. However, there is a way out:

```
\def\test#inbetween{before#{inbetween}after}
```

As the sequence `#{` has a rather low probablility of showing up in a TEX source file, this kind of escaping is part of the game. So, all the following cases are valid:

```
\def\test#oeps{... #oeps ...}
\def\test#oeps{... #{oeps} ...}
\def\test#{main:oeps}{... #{main:oeps} ...}
\def\test#{oeps:1}{... #{oeps:1} ...}
\def\test#{oeps}{... #oeps ...}
```

When you use the braced variant, all characters except braces are acceptable as name, optherwise only lowercase and uppercase characters are permitted.
Normally LuaTEX uses a couple of special tokens like ^ and _. In a macro definition file you can avoid these by using primitives:

```
&   \aligntab
#   \alignmark
^   \Usuperscript
_   \Usubscript
$   \Ustartmath
$   \Ustopmath
$$  \Ustartdisplaymath
$$  \Ustopdisplaymath
```

Especially the `alignmark` is worth noticing: using that one directly in a macro definition can result in unwanted replacements, depending on whether a match can be found. In practice the following works out well

```
\def\test#oeps{test:#oeps
               \halign{##\cr #oeps\cr}}
```

You can use UTF-8 characters as well. For practical reasons this is only possible with the braced variant.

```
\def\blä#{blá}{blà:#{blá}}
```

There will probably be more features in future versions but each of them needs careful consideration in order to prevent interferences.

Utilities

There is currently one utility (or in fact an option to an existing utility):

```
mtxrun --script interface
       --preprocess whatever.mkvi
```

This will convert the given file(s) to new ones, with the default suffix `tex`. Existing files will not be overwritten unless `---force` is given. You can also force another suffix:

```
mtxrun --script interface
       --preprocess whatever.mkvi
       --suffix=mkiv
```

A rather plain module `luatex-preprocessor.lua` is provided for other usage. That variant provides a somewhat simplified version.
Given that you have a `luatex-plain` format you can run:

```
luatex --fmt=luatex-plain
       luatex-preprocessor-test.tex
```

Such a plain format can be made with:

```
luatex --ini luatex-plain
```

You probably need to move the format to a proper location in your TEX tree.

Hans Hagen

```
if not modules then modules = { } end modules ['luat-mac'] = {
    version   = 1.001,
    comment   = "companion to luat-lib.mkiv",
    author    = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
    copyright = "PRAGMA ADE / ConTeXt Development Team",
    license   = "see context related readme files"
}

local P, V, S, R, C, Cs, Cmt = lpeg.P, lpeg.V, lpeg.S, lpeg.R, lpeg.C, lpeg.Cs, lpeg.Cmt
local lpegmatch, patterns = lpeg.match, lpeg.patterns

local insert, remove = table.insert, table.remove
local rep, sub = string.rep, string.sub
local setmetatable = setmetatable

local report_macros = logs.new("macros")

local stack, top, n, hashes = { }, nil, 0, { }

local function set(s)
    if top then
        n = n + 1
        if n > 9 then
            report_macros("number of arguments > 9, ignoring %s",s)
        else
            local ns = #stack
            local h = hashes[ns]
            if not h then
                h = rep("#",ns)
                hashes[ns] = h
            end
            m = h .. n
            top[s] = m
            return m
        end
    end
end

local function get(s)
    local m = top and top[s] or s
    return m
end

local function push()
    top = { }
    n = 0
    local s = stack[#stack]
    if s then
        setmetatable(top,{ __index = s })
    end
    insert(stack,top)
end

local function pop()
    top = remove(stack)
end

local leftbrace   = P("{")   -- will be in patterns
local rightbrace  = P("}")
```

```
local escape      = P("\\")

local space       = patterns.space
local spaces      = space^1
local newline     = patterns.newline
local nobrace     = 1 - leftbrace - rightbrace

local longleft    = leftbrace  -- P("(")
local longright   = rightbrace -- P(")")
local nolong      = 1 - longleft - longright

local name        = R("AZ","az")^1 -- @?! -- utf?
local longname    = (longleft/"") * (nolong^1) * (longright/"")
local variable    = P("#") * Cs(name + longname)
local escapedname = escape * name
local definer     = escape * (P("def") + P("egdx") * P("def"))
local startcode   = P("\\starttexdefinition")
local stopcode    = P("\\stoptexdefinition")
local anything    = patterns.anything
local always      = patterns.alwaysmatched

local pushlocal   = always   / push
local poplocal    = always   / pop
local declaration = variable / set
local identifier  = variable / get

local function matcherror(str,pos)
    report_macros("runaway definition at: %s",sub(str,pos-30,pos))
end

local grammar = { "converter",
    texcode     = pushlocal
                * startcode
                * spaces
                * name
                * spaces
                * (declaration + (1 - newline - space))^0
                * V("texbody")
                * stopcode
                * poplocal,
    texbody     = (   V("definition")
                    + identifier
                    + V("braced")
                    + (1 - stopcode)
                  )^0,
    definition  = pushlocal
                * definer
                * escapedname
                * (declaration + (1-leftbrace))^0
                * V("braced")
                * poplocal,
    braced      = leftbrace
                * (   V("definition")
                    + identifier
                    + V("texcode")
                    + V("braced")
                    + nobrace
                  )^0
```

```
            -- * rightbrace^-1, -- the -1 catches errors
               * (rightbrace + Cmt(always,matcherror)),

    pattern     = V("definition") + V("texcode") + anything,

    converter   = V("pattern")^1,
}

local parser = Cs(grammar)

local checker = P("%") * (1 - newline - P("macros"))^0
             * P("macros") * space^0 * P("=") * space^0 * C(patterns.letter^1)

-- maybe namespace

local macros = { } resolvers.macros = macros

function macros.preprocessed(str)
    return lpegmatch(parser,str)
end

function macros.convertfile(oldname,newname) -- beware, no testing on oldname == newname
    local data = resolvers.loadtexfile(oldname)
    data = interfaces.preprocessed(data) or ""
    io.savedata(newname,data)
end

function macros.version(data)
    return lpegmatch(checker,data)
end

function macros.processmkvi(str,filename)
    if (filename and file.suffix(filename) == "mkvi") or lpegmatch(checker,str) == "mkvi" then
        return lpegmatch(parser,str) or str
    else
        return str
    end
end

if resolvers.schemes then

    local function handler(protocol,name,cachename)
        local hashed = url.hashed(name)
        local path = hashed.path
        if path and path ~= "" then
            local data = resolvers.loadtexfile(path)
            data = lpegmatch(parser,data) or ""
            io.savedata(cachename,data)
        end
        return cachename
    end

    resolvers.schemes.install('mkvi',handler,1) -- this will cache !

    utilities.sequencers.appendaction(resolvers.openers.helpers.textfileactions,
                                      "system","resolvers.macros.processmkvi")

end
```

LuaTEX 0.60

Abstract
TEXLive 2010 will contain LuaTEX 0.60. This article gives an overview of the changes between this version and the version on last year's TEXLive.
Highlights of this release: cweb code base, dynamic loading of lua modules, various font subsystem improvements including support for Apple `.dfont` font collection files, braced input file names, extended pdf Lua table, and access to the line breaking algorithm from Lua code.

General changes

Some of the changes can be organised into sections, but not all. So first, here are the changes that are more or less standalone.

- Many of the source files have been converted into cweb. Early versions of LuaTEX were based on Pascal web, but by 0.40 all code was hand-converted to C. The literate programming comments were kept, and the relevant sources have now been converted back into cweb, reinstating the literate documentation.

  This change does not make LuaTEX a literate program in the traditional sense because the typical C source code layout with pairs of header & implementation files has been kept and no code reshuffling takes place. But it does mean that it is now much easier to keep the source documentation up-to-date, and it is possible to create nicely typeset program listings with indices.
- There are now source repository revision numbers in the banner again, which is a useful thing to have while tracking down bugs. For example, the LuaTEX binary being used to write this article starts up with:

```
This is LuaTeX, Version beta-0.60.1-2010042817 (rev 3659)
```

- The horizontal nodes that are added during line breaking now inherit the attributes from the nodes inside the created line.

  Previously, these nodes (`\leftskip` and `\rightskip` in particular) inherited the attributes in effect at the end of the (partial) paragraph because that is where line breaking takes place.
- All Lua errors now report file and line numbers to aid in debugging, even if the error happens inside a callback.
- LuaTEX can now use the embedded kpathsea library to find Lua `require()` files, and will do so by default if the kpathsea library is enabled by the format (as is the case in plain LuaTEX and the various LuaLaTEX formats).
- The print precision for small numbers in Lua code (the return value of `tostring()`) has been improved.
- Of course there were lots of code cleanups and improvements to the reference manual.

Embedded libraries and other third-party inclusions

The following are changes to third-party code that for the most part should not need much explanation.

- MetaPost is now at version 1.211.
- Libpng is now at version 1.2.40.
- New synctex code is imported from TEXLive.
- The Lua source file from the luamd5 library (which provides the `md5.hexsuma` function) is now embedded in the executable. In older versions of LuaTEX, this file was missing completely.
- The Lua co-routine patch (coco) is now disabled on linux powerpc because it caused crashes on that platform due to a bad upstream implementation.

Dynamic loading of lua modules

LuaTEX now has support for dynamic loading of external compiled Lua libraries.

As with other `require()` files, LuaTEX can and will use kpathsea if the format allows it to do so. For this purpose, kpathsea has been extended with a new file type: `clua`. The associated `texmf.cnf` variable is defined like this by default:

```
CLUAINPUTS=.:$SELFAUTOLOC/lib/{$progname,$engine,}/lua//
```

which means that if your LuaTEX binary lives in

```
/opt/tex/texmf-linux-64/bin/
```

then your compiled Lua modules should go into the local directory, or in a tree below

```
/opt/tex/texmf-linux-64/bin/lib/lua
```

Be warned that not all available Lua modules will work. LuaTEX is a command line program, and on some platforms that makes it near impossible to use GUI-based extensions.

Font related

Lots of small changes have taken place in the font processing.

- The backend message

  ```
  cannot open Type 1 font file for reading
  ```

  now reports the name of the Type1 font file it was looking for.
- It is no longer possible for fonts from included pdf files to be replaced by / merged with the document fonts of the enveloping pdf.
- Support for Type3 `.pgc` files has been removed. This is just for the `.pgc` format invented by Hàn Thế Thành, bitmapped pk files still work.
- For TrueType font collections (`.ttc` files), now the used sub-font name and its index id are printed to the terminal, and if the backend cannot find the font in the collection, the run is aborted.
- It is now possible to use Apple `.dfont` font collection files. Unfortunately, in Snow Leopard (a.k.a. MacOSX 10.6) Apple switched to a `.ttc` format that is not quite compatible with the Microsoft version of `.ttc`. As a result, the system fonts from Snow Leopard cannot be used in LuaTEX 0.60 yet.
- The loading speed of large fonts via the `fontloader` library, and the inclusion speed for sub-setting in the backend have both been improved.
- Two new MathConstants entries have been added. Suppose the Lua math font loading code produces a Lua table named `f`, then in that table, you can set

  ```
  f.MathConstants.FractionDelimiterSize
  ```

```
f.Mathconstants.FractionDelimiterDisplayStyleSize
```

These new fields allow proper setting of the size parameters for LuaTEX's `...withdelims` math primitives, for which there is no ready replacement in the OpenType MATH table.

- Artificially slanted or extended fonts now work via the pdf text matrix so that this now also works for non-Type1 fonts. In other words: the Lua `f.slant` and `f.extend` font keys are now obeyed in all cases.
- There is another new allowed key: `f.psname`. When set, this value should be the original PostScript font name of the font. In the pdf generation backend, fonts inside `.dfont` and `.ttc` collections are fetched from the archive using this field, so in those cases the key is required.
- A related change is made to the font name discovery used by the backend for storage into the pdf file structure: now it tries `f.psname` first, as that is much less likely to contain spaces than `f.fontname` (which the field that 0.40 used). If there is no `f.psname`, it falls back to the old behaviour.
- Finally, Lua-loaded fonts now support a `f.nomath` key to speed up loading the Lua table in the normal case of fonts that do not provide OpenType MATH data.

'TEX'-side extensions and changes

LuaTEX is not actually TEX even though it uses an input language that is very similar, hence the quotes in this section's title. Some of the following items are new LuaTEX extensions, others are adjustments to pre-existing pdfTEX or Aleph functionality.

- The primitives `\input` and `\openin` now accept braced file names, removing the need for double quote escapes in case of files with spaces in their name.
- The `\endlinechar` can now be set to any value between 0 and 127.
- The new primitives `\aligntab` and `\alignmark` are aliases for the characters with the category codes of `&` and `#` in alignments, respectively.
- `\latelua` is now allowed inside leaders. To be used with care, because the lua code will be executed once for each generated leader item.
- The new primitive `\gleaders` provides 'globally aligned' leaders. These leaders are aligned on one side of the main output box instead of to the side of the immediately enclosing box.
- From now on LuaTEX handles only 4 direction specifiers:
  `TLT` (latin),
  `TRT` (arabic),
  `RTT` (cjk), and
  `LTL` (mongolian).
  Other direction specifiers generate an error.
- The `\pdfcompresslevel` is now effectively fixed as soon as any output to the pdf file has occurred.
- `\pdfobj` has gained an extra optional keyword: `uncompressed`. This forces the object to be written to the pdf in plain text, which is needed for certain objects containing meta data.
- Two new token lists are provided: `\pdfxformresources` and `\pdfxformattr`, as an alternative to `\pdfxform` keywords.
- The new syntax

  ```
  \pdfrefxform [width <dimen>] [height <dimen>] [depth <dimen>] <formref>
  ```

  scales a single form object; using similar principle as with `\pdfximage`: depth alone doesn't scale, it shifts vertically.

- Similarly,

```
\pdfrefximage [width <dimen>] [height <dimen>] [depth <dimen>] <imageref>
```

  overrules settings from `\pdfximage` for this image only.
- The following obsolete pdfTeX primitives have been removed:
  `\pdfoptionalwaysusepdfpagebox`
  `\pdfoptionpdfinclusionerrorlevel`
  `\pdfforcepagebox`
  `\pdfmovechars`
  These were already deprecated in pdfTeX itself.

Lua table extensions

In most of the Lua tables that LuaTeX provides, only small changes have taken place, so they do not deserve their own subsections.

- There is a new callback: `process_output_buffer`, for post-processing of `\write` text to a file.
- The callbacks `hpack_filter`, `vpack_filter` and `pre_output_filter` pass on an extra string argument for the current direction.
- `fontloader.open()` previously cleared some of the font name strings during load that it should not do.
- The new function `font.id("tenrm")` returns the internal id number for that font. It takes a bare control sequence name as argument.
- The `os.name` variable now knows about `cygwin` and `kfreebsd`.
- `lfs.readlink("file")` returns the content of a symbolic link (unix only). This extension is intended for use in `texlua` scripts.
- `lfs.shortname("file")` returns the short (FAT) name of a file (windows only). This extension is intended for use in `texlua` scripts.
- `kpse.version()` returns the kpathsea version string.
- `kpse.lookup({...})` offers a search interface similar to the kpsewhich program, an example call looks like this:

```
kpse.set_program_name('luatex')
print(kpse.lookup ('plain.tex',
                   { ["format"] = "tex",
     ["all"] = true,
     ["must-exist"] = true }))
```

The 'node' table

In the verbatim code below, `n` stands for a userdata node object.

- `node.vpack(n)` packs a list into a vlist node, like `\vbox`.
- `node.protrusion_skippable(n)` returns `true` if this node can be skipped for the purpose of protrusion discovery.
  This is useful if you want to (re)calculate protrusion in pure Lua.
- `node.dimensions(n)` returns the natural width, height and depth of a (horizontal) node list.
- `node.tail(n)` returns the tail node of a node list.
- Each glyph node now has three new virtual read-only fields: `width`, `height`, and `depth`. The values are the number of scaled points.
- `glue_spec` nodes now have an extra boolean read-only field: `writable`.
  Some glue specifications can be altered directly, but certain key glue specifications are shared among many nodes. Altering the values of those is prohibited because

it would have unpredictable side-effects. For those cases, a copy must be made and assigned to the parent node.
- hlist nodes now have a subtype to distinguish between hlists generated by the paragraph breaking, explicit `\hbox` commands, and other sources.
- `node.copy_list(n)` now allows a second argument. This argument can be used to copy only part of a node list.
- `node.hpack(n)` now accepts `cal_expand_ratio` and `subst_ex_font` modifiers. This feature helps the implementation of font expansion in a pure Lua paragraph breaking code.
- `node.hpack(n)` and `node.vpack(n)` now also return the 'badness' of the created box, and accept an optional direction argument.

The 'pdf' table

- The new functions `pdf.mapfile("...")` and `pdf.mapline("...")` are aliases for the corresponding pdfTEX primitives.
- `pdf.registerannot()` reserves a pdf object number and returns it.
- The functions `pdf.obj(...)`, `pdf.immediateobj(...)`, and `pdf.reserveobj(...)` are similar to the corresponding pdfTEX primitives. Full syntax details can be read in the LuaTEX reference manual.
- New read-write string keys:
  `pdf.catalog` string goes into the Catalog dictionary.
  `pdf.info` string goes into the Info dictionary.
  `pdf.names` string goes into the Names dictionary. referenced by the Catalog object.
  `pdf.trailer` string goes into the Trailer dictionary.
  `pdf.pageattributes` string goes into the Page dictionary.
  `pdf.pageresources` string goes into the Resources dictionary referenced by the Page object.
  `pdf.pagesattributes` string goes into the Pages dictionary.

The 'tex' table

Finally, there are some extensions to the `tex` table that are worth mentioning.

- `tex.badness(f,s)` interfaces to the 'badness' internal function.
  Accidentally, this disables access to the `\badness` internal parameter. This will be corrected in a future LuaTEX version.
- `tex.sp("1in")` converts Lua-style string units to scaled points.
- `tex.tprint({...},{...})` is like a sequence of `tex.sprint(...)` calls.
- `tex.shipout(n)` ships out a constructed box.
- `tex.nest[]` and `tex.nest.ptr` together allow read-write access to the semantic nest (mode nesting).
  For example, this prints the equivalent of `\prevdepth` at the current mode nesting level.

```
print (tex.nest[tex.nest.ptr].prevdepth)
```

  `tex.nest.ptr` is the current level, and lower numbers are enclosing modes.
  Each of the items in the `tex.nest` array represents a mode nesting level and has a set of virtual keys that be accessed both for reading and writing, but you cannot

change the actual `tex.nest` array itself. The possible keys are listed in the LuaTEX reference manual.

- `tex.linebreak(n, {...})` allows running the paragraph breaker from pure Lua. The second argument specifies a (potentially large) table of line breaking parameters: the parameters that are not passed on explicitly are taken from the current typesetter state.
  The exact keys in the table are documented in the reference manual, but here is a simple, but complete example how to run line breaking on the content of `\box0`:

```
\setbox0=\hbox to \hsize{\input knuth }
\startluacode
local n = node.copy_list(tex.box[0].list)
local t = node.tail(n)
local final = node.new(node.id('glue'))
final.spec = node.new(node.id('glue_spec'))
final.spec.stretch_order = 2
final.spec.stretch = 1
node.insert_after(n,t, final)
local m = tex.linebreak(n,
          { hangafter = 2, hangindent = tex.sp("2em")})
local q = node.vpack(m)
node.write(q)
\stopluacode
```

The result is:

Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important. But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

Summary

All in all, there are not that many incompatible changes compared to LuaTEX 0.40, and the LuaTEX project is progressing nicely.

LuaTEX beta 0.70 will be released in the autumn of 2010. Our current plans for that release are: access to the actual pdf structures of included pdf images; a partial redesign of the mixed direction model; even more access to the LuaTEX internals from Lua; and probably some more ...

Taco Hoekwater & Hartmut Henkel

Luna—my side of the moon

Perhaps everyone knows this pleasant feeling when a long lasting project is finally done. A few years ago, just when I was almost happy with my pdfTEX environment, I saw LuaTEX for the first time. Instead of enjoying a relief, I had to take a deep breath and started moving the world to the Moon. The state of weightlessness resulted in that now, I am not able to walk on the 'normal' ground anymore. But I don't even think about going back. Although I still haven't settled down for good, the adventure is delightful. To domesticate a new environment I gave it a name—Luna.

First thoughts

My first thought after meeting LuaTEX was 'wow!'. Scripting with a neat programming language, access to TEX lists, the ability to hook some deep mechanisms via callbacks, font loader library on hand, integrated MetaPost library and more. All this was tempting and I had no doubts I wanted to go for it. At the first approach I was thinking of migrating my workflows step-by-step, replacing some core mechanisms with those provided by LuaTEX. But not only the macros needed to change. It was considering 'TEX' a programming language that needed to change. In LuaTEX I rather treat TEX as a paragraph and page building machine to which I can talk in a real programming language.

There were a lot of things I had to face before I was able to typeset anything, at least UTF-8 regime and a new TEX font representation. A lot of work that I never wanted to do myself. So just after 'wow!' also 'ooops... ' has come. In this article I focus on things rather tightly related to pdf graphics, as I find that part the most interesting, at least in a sense of taking advantage of Lua and LuaTEX functionalities.

\pdfliteral retires

TEX concentrates on texts, providing only a raw mechanism for document graphics features, such as colors, transparencies or geometry transformations. pdfTEX goes a little bit further providing some concept of a graphic state accessible for the user. But the tools for the graphic control remain the same. We have only specials in several variants.

What's wrong with them? The things they do behind the scenes may be harmful.

```
\def\flip#1{%
  \pdfliteral{q -1 0 0 -1 20 6 cm}%
  \hbox to0pt{#1\hss}%
  \pdfliteral{Q}\hbox to20bp{\hss}}
\def\red#1{%
  \pdfliteral page{q 0 1 1 0 k}%
  #1\pdfliteral page{Q}}
```

The first macro applies a transformation to a text object, the second applies a color. If used separately, they work just fine. If used as `\flip{\red{text}}`, it's still ok: text. Now try to say `\red{\flip{text}}`. The text is transformed and colored as expected. But all the rest of the page is broken, as its content is completely displaced! And now try `\red{\flip{text}?}` (with a question mark at the end of a parameter text). Everything is perfectly ok again: text?

Here is what happens. When `\pdfliteral` occurs, pdfTEX outputs a whatsit. This whatsit will cause writing the data into the output pdf content stream at the shipout time. If the literal was used in a default mode (with no `direct` or `page` keywords) pdfTEX first writes a transformation from lower-left corner of the page to the current position, then prints the user data, then writes another transformation from the current position back to the pdf page origin. Actually the transform restoration is not performed immediately after writing the user data, but on the beginning of the very next textual node. So in the case of several subsequent literal whatsit nodes, the transform may occur not where the naive user expects it. Simplifying the actual pdf output, we expected something like

```
q 0 1 1 0 k         % save, set color
1 0 0 1 80 750 cm   % shift to TeX pos
q -1 0 0 -1 20 6 cm % save, transform
BT ... ET           % put text
Q                   % restore transform
1 0 0 1 -80 -750 cm % shift (redundant)
Q                   % restore color
```

but we got

```
q 0 1 1 0 k
1 0 0 1 80 750 cm
q -1 0 0 -1 20 6 cm
```

```
BT ... ET
Q
Q
1 0 0 1 -80 -750 cm
```

In general, the behavior of \pdfliterals depends on the surrounding node list. There are reasons behind it. Nevertheless, one can hardly control lists in pdfTEX, so it's hard to avoid surprises.

Does LuaTEX provide something better then \pdfliterals? It provides \latelua. Very much like \pdfliteral, a \latelua instruction inserts a whatsit. At the time of shipout, LuaTEX executes the Lua code provided as an argument to \latelua. The code may call the standard pdf.print() function, which writes a raw data into a pdf content stream. So what's the difference? The difference is that in \latelua chunks we know the current position on the page, it is accessible through pdf.h and pdf.v fields. We can therefore use the position coordinates explicitly in the literal content. To simulate the behavior of \pdfliteral one can say

```
\latelua{
  local bp = 65781
  local cm = function(x, y)
    return string.format(
     "1 0 0 1 \%.4f \%.4f cm\string\n",
     x/bp, y/bp
    )
  end
  pdf.print("page", cm(pdf.h, pdf.v))
  % special contents
  pdf.print("page", cm(-pdf.h, -pdf.v))
}
```

Having the \latelua mechanism and the pdf.print() function, I don't need and don't use \pdfliterals any longer.

Graphic state

Obviously writing raw pdf data is supposed to be covered by lower level functions. Here is an example of how I set up graphic features in the higher level interface:

```
\pdfstate{
  local cmyk = color.cmyk
  cmyk.orange =
    (0.8*cmyk.red+cmyk.yellow)/2
  fillcolor = cs.orange
  opacity = 30
  linewidth = '1.5pt'
  rotate(30)
  ...
}
```

The definition of \pdfstate is something like

```
\long\def\pdfstate#1{%
  \latelua{setfenv(1, pdf) #1}}
```

The parameter text is Lua code. setfenv() call simply allows me to omit the 'pdf.' prefix before variables. Without that I would need

```
\latelua{
  pdf.fillcolor = pdf.color.cmyk.orange
  pdf.opacity = 30
  pdf.linewidth = '1.5pt'
  pdf.rotate(30)
  ...
}
```

pdf is a standard LuaTEX library. I extend its functionality, so that every access to special fields causes an associated function call. Every such function updates the internal representation of a graphic state and keeps the output pdf graphic state synchronized by writing out the appropriate content stream data. But whatever goes on behind the scenes, on top I have just key=value pairs. I'm glad I no longer need to think about obscure TEX interfaces for that. The Lua language is the interface.

I expect graphic features to behave more or less like the basic text properties, such as font and size. They should obey grouping and they should remain active across page breaks. The first requirement can be satisfied simply by using \aftergroup in conjunction with \currentgrouplevel. A simple grouping-wise graphic state could be made as follows:

```
\newcount\gstatelevel
\def\pdfsave{\latelua{
  pdf.print("page", "q\string\n")}}
\def\pdfrestore{\latelua{
  pdf.print("page", "Q\string\n")}}
\def\pdflocal#1{
  \ifnum\currentgrouplevel=\gstatelevel
  \else
    \gstatelevel=\currentgrouplevel
    \pdfsave \aftergroup\pdfrestore
  \fi \latelua{pdf.print"#1\string\n"}}

\begingroup \pdflocal{0.5 g}
this is gray
\endgroup
this is black
```

Passing the graphic state through page breaks is rel-

atively difficult due to the fact that we usually don't know where TEX thinks the best place to break is. In my earth-life I was abusing marks for that purpose or, when a more robust mechanism was needed, I used `\writes` at the price of another TEX run and auxiliary file analysis. And here is another advantage of using `\latelua`: since Lua chunks are executed during shipout, we don't need to worry about the page break because it has already happened. If every graphic state setup is a Lua statement performed in order during shipout and every such statement keeps the output pdf state in sync through `pdf.print()` calls, then after the shipout the graphic state is exactly what should be passed on to the next page.

In a well structured pdf document every page should refer only to those resources, which were actually used on that page. The pdfTEX engine guarantees that for fonts and images, the `\latelua` mechanism makes it straightforward for other resource types.

Note a little drawback of that late graphic state concept: before shipout one can only access the state at the beginning of the page, because recent `\latelua` calls that should update the current state have not happened yet. I thought this might be a problem and made a mechanism that updates a pending-graphic state for early usage, but, so far, I never needed to use it in practice.

PDF data structures

When digging deeper, we have to face creating custom pdf objects for various purposes. Due to the lack of composite data structures, in pdfTEX one was condemned to strings. Here is an example of pdf object creation in pdfTEX.

```
\immediate\pdfobj{<<
/FunctionType 2
/Range [0 1 0 1 0 1 0 1]
/Domain [0 1] /N 1
/C0 [0 0 0 0] /C1 [0 .4 1 0]
>>}
\pdfobj{
    [/Separation /Spot /DeviceCMYK
    \the\pdflastobj\space 0 R]
}\pdfrefobj\pdflastobj
```

In LuaTEX one can use Lua structures to represent pdf structures. Although it involves some heuristics, I find it convenient to build pdf objects from clean Lua types, like in this example:

```
\pdfstate{create
    {"Separation","Spot","DeviceCMYK",
        dict.ref{
            FunctionType = 2,
            Range = {0,1,0,1,0,1,0,1},
            Domain = {0,1}, N = 1,
            C0 = {0,0,0,0}, C1 = {0,.4,1,0}
        }
    }
}
```

Usually, I don't need to create an independent representation of a pdf object in Lua. I rather operate on more abstract constructs, which may have a pdf-independent implementation and may work completely outside of LuaTEX. For color representation and transformations I use my color library, which has no knowledge about pdf at all. An additional LuaTEX-dependent binder extends that library with extra skills necessary for the pdf graphic subsystem.

Here is an example of a somewhat complex colorspace, a palette of duotone colors, each consisting of two spot components with lab equivalent (the pdf structure representation for this is much too long to be shown here):

```
\pdfstate{
  local lab = colorspace.lab{
    reference = "D65"
  }
  local duotone = colorspace.poly{
    {name = "Black", lab.black},
    {name = "Gold",  lab.yellow},
  }
  local palette = colorspace.trans{
    duotone(0,100), duotone(100,0),
    n = 256
  }
  fillcolor = palette(101)
}
```

On the last line, the color object (simple Lua table) is set in a graphic state (Lua dictionary), and its colorspace (another Lua dictionary) is registered in a page resources dictionary (yet another Lua dictionary). The graphic state object takes care of updating a pdf content stream, and finally the resources dictionary 'knows' how to become a pdf dictionary.

It's never to late

When talking about pdf objects construction I've concealed one sticky difficulty. If I want to handle graphic setup using `\latelua`, I need to be able to create pdf objects during shipout. Generally, `\latelua` provides no legal mechanism for that. There is the `pdf.obj()` standard function, a LuaTEX equivalent of the `\pdfobj` primitive, but it only obtains an allo-

cated pdf object number. What actually ensures writing the object into the output is a whatsit node inserted by the `\pdfrefobj<number>` instruction. But in `\latelua` it is too late to use it. Also, don't try to use `pdf.immediateobj()` variant within `\latelua`, as it writes the object into the page content stream, resulting in an invalid pdf document.

So what can one do? LuaTEX allows one to create an object reference whatsit by hand. If we know the tail of the list currently written out (or any list node not yet swallowed by a shipout procedure), we can create this whatsit and put it into the list on our own (risk), without the use of `\pdfrefobj`.

```
\def\shipout{%
  \setbox256=\box\voidb@x
  \afterassignment\doshipout\setbox256=}
\def\doshipout{%
  \ifvoid256 \expandafter\aftergroup \fi
  \lunashipout}
\def\lunashipout{\directlua{
  luna = luna or {}
  luna.tail =
    node.tail(tex.box[256].list)
  tex.shipout(256)
}}
\latelua{
  local data = "<< /The /Object >>"
  local ref = node.new(
    node.id "whatsit",
    node.subtype "pdf_refobj"
  )
  ref.objnum = pdf.obj(data)
  local tail = luna.tail
  tail.next = ref  ref.prev = tail
  luna.tail = ref % for other lateluas
}
```

In this example, before every `\shipout` the very last item of the page list is saved in `luna.tail`. During shipout all code snippets from `late_lua` whatsits may create a `pdf_refobj` node and insert it just after the page tail which ensures that the LuaTEX engine will write them out.

Self-conscious \latelua

If every `\latelua` code chunk may access the page list tail, why not to give it access to the `late_lua` whatsit node to which this code is linked? Here is the conceptual representation of a whatsit that contains Lua code that can access the whatsit itself:

```
\def\lateluna#1{\directlua{
  local self = node.new(
    node.id "whatsit",
    node.subtype "late_lua"
  )
  self.data = "\luaescapestring{#1}"
  luna.this = self
  node.write(self)
}}
\lateluna{print(luna.this.data)}
```

Beyond the page builder

A self-printing Lua code is obviously not what I use this mechanism for. It is worthwhile to note that if we can make a self-aware `late_lua` whatsit, we can also access the list following this whatsit. It is too late to change previous nodes, as they were already eaten by shipout and written to the output, but one can freely (which doesn't mean safely!) modify the nodes that follow the whatsit.

Let's start with a more general self-conscious `late_lua` whatsit:

```
\long\def\lateluna#1{\directlua{
  node.write(
    luna.node("\luaescapestring{#1}")
  )
}}
\directlua{
luna.node = function(data)
  local self = node.new(
    node.id "whatsit",
    node.subtype "late_lua"
  )
  local n = \string#luna+1
  luna[n] = self
  self.data =
    "luna.this = luna["..n.."] "..data
  return self
end
}
```

Here is a function that takes a text string, font identifier and absolute position as arguments and returns a horizontal list of glyph nodes:

```
local string = unicode.utf8
function luna.text(s, font_id, x, y)
  local head = node.new(node.id "glyph")
  head.char = string.byte(s, 1)
  head.font = font_id
  head.xoffset = -pdf.h+tex.sp(x)
  head.yoffset = -pdf.v+tex.sp(y)
  local this, that = head
  for i=2, string.len(s) do
```

```
    that = node.copy(this)
    that.char = string.byte(s, i)
    this.next = that  that.prev = this
    this = that
  end
  head = node.hpack(head)
  head.width = 0
  head.height = 0
  head.depth = 0
  return head
end
```

Now we can typeset texts even during shipout. The code below results in typing the `it is never too late!` text with a 10bp offset from the page origin.

```
\lateluna{
  local this = luna.this
  local text = luna.text(
    "it is never too late!",
    font.current(), '10bp', '10bp'
  )
  local next = this.next
  this.next = text  text.prev = this
  if next then
    text = node.tail(text)
    text.next = next  next.prev = text
  end
}
```

Note that when mixing shipout-time typesetting (manually generated lists) and graphic state setups (using `pdf.print()` calls), one has to ensure that the placement of things is in the correct order. Once a list of glyphs is inserted after a `late_lua` whatsit, the embedded Lua code should not print literals into the output. All literals will effectively be placed before the text anyway. Here is a funny mechanism to cope with that:

```
\lateluna{
luna.thread = coroutine.create(
function()
  local this, next, text, tail
  for i=0, 360, 10 do
    % graphic setup
    pdf.fillcolor =
      pdf.color.hsb(i,100,100)
    pdf.rotate(10)
    % glyphs list
    this = luna.this  next = this.next
    text = luna.text("!",
      font.current(), 0, 0)
    this.next = text  text.prev = this
    text = node.tail(text)
    % luna tail
    tail = luna.node
      "coroutine.resume(luna.thread)"
    text.next = tail tail.prev = text
    if next then
      tail.next = next next.prev = tail
    end
    coroutine.yield()
  end
end)
coroutine.resume(luna.thread)
}\end
```

This is the output:

Once the page shipout starts, the list is almost empty. It contains just a `late_lua` whatsit node. The code of this whatsit creates a Lua coroutine that repeatedly sets some color, some transformation and generates some text (an exclamation mark) using an already known method. The tail of the text is another `late_lua` node. After inserting the newly created list fragment, the thread function yields, effectively finishing the execution of the first `late_lua` chunk. Then the shipout procedure swallows the recently generated portion of text, writes it out and takes care of font embedding. After the glyph list, the shipout spots the `late_lua` whatsit with the Lua code that resumes the thread and performs another loop iteration, creating a graphic setup and generating text again. So the execution of the coroutine starts in one whatsit, but ends in another, that didn't exist when the procedure started. Every list item is created just before being processed by the shipout.

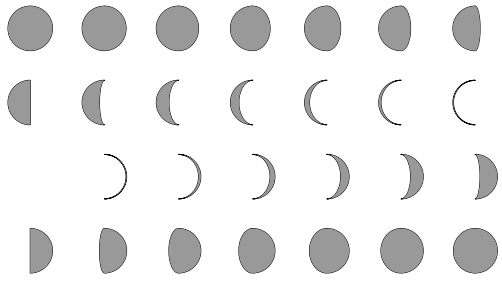
Reinventing the wheel

Have you ever tried to draw a circle or ellipse using `\pdfliterals`? It is very inconvenient, because the pdf format provides no programming facilities and painting operations are rather limited in comparison with its PostScript ancestors. Here is an example of some PostScript code and its output. The code uses control structures, which are not available in pdf. It also

takes advantage of the arc operator that approximates arcs with Bézier curves. To obtain elliptical arcs, it uses the fact that (unlike in pdf) transformations can be applied between path construction operators.

```
/r 15 def
/dx 50 def /dy -50 def
/pos {day 7 mod dx mul week dy mul} def
/arx /arc load def

dx dy 4 mul neg translate
0.6 setgray 0.4 setlinewidth
1 setlinejoin 1 setlinecap
0 1 27 {
  /day exch def /week day 7 idiv def
  /s day 360 mul 28 div cos def
  day 14 eq {
    /arx /arcn load def
  } {
    gsave pos r 90 270 arx
    day 7 eq day 21 eq or {
      closepath
      gsave 0 setgray stroke grestore
    } {
      s 1 scale
      pos exch s div exch r 270 90 arx
      gsave 0 setgray initmatrix stroke
      grestore
    } ifelse
    fill grestore
  } ifelse
} for
```



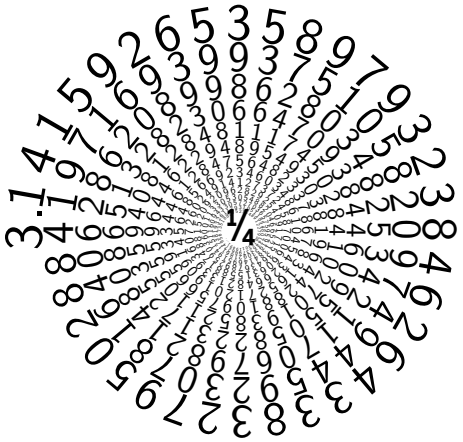
In LuaTEX one can hire MetaPost for drawings, obtaining a lot of coding convenience. The above program wouldn't be much simpler, though. As for now MetaPost does not generate pdf, the data it outputs still needs some postprocessing to include the graphic on-the-fly into the main pdf document.

As I do not want to invent a completely new interface for graphics, I decided to involve PostScript code into the document creation. Just to explain how it may pay off, after translating the example above into a pdf content stream I obtain 30k bytes of code, which is quite a lot in comparison with 500 bytes of PostScript input.

PostScript support sounds scary. Obviously I'm not aiming to develop a fully featured PostScript machine on the LuaTEX platform. The PostScript interpreter is supposed to render the page on the output. In Luna I just write the vector data into the pdf document content, so what I actually need is a reasonable subset of PostScript operators. The aim is to control my document graphics with a mature language dedicated to that purpose. The following two setups are equivalent, as at the core level they both operate on the same Lua representation of a graphic state.

```
\pdfstate{% lua interface
  save()
  fillcolor = color.cmyk(0,40,100,0)
  ...
  restore()}
\pdfstate{% postscript interface
  ps "gsave 0 .4 1 0 setcmykcolor"
  ...
  ps "grestore"
}
```

A very nice example of the benefit of joining typesetting beyond the page builder and PostScript language support is the π-spiral submitted by Kees van der Laan:



(see www.gust.org.pl/projects/pearls/2010p)

Paweł Jackowski
GUST

PDF/A-1a in ConTEXt MkIV

Abstract
I present some considerations on electronic document archiving and how ConTEXt MkIV supports the ISO Standard 19500-1 Level A Conformance (PDF/A-1a:2005), an ISO standard for long-term document archiving.

Keywords
LuaTeX, ConTeXt MkIV, PDF/A, color, font.

Introduction

In this paper I will briefly talk about the ISO Standard PDF/A-1 and how ConTEXt MkIV tries to adhere to its requirements by showing some practical examples. About the typographic style of this paper: I will follow these simple rules: I will avoid footnotes and citations on running text, and I will try to limit lists (e.g. only itemize and enumerate) and figures; the last section before the References one will collect all citations.

The PDF/A-1 ISO Standard

Probably one of the best known PDF versions is PDF 1.4 (around 2001, almost ten years ago) maybe because the companion Acrobat 5.0 was a robust program and the PDF Reader was freely available for several platforms both as a program and as plug in for browsers. We keep having a huge amount of electronic documents that are in PDF 1.4, hence we should not be surprised if Adobe pushed it as reference for document archiving. What follows is a verbatim copy from http://www.digitalpreservation.gov/formats/fdd/fdd000125.shtml and it's a good description:

> PDF/A-1 is a constrained form of Adobe PDF version 1.4 intended to be suitable for long-term preservation of page-oriented documents for which PDF is already being used in practice. The ISO standard [ISO 19005-1:2005] was developed by a working group with representatives from government, industry, and academia and active support from Adobe Systems Incorporated. Part 2 of ISO 19005 (as of September 2010, an ISO Draft International Standard) extends the capabilities of Part 1. It is based on PDF version 1.7 (as defined in ISO 32000-1) rather than PDF version 1.4 (which is used as the basis of ISO 19005-1).
>
> PDF/A attempts to maximize device independence, self-containment, self-documentation. The constraints include: audio and video content are forbidden, JavaScript and executable file launches are prohibited, All fonts must be embedded and also must be legally embeddable for unlimited, universal rendering, colorspaces specified in a device-independent manner, encryption is disallowed, use of standards-based metadata is mandated.
>
> The PDF/A-1 standard defines two levels of conformance: conformance level A satisfies all requirements in the specification; level B is a lower level of conformance, 'encompassing the requirements of this part of ISO 19005 regarding the visual appearance of electronic documents, but not their structural or semantic properties'.

In essence the standard wants to ensure that every typographic element, from the low level character to the high level logical structure is unambiguously defined and unchangeable — and it does, it achieves its purpose: every character must be identified by a Unicode id, which is an international standard, every color must be device independent by means of a color profile or output intent, there must be precise meta data informations for classifications and the document must have a logical structure described by a (possible ad-hoc) markup language.

Unfortunately the PDF version 1.4 is quite old: animations and 3D pictures cannot be embedded, the font format cannot be OpenType, JavaScript programs are not permitted at all, even if they don't modify the document in any way as, for example, a calculator. Ten years ago it was very important to guarantee that the document would always be printed as intended, nowadays screen is slowly replacing paper and animations play a fundamental role: PDF/A-1 is good for paper but less than optimal for 'electronic paper'.

PDF/A-1a in ConTEXt MkIV

Given that it is still under heavy development, ConTEXt

MkIV has the opportunity to be developed on two fronts: the 'low level' luaTEX (CWEB code and Lua primitives) and the 'high level' macros that build the format itself. One of this year's results is the implementation of 'tagged PDF', the Adobe document markup language for PDF documents, and the development of color macros for the PDF/X specifications. As a consequence, it was possible to use these results to test some real code for producing PDF/A-1a compliant documents. Let's start with an example explained step-by-step.

```
%% Debug
\enabletrackers[backend.format,
                backend.variables]
%% For PDF/A
\setupbackend[
format={pdf/a-1a:2005},
profile={default_cmyk.icc,
         default_rgb.icc,default_gray.icc},
intent={%
ISO coated v2 300\letterpercent\space (ECI)}
]
%% Tagged PDF
%% method=auto ==> default tags by Adobe
\setupstructure[state=start,method=auto]

\definecolor[Cyan][c=1.0,m=0.0,y=0.0,k=0.0]
\starttext
\startchapter[title={Test}]
\startparagraph
\input tufte
%% Some ConTeXt env. are already mapped:
%% colors
\color[red]{OK}
\color[Cyan]{OK}
%% figures
\externalfigure[rgb-icc-srgb.jpg]
               [width=0.4\textwidth]
\stopparagraph
%% Natural tables
\bTABLE
\bTR\bTD 1 \eTD \bTD 2 \eTD \eTR
\bTR \bTD[nx=2] 3 \eTD\eTR
\eTABLE
\stopchapter
\stoptext
```

As usual the file is processed with

```
#>context test.tex
```

and it doesn't hurt to enable some debug information with

```
\enabletrackers[backend.format,
                backend.variables]
```

Enable the PDF/A-1a

To enable PDF/A-1a we must setup the backend with the appropriate variant of PDF/A. From the very beginning ConTEXt has had a backend system that permits to use almost the same macro-format for different outputs (i.e. DVI and PDF), and with luaTEX this system is increasingly enhanced, as we'll see later on.
With

```
format={pdf/a-1a:2005}
```

we select the 1a variant of PDF/A standard and the label is mandatory because it also puts some default meta data into the output (see `lpdf-pda.xml`; a complete list of formats is currently in `lpdf-fmt.lua` and also as a Lua table `lpdf.formats`).

Next comes the colors part, and we must pay attention here. The key concept is:

every color must be independent of any device.

Usually in a PDF we have two sources for colors: the colors specified by the author, e.g. something like `\definecolor[orange][r=1.0,g=0.5,b=0.0]`, and the images. The most used color spaces `DeviceGray`, `DeviceRGB`, `DeviceCMYK` are device dependent because the reproduction of a color from these color spaces *depends* on the particular output device, and the real output devices are all different due both to the different nature (screen vs. printer, for example) and different technologies (CRT vs. LCD screen, or inkjet vs laser printer, for example). Every device can be classified by means of a *color profile* which maps an input color (rgb, cmyk or gray) to an *independent color space*: such maps ensure that each device will correctly reproduce the color, and also the independent color space permits to compare colors from different color spaces.

With

```
profile={default_cmyk.icc,
         default_rgb.icc,default_gray.icc},
```

we associate all the document colors with the corresponding color profile by mean of a filename (the file `colorprofiles.xml` has a list of predefined profiles). Be careful here: it's wrong to associate a rgb color space with a cmyk profile, and not all profiles are good, especially those for printing. Moreover PDF/A-1a allows only profiles having version 3 or below.

There is a second way to specify colors, and it's a bit complicated. We must specify that all the colors *without profile* are intended to be used with a common output profile, i.e. we must impose an *output intent*: this is the meaning of

```
intent={%
ISO coated v2 300\letterpercent\space (ECI)}
```

which is a cmyk profile for coated paper. Note that we are using a name and not a filename to avoid clashing with the values of the `profile` key.
By doing so we accept these implicit limitations and color space conversions:

- if the output intent is a cmyk profile then the document can have only cmyk and gray colors;
- if the output intent is a rgb profile then the document can have only rgb and gray colors;
- if the output intent is a gray profile then the document can have only gray colors.

They are reasonable: in general we cannot use a rgb color with a cmyk profile because there are rgb colors without equivalent cmyk ones (that is to say that screens display more colors than printers). We can convert a gray color to rgb or cmyk because usually gray color spaces are a subset of the former (otherwise we have a really poor device). It's not an error if we specify both profiles and output intent: at least if all color spaces have their own profiles, as in the example, then the output intent is simply ignored by a PDF/A compliant PDF reader.

Finally the images: we must be sure that every image has its color profile — and this can be a bit complicated.

In the following example, `rgb-noprofile.jpg` is a jpeg image with a RGB color space and without a color profile:

```
\setupbackend[
format={pdf/a-1a:2005},%level=0,
profile={default_cmyk.icc,
         default_rgb.icc,default_gray.icc},
]
\setupstructure[state=start]
\starttext
\startchapter[title={Test}]
\startparagraph
\externalfigure[rgb-noprofile.jpg]
             [width=0.4\textwidth]
\stopchapter
\stoptext
```

The luatex program loads the image, it wraps it in a `/XObject`, and sets its ColorSpace to DeviceRGB:

```
<<
/Type /XObject
/Subtype /Image
/Width 640
/Height 400
/BitsPerComponent 8
/Length 13238
/ColorSpace /DeviceRGB
/Filter /DCTDecode
>>
stream...endstream
```

This is a valid PDF/A-1a document, but if we delete the `default_rgb.icc` profile

```
profile={default_cmyk.icc,default_gray.icc},
```

then the resulting PDF is an *invalid* PDF/A. We should not be surprised: there is color space which is device dependent and hence we cannot guarantee the correct reproduction of the colors.

In the next example we use a rgb image *with a valid color profile*:

```
\setupbackend[
  format={pdf/a-1a:2005},%level=0,
  rofile={default_cmyk.icc,default_gray.icc}]
\setupstructure[state=start]
\starttext
\startchapter[title={Test}]
\startparagraph
\externalfigure[rgb-icc-srgb.jpg]
               [width=0.4\textwidth]
\stopchapter
\stoptext
```

For the same reason seen before, this PDF is still an invalid PDF/A: the image is again wrapped in a `/XObject` with a `/DeviceRGB` color space — but this time it's not correct: the image has its own profile and hence its colors are device independent. If we add a rgb profile we have again a valid PDF/A:

```
profile={default_cmyk.icc,
default_rgb.icc,default_gray.icc},
```

but this is dangerous because we don't know if it's correct for the image and also in this way *all* the rgb color spaces of others images are associated to this specific profile.

To remedy this situation, I present here a practical solution that relies on the MagickWand suite which is available for free for Windows, Linux and Mac platforms. The first step is to verify if the image has a profile:

```
#>gm identify -verbose rgb-icc-srgb.jpg
 :
 Profile-color: 3144 bytes
 :
```

The second step is to save the profile:

```
#>gm convert  rgb-icc-sRGB_v4_ICC.jpg  sRGB.icc
```

and the last step is to build a /XObject with the appropriate color space. This is a bit tricky, but fundamentally we mimic the behavior of luatex with ConTEXt MkIV. I will show only an example for a jpeg image with a /DeviceRGB color space:

```
%% rgb-icc-srgb.pdf
\pdfminorversion4
\starttext\startTEXpage%
\startluacode
local a=img.scan{filename="rgb-icc-srgb.jpg"}

tex.sprint(tex.ctxcatcodes,
 string.format(
  "\\startfoundexternalfigure{\%ssp}{\%ssp}",
  a.width,a.height))
 local icc_ref = pdf.immediateobj("streamfile",
  "srgb.icc",
  " /Alternate /DeviceRGB\n" ..
  "/Filter /FlateDecode\n/N 3")

local icc_dict_ref = pdf.immediateobj(
  string.format("[ /ICCBased \%d 0 R ]\n",
                icc_ref) )

a=img.new{filename="rgb-icc-srgb.jpg",
  colorspace=icc_dict_ref}

a=img.immediatewrite(a)

node.write(img.node(a))

tex.sprint(tex.ctxcatcodes,
  "\\stopfoundexternalfigure")
\stopluacode%
\stopTEXpage\stoptext
```

As we can see the XObject has now an ICCBased color space:

```
15 0 obj
<<
/Alternate /DeviceRGB
/Filter /FlateDecode
/N 3
/Length 3144
>>
stream...endstream
16 0 obj
[ /ICCBased 15 0 R ]
endobj
17 0 obj
<<
/Type /XObject
/Subtype /Image
/Width 640
/Height 400
/BitsPerComponent 8
/Length 9948
/ColorSpace 16 0 R
/Filter /DCTDecode
>>
stream...endstream
```

Once the image with the correct color space is wrapped in a PDF file (rgb-icc-srgb.pdf in this case), we can use it in our documents:

```
\setupbackend[
format=[{pdf/a-1a:2005},%level=0,
    profile={default_cmyk.icc,default_gray.icc},
      ]
\setupstructure[state=start]
\starttext
\startchapter[title={Test}]
\startparagraph
\externalfigure[rgb-icc-srgb.pdf]
               [width=0.4\textwidth]
\stopchapter
\stoptext
```

which is again a valid PDF/A.

Tagged PDF

Next we must enable the tagging system with \setupstructure[state=start,method=auto]. ConTEXt MkIV permits the author to define his own document markup language (the tags used inside the PDF document) but of course we also need the associated TEX macros. This naturally needs to start with a sort of XML document:

```
\setupstructure[state=start,method=none]
\starttext
\startelement[document]
\startelement[chapter]
opes
\startelement[p]\input ward\stopelement \par
\stopelement
\stopelement
\stoptext
```

The internal tag names are <document>, <chapter> and <p> as we see in fig. 1 from Acrobat 9.0, but we still need to put the appropriate typographic elements into the PDF.

In the context of PDF/A, a validation program expected the tags as defined by Adobe and this leads to some 'syntactic sugar' macros, i.e instead of

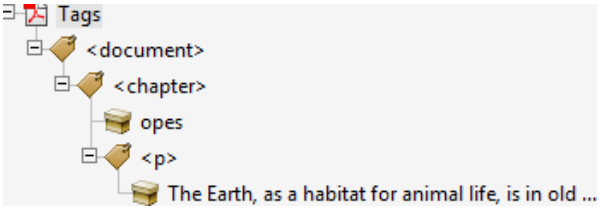


Figure 1 The tags structure of a simple document

`\startelement[chapter]...\stopelement`

it's better to use

`\startchapter[title={Test}]...\stopchapter`

which puts the correct tags and also typesets the chapter title Test as expected.

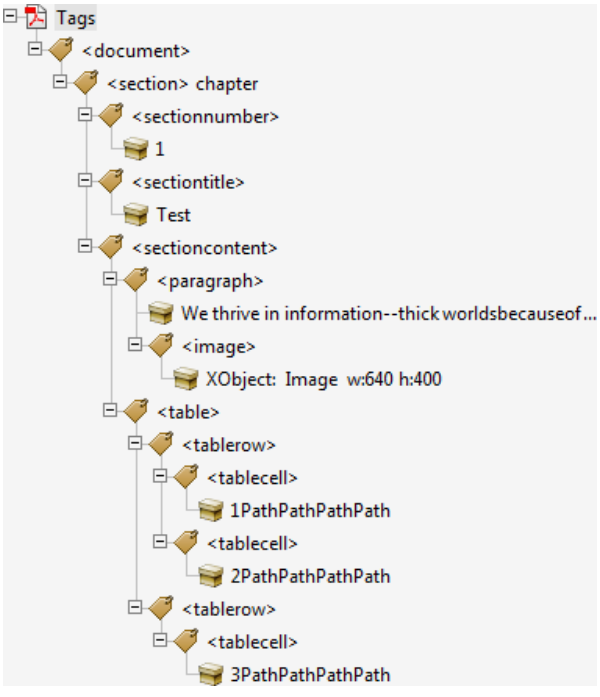


Figure 2 The tag structure of complex document

The complete list of tags can be found in `strc-tag.mkiv` and of course ConTEXt MkIV permits to redefine the default mapping. In fig. 2 our document shows that ConTEXt MkIV had already mapped some predefined typographic objects like figures and tables to the appropriate tags.

We can use this mechanism to embed an XML document into a tagged PDF document, which opens quite interesting perspectives, but we can also start from a 'structured TEX' document and end into an XML one, and this is more interesting because it's a matter of backend only — and because it's already implemented:

```
\setupbackend[export=yes]
\setupstructure[state=start,method=none]
\starttext
\startelement[document]
\startelement[chapter][title=Test]
opes
\startelement[p]\input ward\stopelement \par
\stopelement
\stopelement
\stoptext
```

produces a `<tex-file>.export` like this (original XML spaces are not preserved in this listing)

```
<?xml version='1.0' standalone='yes' ?>
<!-- input filename   : test-2            -->
<!-- processing date  : 10/09/10 15:28:48 -->
<!-- context version  : 2010.09.24 11:40  -->
<!-- exporter version : 0.10              -->
<document language='en'
   file='test-2' date='10/09/10 15:29:04'
   context='2010.09.24 11:40'
   version='0.10'>
<chapter title="Test">opes
<p>
The Earth, as a habitat for animal life, is
in old age and has a fatal illness. Several,
in fact. It would be happening whether humans
had ever evolved or not. But our presence is
like the effect of an old-age patient who
smokes many packs of cigarettes per day
------ and we humans are the cigarettes.
</p>
</chapter>
</document>
```

Fonts and encoding

In the previous subsection we have seen that with simple macros we can have a *valid* (i.e validated by Acrobat 9.0) PDF/A-1a PDF document. We still didn't talk about fonts.

The default fonts used by ConTEXt MkIV are the OpenType version of LatinModern, and, as of now, they cannot be embedded into PDF/A documents because OpenType isn't supported in version 1.4; this is not a problem because, in essence, ConTEXt MkIV strips the OpenType part and embeds a valid Type1 or TrueType font. Given an OpenType font, ConTEXt MkIV is also able to map each glyph to its Unicode `id`, so even this side is not problematic.

Unfortunately, it's already known that typesetting mathematics with the Computer Modern and Latin

Modern fonts easily leads to invalid PDF/A documents due to misleading dimensional information of some fonts. As widely noted by C. Beccari, just the simple `$a\not=b$` invalidates the whole document, due the wrong dimension of the `\not` sign (it has BoundingBox=(139,139,-960,775) hence a width equal to zero). What are the solutions? There are two of them, both unsatisfactory:

1. choose another (valid) math family;
2. make a high resolution (more than 300dpi) bitmap of each invalid formula.

Of course it's possible to edit the fonts, but it's not a general solution: there are limitations due to copyright and we should embed a modified copy of the font that differs from the original version — an error prone situation because modifications of PDF/A-1a document are permitted, and an editor can use the system fonts. The problem remains even if ConTEXt MkIV can patch the font on the fly. A way out is the complete embedding of the patched fonts, so that the editor uses the document fonts, but it's not a robust solution — some editors can still use the original system fonts.

Conclusion

The PDF/A-1a is a good standard for document archiving: it's a complete *Page Description Language*, it relies on Unicode which is also a good *Character Language* and on Type1 and TrueType as *digital typography formal language*; it has also a good *Document Markup Language*. The binary electronic format and the digital signature for detection and prevention of document modifications complete the picture. The restrictions (e.g. profiles for colors) together with a freely available PDF/A-1a PDF reader lead to a concrete self-containment format.

PDF/A-1a support in ConTEXt MkIV is still experimental: it needs more tests, but programming in luaTEX is simpler than in pdfTEX, and the 1.4 is a well known PDF version. The color management can probably be improved by permitting to specify a color and its profile for a specific object and not for the whole document, as it currently is.

On the other hand, the model of PDF/A-1 is the traditional paper. Omitting animations and 3D pictures is questionable and perhaps also scripting languages should be permitted if they don't modify the document.

The ISO standard is not freely available and the PDF/A-1a validators are complex to implement and usually expensive commercial products; this is an obstacle for the diffusion of PDF/A.

Notes on References

For the first section, some informations on PDF/A-1 are at Wikipedia [1], the techdoc at [2], and [4]. Very useful are also the references of C. Beccari's paper at [9]. An interesting use of JavaScript in PDF is [3].

For the second section, the ConTEXt wiki [6] has some terse informations, because the code is the ultimate reference. Tagged PDF is described in the version of `hybrid.pdf` [7] that is part of '*Proceedings of the 4th ConTEXt meeting*' [8](to be published). For ICC profiles a good starting point is [5]; the problems about fonts are described by C. Beccari in [9] and [10].

References

All links were verified between 2010.10.19 and 2010.10.22.

[1] http://en.wikipedia.org/wiki/PDF/A.
[2] http://www.pdfa.org/doku.php?id=pdfa:en:techdoc
[3] www.tug.org/applications/pdftex/calculat.pdf.
[4] http://www.digitalpreservation.gov/formats/fdd/fdd000125.shtml.
[5] http://en.wikipedia.org/wiki/ICC_profile.
[6] http://wiki.contextgarden.net/PDFX.
[7] http://www.pragma-ade.com/general/manuals/hybrid.pdf
[8] http://meeting.contextgarden.net/2010/talks/
[9] http://www.guit.sssup.it/downloads/Beccari_Pdf_archiviabile.pdf
[10] http://dw.tug.org/pracjourn/2010-1/beccari

Luigi Scarso

Three things you can do with LuaTEX that would be extremely painful otherwise[1]

Introduction

LuaTEX has made some typographic operations so easy one might wonder why it wasn't invented thirty years ago (probably because Lua didn't exist then).[2]

Here I'm going to describe three simple features that would require advanced wizardry to do the same with any other engine. LuaTEX allows you to explore some of TEX's most intimate parts with a rather easy programming language, and the result is you can quite readily access things that were unreachable before. The three issues I'm going to address are:

- Turning lines into rules whose color depends on the line's original stretch or shrink.
- Underlining.
- Margin notes that align properly with the text.

I'll try to explain some of LuaTEX's basic functionality as we encounter these issues, but two of them are worth mentioning right now: callbacks and nodes.

First, we can control TEX's operations at various stages thanks to *callbacks*. These are points at which we can insert Lua code to modify or enhance TEX's processing. Callbacks range from processing TEX's input buffer (e.g. to accommodate a special encoding) to rewriting the paragraph builder and loading OpenType fonts.

Second, we can manipulate lists of *nodes*. To put it simply, nodes are the atoms that TEX uses to create pages: boxes, glyphs, glues, but also penalties, whatsits, etc. A list of nodes is a sequence of such atoms linked together. A simple paragraph, for instance, is a list made of horizontal boxes (the lines), penalties and glues. The boxes themselves are lists containing mostly glyph and glue nodes. Nodes are linked together like beads on a string, and the `prev` field of a node points to the preceding node in the list, whereas the `next` field returns the one that follows (there is an understandable exception for the first and last nodes of a list, whose `prev` and `last` fields respectively return `nil`). An important point to keep in mind is that when you query the content of, say, an `\hbox`, which in TEX's internal is a horizontal list, what you get is the first node of that list; you access the rest by sliding from `next` to `next`.

Nodes also have several other *fields*, depending on their types. These types are recorded as a number in their `id` field, a numeric value. For instance, a glue node has `id` 10, whereas a glyph node has `id` 37. As long as LuaTEX hasn't reached version 1, though, such values might change. So, in order for our code to last, we must use the following workaround: the `node.id()` function, when fed a string denoting a node type, returns the associated `id` number. For instance, `node.id("glue")` returns `10`. Thus, when using symbolic names, we can get the right `id` value, regardless of changes in versions of LuaTEX. Another important field for nodes is `subtype`, which distinguishes between nodes with the same `id`. It's a numeric value, and for whatsits (which are numerous), one should use `node.subtype()` like `node.id()`.

Symbolic names won't change; they are listed in the LuaTEX reference manual, in the chapter called *Nodes*, available from the LuaTEX web site; they're also listed in the tables returned by `node.types()` and `node.whatsits()`. It's simpler to define variables beforehand rather than call `node.id` and `node.subtype` each time we need them. That's what we'll do here: the following declarations should start any file containing our code; it can also be made global by removing the `local` prefix and thus used anywhere once declared, but local variables are faster and safer. I use uppercase to mark their status.

```
local HLIST = node.id("hlist")
local RULE  = node.id("rule")
```

1. First published in *TUGboat* **31**:3 (2010), pp. 184–190.
2. *Author's note:* I'm not a member of the LuaTEX team and this paper has no kind of official authority—it's just the result of experimentation by a LuaTEX user. Any error or misconception is mine.

```
local GLUE  = node.id("glue")
local KERN  = node.id("kern")
local WHAT  = node.id("whatsit")
local COL   = node.subtype("pdf_colorstack")
```

The color of a page

Typographers speak of a page's color. While the color itself depends on several factors, its *evenness* depends on how lines are justified: loose lines make the page uneven in color, because large interword space creates holes in the overall greyness.

The code that follows takes the metaphor literally: it turns a page's color into a real color pattern. The idea is to replace each line with a rule of the same height and width, and whose color depends on the line's badness. If we take 0 as black and 1 as white, then a good line gets .5, tight lines approach 0 (which represents an overfull line) and loose lines tend to 1 (an underfull line). Now we have paragraphs and pages made of grey bars; the less contrast between them, the better the page.

To do this, we retrieve the horizontal boxes created by the paragraph builder, check the badness of each, then replace the box with the desired rule. This is easy to do in LuaTEX: we register a function in the `post_linebreak_filter` callback. This callback accesses the list of nodes output by the paragraph builder, i.e. the lines of text interspersed with interline penalties and glues, plus perhaps other things (whatsits, inserts, ...) that we'll ignore. Among these nodes we retrieve the ones we want, namely the lines of text, and replace them as described.

The code that follows, as all Lua code, should be fed to `\directlua` or stored in a `.lua` file.

```
local color_push = node.new(WHAT, COL)
local color_pop  = node.new(WHAT, COL)
color_push.stack = 0
color_pop.stack  = 0
color_push.cmd   = 1
color_pop.cmd    = 2
```

Here we have created two new whatsit nodes identified by their `subtype` as the Lua equivalents of `\pdfcolorstack`. They both modify stack 0 and `color_push` adds code to the stack while `color_pop` removes it. We'll use them to set the color of each line, with the exact content of the code added by `color_push` to be specified each time.

```
textcolor = function (head)
  for line in node.traverse_id(HLIST, head) do
    local glue_ratio = 0
    if line.glue_order == 0 then
      if line.glue_sign == 1 then
        glue_ratio = .5 * math.min(line.glue_set,
                                   1)
      else
        glue_ratio = -.5 * line.glue_set
      end
    end
    color_push.data = .5 + glue_ratio .. " g"
```

Here's the beginning of our main function. It takes a node as its argument: it will be the first node of the list returned by the paragraph builder. That node, remember, denotes the entire list. We retrieve each line of text in this list, i.e. each node with `id` `HLIST`, and check its `glue_order` field; if it is 0, then the line has been justified with finite glue and we want to know how bad it is (if the line uses infinite glue then it is good by definition, as far as glue setting is concerned). We access `glue_sign` to know whether stretching or shrinking was used and `glue_set` to know the ratio (1 means the stretch/shrink was fully used; glues can also be overstretched, but we don't allow more than 1 in order to remain in the color range).

The last line sets the color of the line as the code to `color_push`, i.e. `` `n g' ``, where `n` is a number between 0 and 1 and `g` a pdf operator setting the color in the grey model. In the rest of the loop we replace the line's content with a sequence of three nodes: `color_push`, a rule, and `color_pop`:

```
    local rule = node.new(RULE)
    rule.width = line.width
    local p = line.list
    line.list = node.copy(color_push)
    node.flush_list(p)
    node.insert_after(line.list,
      line.list, rule)
    node.insert_after(line.list,
      node.tail(line.list),
      node.copy(color_pop))
  end
```

What is done here is: first, we create a rule whose width is the same as the original line's (we could have created this rule beforehand with a width equal to `\hsize`, but this way we accommodate changing line widths). Then we set the line's list as a copy of `color_push` (we use a copy since we need that node for each line), and then we insert the rule node and a copy of `color_pop`. The first argument to `node.insert_after` is the list (denoted by its first node!) where we perform the insertion, the second one is the node in that list after which the insertion is performed, and the last one is the inserted node; `node.tail` returns the last node of its argument,

so the third node.insert_after inserts at the end of the list.

The story with p is this: we retrieve the line's content before replacing it, so we can erase it from TEX's memory; it has no effect on the output.

Finally, and most importantly, we return the mutated list for TEX to continue its operations, and close the function.

```
  return head
end
```

Now, to use the function, we register it in the post_linebreak_filter callback:

```
\directlua{%
  callback.register("post_linebreak_filter",
    textcolor)}
```

Note that we could improve this code for the first and last lines of a paragraph, taking the indent and \parfillskip into account to create more faithful images of those lines. I leave it as an exercise to the reader, as is customary.

Underlining

The previous code was (hopefully) fun but not terribly useful (well, who knows?); let's do something (hopefully) more useful and no less fun.

Everybody knows that underlining is in bad typographic taste. That said, it may have its uses, and anyway allows us to investigate LuaTEX further. Underlining has been done in TEX (see Donald Arseneau's ulem, for instance); it requires great wizardry and has some limitations. With LuaTEX, it's (almost) child's play.

The problem with underlining in TEX is that you have to add the underline before the paragraph is built, and this hinders hyphenation. In LuaTEX we can do it after hyphenation is done: we retrieve the nodes to underline in the typeset lines. But how do we spot them? The answer lies with another basic LuaTEX functionality, namely attributes. These are very simple yet very powerful. An attribute is like a count register in that it holds a number. The difference with a count register is that nodes retain the values of all attributes in force *when they were created.* Thus, we can set an attribute to some value, input some text, and then reset the attribute; the text will have the value attached to it for the rest of TEX's processing.

This leads to the first definition:

```
\def\underline#1{%
  \quitvmode \attribute100 = 1 #1%
  \attribute100 = -"7FFFFFFF
  \directlua{callback.register(
    "post_linebreak_filter", get_lines)}%
}
```

It's important to use \quitvmode so that the indentation box is inserted before the attribute is set and not be underlined (in case the underlined text is the beginning of a paragraph).

An attribute is `set' if it has any value but -"7FFFFFFF. So setting it to 1 here would be the same thing as setting it to −45 (see the end of this section for an example of use for different values). Now all nodes produced by the argument to underline have the value 1 for attribute 100—which was arbitrarily chosen. Attribute 458 would have been equally good. Actually, one should use attributes with greater care, i.e. they should be allocated with macros like \newcount, so that one never uses the same attribute for different tasks.

The last action performed by \underline is to register a function in the post_linebreak_filter callback. It does so because the Lua function used to underline clears the callback (as we'll see), so that it is called only on those paragraphs where it is required. It could be called on all paragraphs, but it'd waste TEX's time.

Let's now turn to the Lua functions:

```
get_lines = function (head)
  for line in node.traverse_id(HLIST, head) do
    underline(line.list, line.glue_order,
      line.glue_set, line.glue_sign)
  end
  callback.register
    ("post_linebreak_filter", nil)
  return head
end
```

This first function retrieves all lines in the paragraph and feeds their content to the underline function along with information about glue setting. It then clears the callback and returns the head. This part is nothing we haven't seen in the previous code.

Some nodes might have inherited the attribute's value, although we don't want to underline them: \leftskip, \rightskip, and \parfillskip. These are glue nodes and their subtypes are 8, 9 and 15, respectively. The following function is meant to filter them out. (Note: versions prior to v0.62 had a bug where \leftskip and \rightskip were not properly identified, so item.subtype == 7 should be added to the or conditional below. Both TEX Live 2010 and MikTEX 2.9 uses v0.60, so they are affected.)

```
local good_item = function (item)
  if item.id == GLUE and
   (item.subtype == 8 or item.subtype == 9
```

```
   or item.subtype == 15) then
    return false
  else
    return true
  end
end
```

Now, here's how the `underline` Lua function starts:

```
underline =
  function (head, order, ratio, sign)
  local item = head
  while item do
    if node.has_attribute(item,100)
     and good_item(item) then
      local item_line = node.new(RULE)
      item_line.depth = tex.sp("1.4pt")
      item_line.height = tex.sp("-1pt")
```

The `while` loop is basically the same thing as traversing the list, but we'll sometimes want to skip nodes, so we'll set the `next` one by hand. We scan nodes, and once we've found one with the right value for the attribute (and which is not one of the glues above), we create our rule (with arbitrary dimensions). `tex.sp` turns a dimension (expressed as a string) into scaled points, the native measure for Lua code. How wide should the rule be? The length of the material starting at the current node up to the last node with the right attribute. To find this last node, we use the following loop, and then retrieve the length of that material via `node.dimensions`, which returns the material's length when it is typeset with the text line's glue setting. We use `end_node.next` because the function actually measures up to its last argument's `prev` node.

```
      local end_node = item
      while end_node.next and
       good_item(end_node.next) and
       node.has_attribute(end_node.next, 100) do
        end_node = end_node.next
      end
      item_line.width = node.dimensions
        (ratio, sign, order, item, end_node.next)
```

Finally we insert the line into the list. That's pretty simple: we insert a negative kern (with `subtype` 1, i.e. a handmade kern, not a font kern) as long as the line after the last underlined node, followed by the line itself. This is equivalent to using `\llap` in plain TEX. The end of the code sets the `next` node to be analyzed (including the false part of the overall conditional).

```
      local item_kern = node.new(KERN, 1)
      item_kern.kern = -item_line.width
      node.insert_after(head, end_node,
                        item_kern)
      node.insert_after(head, item_kern,
                        item_line)
      item = end_node.next
    else
      item = item.next
    end
  end
end
```

We could use different values of the attribute to distinguish different underlining styles. To do so, we would still use `node.has_attribute`, since it returns the value of the attribute, or `nil` if the attribute isn't set. That's another exercise left to the reader.

Marginal notes

When a document has comfortable margins and notes are infrequent and short, marginal notes are an elegant and convenient alternative to footnotes. They are best typeset with their first line level with the line in the text to which they refer. However, such a rule cannot be absolute. Suppose for instance that a note is called on the last line of a page, and itself is made of more than one line. If we follow the rule then the note will invade the bottom margin and ruin the design of the page. So it should be shifted up so that its last line is level with the last line of the page. Doing this is also an improvement when the text doesn't fill the page, e.g. at the end of a chapter, even though there might remain space on the page to accommodate the note. The page looks better that way: a note is a note and would be too conspicuous if it were allowed to run without the main text by its side. Ideally, a note should also be shifted up if it runs along a section break, but I'll ignore that case, to keep things simpler. (For an alternative approach in LATEX, see Stephen Hicks' article in TUGboat 30:2.)

Generally marginal notes are typeset in a smaller font size and on a smaller leading than the main text. Since the leading is smaller, some lines of the notes won't be level with the textblock's lines; however, there should be some `cyclical synchronicity' between the two blocks, so that for instance three lines of the main text have the same height as four lines of the note (in TEX terms it would mean, for instance, `\baselineskip` at 12pt and 9pt respectively), and the following lines are level again.

Here, however, I will typeset notes with the same leading as the main text to avoid complications. Extra calculations are required to achieve what's been previously described—nothing very complicated, though. I'll simply use italics to distinguish the notes from the main text.

Margin notes so numerous that they sometimes overlap each other and must be shifted upward should probably be converted to footnotes, all the more as they'll require a number or symbol so the reader can spot where in the main text they refer to—whereas sparse notes don't need such a mark, since they're supposed to start on the same line as the text they comment, with the known exception we're investigating here. However, we can use the code below to shift notes whatever the reason, so we'll leave æsthetics aside and shift all notes (the shift might go wrong if there are stretchable vertical glues on the page, e.g. `\parskip`; that can be amended, and it's left as yet another exercise). We won't allow more than one note per line, though, because that definitely doesn't make sense.

Here's the TEX part of the code:

```
\newcount\notecount
\suppressoutererror=1
\def\note#1{%
  \advance\notecount 1
  \expandafter\newbox
    \csname marginnote_\the\notecount\endcsname
  \expandafter\setbox
    \csname marginnote_\the\notecount\endcsname=
    \vtop{\hsize=4cm
      \rightskip=0pt plus 1fil
      \noindent\it #1}%
  \bgroup
  \attribute100=\expandafter\the
    \csname marginnote_\the\notecount\endcsname
  \vadjust pre {\pdfliteral{}}%
  \egroup
}
```

This might be somewhat unfamiliar, even to advanced TEXies, because what we're doing is preparing the ground for Lua code. First, we choose not to insert the note directly in the paragraph (to be shifted later if necessary). Instead, we store the note in a box. For each note, we create a new box; that might seem somewhat resource-consuming, but there are 65,536 available boxes in LuaTEX, so a shortage seems only a distant possibility. Alternatively, we could store only the source code for the note (in a macro), and typeset it in a box only when we place notes on the page in the output routine, but the asynchronicity between the processing of the main text and the note might lead to trouble.

So we create boxes instead, with proper settings (mostly, a reduced `\hsize`). To allow `\newbox` to appear inside a macro definition in plain TEX, we suppress the `outer` error beforehand; then we set the note in its box with a uniquely defined name (thanks to `\newcount`), and most importantly we set an attribute to the value of the box register and `\vadjust` a literal with that attribute. This literal's only role is to mark the line it comes from, so we'll be able to spot lines with margin notes when needed, along with the box's number (the value of the attribute).

The following Lua function, to be inserted in the `post_linebreak_filter` callback, does exactly that: our special `\pdfliterals` give their attributes to the lines they come from, and are removed. Now, the reader might have wondered why we used the `pre` version of `\vadjust` instead of the default: it's because of a bug in the actual version of LuaTEX (to be fixed in v0.64, I am told): some `prev` fields are sometimes wrong, as would be the case here, and we couldn't link each literal to its line if the latter was before the former. So we use `next` instead. Note that we can't just take for granted that the first `next` node is the line, first because `pre-\vadjusted' material is inserted before the baselineskip glue, and because there might be more adjusted material between the literal and the line. So we recurse over `next` fields until we find a line (i.e. a node `id HLIST`).

```
mark_lines = function (head)
  for mark in node.traverse_id(WHAT, head) do
    local attr = node.has_attribute(mark, 100)
    if attr then
      local item = mark.next
      while item do
        if item.id == HLIST then
          node.set_attribute(item, 100, attr)
          item = nil
        else
          item = item.next
        end
      end
      head = node.remove(head, mark)
    end
  end
  return head
end
```

The following function scans the content of a vertical list, probably box 255, finds the lines that have attribute 100 set to some value, and adds the margin notes to those lines. Remember that our goal is to avoid margin notes running into the space below the textblock (either the bottom margin or the vacant space at the end of a chapter). So we must compute how much space remains to accommodate the note. To do so, we scan the box (the page), starting at the bottom, and accumulate the height and depth of lines and the width of kerns and glues—except kerns and glues that might appear before the last line, i.e. space filling the page. To

do so, we have a `first` boolean that is `true` as long as a line hasn't been found and prevents adding the width of glues and kerns. With `node.slide` we grasp the last node of the list, since we're reading it backward.

```
process_marginalia = function (head)
  local remainingheight, first, item =
    0, true, node.slide(head)
  while item do
    if node.has_field(item, "kern") then
      if not first then
        remainingheight = remainingheight
          + item.kern
      end
    elseif node.has_field(item, "spec") then
      if not first then
        remainingheight = remainingheight
          + item.spec.width
      end
```

Now, if we find a line, we add its depth if and only if it's not the first one we encounter (i.e. the last one on the page), because in that case its depth belongs to the bottom margin. Its height is added later, if and only if the line doesn't take a note.

```
    elseif node.has_field(item, "height") then
      if first then
        first = false
      else
        remainingheight = remainingheight
          + item.depth
      end
```

If attribute 100 is set to some value, then the line takes a note. In that case, we retrieve the box, measure its depth, and compare it to the remaining height. Note that the depth of the box is all its material barring the height of its first line (since we used a `\vtop`), which is exactly what we want: its first line can't go wrong, since it's level with the main text's line from whence it came. We also remove the depth of the last line, since its going into the bottom margin is perfectly ok.

```
     local attr = node.has_attribute(item, 100)
     if attr then
        local note = node.copy(tex.box[attr])
        local upward = note.depth
          - node.tail(note.list).depth
        if upward > remainingheight then
          upward = remainingheight - upward
        else
          upward = 0
        end
```

Now we insert the note box after the line: first, we add a negative vertical kern to account for the upward shift (possibly 0), plus the line's depth and the note's height (i.e. the height of its first line), so it is level with the line. We then set the note's height and depth to 0, so it doesn't take up space on the page. (Since the kern becomes the head of the list, we have to explicitly set `note.list` to it, otherwise TEX still thinks the previous head is the good one.)

```
        local kern = node.new(KERN, 1)
        kern.kern = upward - note.height
          - item.depth
        node.insert_before(note.list,
                           note.list, kern)
        note.list = kern
        note.height, note.depth = 0, 0
```

Finally, we insert the note and set its horizontal `shift` (here it goes into the right margin, but this should depend on whether the page is even or odd), and reset `first` and `remainingheight`, the latter to `upward` so the vertical shift of the current note (if any) is taken into account for the following one. The rest of the code is the end of the `attr` conditional (false, so we add the line's height to the `remainingheight`) and the end of the main loop.

```
        node.insert_after(head, item, note)
        note.shift = tex.hsize + tex.sp("1em")
        first = true
        remainingheight = upward
      else
        remainingheight = remainingheight
          + item.height
      end
    end
    item = item.prev
  end
end
```

When a page is found good, before we ship it out (and before we add inserts too), we feed it to the function, so notes are added. For instance, a very simple output routine would be:

```
\output{%
  \directlua{%
    process_marginalia(tex.box[255].list)
    }%
  \shipout\box255}
```

The important part is, of course, the Lua code.

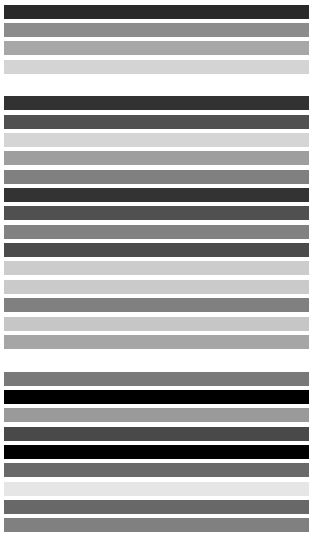
Conclusion

LuaTEX has much to offer: utf-8 encoding, non-tfm fonts, a comfortable programming language, ... Access to TEX's internals is, to me, one of its most valuable features: it enables the user to do things that were previously unthinkable, and gives such control over typography that the software's limitations almost vanish, as if we were working on a hand press—except we don't manipulate metal, but nodes.

A final note: in this paper, functions have been added to callbacks with LuaTEX's bare mechanism. If two functions are added to the same callback this way, the second erases the first. To do this properly, the `luatexbase` package can be used for plain TEX and LATEX, and it is taken care of in ConTEXt.

The next page shows examples of our three programs. First comes the page color, displaying a typeset text and its translation to shades of grey; the second text uses font expansion to show the resulting improvement in justification. Then are examples of underlining and marginal notes. The text used is the first page of Robert Coover's novel *The Adventures of Lucky Pierre*.

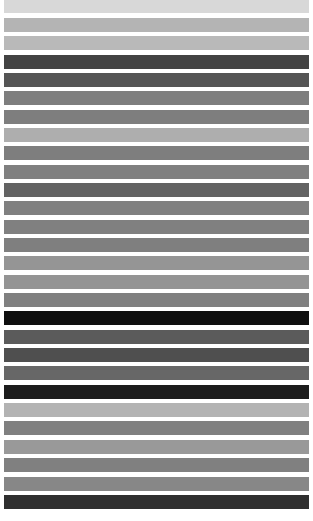
Paul Isambert
Université de la Sorbonne Nouvelle
France
zappathustra (at) free dot fr



In the darkness, softly. A whisper becoming a tone, the echo of a tone. Doleful, incipient lament blowing in the night like a wind, like the echo of a wind, a plainsong wafting silently through the windy chambers of the night, wafting unisonously through the spaced chambers of the bitter night, alas, the solitary city, she that was full of people, thus a distant and hollow epiodion laced with sibilants bewailing the solitary city.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, casting its doubtful luster on this faint white plane, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, affanato, piangevole, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she weepeth.



In the darkness, softly. A whisper becoming a tone, the echo of a tone. Doleful, incipient lament blowing in the night like a wind, like the echo of a wind, a plainsong wafting silently through the windy chambers of the night, wafting unisonously through the spaced chambers of the bitter night, alas, the solitary city, she that was full of people, thus a distant and hollow epiodion laced with sibilants bewailing the solitary city.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, casting its doubtful luster on this faint white plane, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, affanato, piangevole, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she weepeth.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, casting its doubtful luster on this faint white plane, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, **affanato**, **piangevole**, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she **weepeth**.

'Affanato' means 'anguished'
'Piangevole' means 'plaintive'

'Weepeth' is an archaic form of 'weeps'

Toward Subtext

A Mutable Translation Layer for Multi-Format Output

Abstract
The demands of typesetting have shifted significantly since the original inception of TEX. Donald Knuth strove to develop a platform that would prove stable enough to produce the same output for the same input over time (assuming the absence of bugs). Pure TEX is a purely formal language, with no practical notion of the semantic characteristics of the text it is typesetting. The popularity of LATEX is largely related to its attempt to solve this problem. The flexibility of ConTEXt lends it to a great diversity of workflows. However, document creation is not straight-forward enough to lend itself to widespread adoption by a layman audience, nor is it particularly flexible in relation to its translatability into other important output formats such as HTML. Subtext is a proposed system of *generative typesetting* designed for providing an easy to use abstraction for interfacing with TEX, HTML, and other significant markup languages and output formats. By providing a *mutable* translation layer in which both syntax and the actual effects of translation are defined within simple configuration files, the infinitely large set of typographic workflows can be accomodated without being known in advance. At the same time, once a workflow has been designed within the Subtext system, it should enjoy the same long-term stability found in the TEX system itself. This article briefly explains the conditions, motivations, and initial design of the emerging system.

Keywords
generative typesetting, multi-output, translation layer, pre-format

Conditions for Subtext

Subtext arose as a practical conclusion during the writing of my masters thesis in New Media at the Universiteit van Amsterdam.[1]The initial impulse for the thesis itself was to investigate what available media theories existed that could articulate the dynamics of a *generative workflow* pre-occupied with outputting itself in multiple formats. In the case of the thesis, this meant PDF and HTML. Having heard about the translation software Pandoc,[2]I chose to utilize this software in my quest to produce a thesis whose materiality spanned not a single document but multiple files, programs, and 'glue' scripts. In other words, the thesis would not be a *product*, set in proprietary software like MS Word, but a *process* that could self-correct later in the future should a new format come into existence.

The raw fact of text on the computer screen is that, overall, the situation is awful. Screenic text can be divided into three categories: semantic, formal, and WYSIWYG. The semantic formats, for example HTML and XML, are notoriously machine-readable. Text can easily be highlighted, copied, pasted, processed, converted, etc. Yet the largest "reading" software for semantic formats is the web browser. Not a single web browser seems to have bothered to address line-breaking with any sort of seriousness.[3]The ubiquity of HTML, tied with its semantic processibility, means that its importance cannot be ignored as an output format. At this point, not producing an HTML version of a document that one wishes to see widely read is tantamount to removing such widespread reading as an achievable goal. To top off the complexity of the situation, the machine-readability of a semantic document is offset by a distinct reduction of human readability. Asking anyone to write a thesis directly in XML. for instance, is going to be a non-starter.

The second class of text are those defined by their *formal* nature. This is not referring to some buttoned-down attitude, but rather to an opposite directionality in terms of how the text is presented. In semantic markup, the format is not itself responsible for how a display program arranges the text---rather, the display program digests the text in light of its semantic qualities and then lays that text out according to algorithms that can and do vary between programs. The easiest way to describe this approach is that it is *top-down*.

Formal markup, on the other hand, is *bottom-up*. The final display of text is defined by discrete instructions to a program that assembles that text in a highly specific way. TEX is one obvious example of this. Likewise, PostScript and PDF are formal specifications for typesetting text. The immediate drawbacks of formal markups include an often byzantine syntax and a lack of processibility into anything other than the output formats that the formal system knows how to handle. To this day, copy-pasting from a PDF document often leads to awkward extra characters such as linebreaks in the pasted text.

The third class of screenic text system is WYSIWYG. While WYSIWYG is first and foremost a user interface design pattern (and thus can be used to output files in both formal and semantic formats), it is also defines the extremely pervasive Microsoft Word file formats. By positioning the comfort of the user above all other considerations, WYSIWYG finds its strengths in its ease of use and its inherent predictability: whatever you se on the screen should appear exactly that way on paper. By privileging the human to such an extent, however, both translatability and the typographic quality of the text suffer. Since text is intended to always appear exactly as it was input, MS Word can do no calculations for line breaks other than on a per-line basis.[4]Worst of all, WYSIWYG formats (especially those derived from Microsoft products) are difficult to integrate into a generative typesetting workflow which targets many output formats.

Problematics Within Generative Typesetting

Generative typesetting itself emerges from a very specific set of problematics. A primary concern is a reduction in syntax complexity. This is solved by the introduction of a pre-format that provides *sight-level semantics* for specifying desired outcomes in the output formats. For example, the Markdown pre-format was designed such that "a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions."[5]

To demonstrate, while a top-level header in Markdown reads as

```
# My Header #
```

Once converted into HTML the above turns into

```
<h1>My Header</h1>
```

Sight-level semantics rely on visually distinct identifiers. This stands in sharp contrast to both HTML/XML and TEX, which rely on distinct tags combined with reserved characters. In short, this approach to semantic formatting relies on utilizing *more* reserved characters than these other systems. Which characters are chosen and the nature of their organization is an attempt to strike a balance between both readability and processibility. Like WYSIWYG, sight-level semantics represent a redistribution of agency between the human and the machine. Unlike WYSIWYG, the utilization of Markdown implies an intention for translating it into other formats.

The second problematic is an inevitable result of the first: there is *always* an edge case. Take as an example a variation on the code I've already shown. Say that instead of converting to HTML, one would rather generate a PDF using ConTEXt. Seems straight-forward right?

```
# My Header #
```

The above should simply convert into the top-level equivalent in ConTEXt. But wait.. That would be a matter of what one was trying to accomplish, wouldn't it?

After all, the above Markdown snippet could easily refer to

```
\subject{My Header}
```

or

```
\section{My Header}
```

or

```
\chapter{My Header}
```

or even

```
\title{My Header}
```

What is the solution here? Should a reserved character be adopted for each of these cases? Questions of how to deal with such edge cases are intrinsically tied to the translation layer itself: because all format translation occurs within the translation layer, it is the decisions which that layer makes that determine how edge cases are handled.

`Pandoc` provides command-line switches for turning on numbered sections and for determining the top-level "section." However, were one to desire that a custom command or macro be used in place of any of the above, a knowledge of Haskell is required to write scripts or otherwise modify the way that `Pandoc` converts its inputs. Other tricks can be employed, such as the introduction of a 'glue' layer based in a script which solves certain edge cases with regular expressions and `if` statements. From the standpoint of a generative typesetting workflow that does not require programming expertise, these solutions for edge cases are far from optimal.

One Mutable Interface to Produce Them All

Today the largest demands of digital publishing revolve around flexibility. The vast array of existing and on-coming e-readers is but one example of this. More general concerns include the necessity of both machine-readable formats and typographically sound documents. Currently this means HTML/XML and PDF.

Yet once e-readers are brought into the mix, the ePub format becomes imperative.

Yet while ePub is the most accepted format for e-reader publishing today, there is always the chance (one might even say inevitability) that a new format will become standard in the future. *Future-proofing* is a significant advantage of a generative typesetting workflow, but the programming-required nature of edge cases--and, indeed, any modification to the translation layer--decreases the adoptability of generative typesetting for non-technical fields such as the humanities.

The solution that `Subtext` proposes is to disengage both the interface to the translation layer as well as the effects of that layer. In this way `Subtext` can be seen as a very thin layer, one that takes interface primitives from a configuration file and translates them into an AST. The effects of this AST are then interpreted according to rules defined in a seperate configuration file. This file explains what should literally appear in the output file for any given AST element.

One immediately obvious benefit of this approach is the capacity to internationalize the pre-format with ease. For standard Markdown, `Subtext` would define the effect of American quotation marks ("x") as `\quotation{x}`. The interface file could be quickly modified to intepret double angle quotation marks (« x ») in the same way (`\quotation{x}`).

The effects configuration can also incorporate 'setup' requirements. If a generative typesetting workflow involved dealing with documents of either English or French, then it would be known that when double-angle quotation marks are used in the pre-format that the resulting document should have French style punctuation and spacing. The `Subtext` interpreter would then add

```
\mainlanguage[french]
\setcharacterspacing[frenchpunctuation]
```

to the pre-amble of a ConTEXt document. Likewise, specific character spacing settings could be added to the CSS of an HTML or ePub output file.

The mutability of this system is its primary characteristics. Specific text elements need not fit a pre-existing notion, as new rules can be invented and interpreted within configuration files. This capacity to 'unlock' the translation layer into an instrinsically customizable tool not only guarantees future-proofing: it also allows for highly specific workflows to be developed, as the interface and effects can be custom-crafted according to the requirements of the task.

Preliminary Thoughts on Implementation

There has yet to be a line of code committed to `Subtext`. At present it is a simple design impulse, with a variety of expectations and desires tied into a proposed means of accomplishing a more fluid and responsive generative typesetting workflow. This does not mean, however, that there has yet to be any thought put into the platforms that will underpin `Subtext`.

The first choice is the programming language. Considering the importance of parsing, grammar, and metaprogramming functionality to the implementation of a mutable translation layer, my first impulse is to write `Subtext` in Perl 6. This might come as a slight shock, but that shock should not last beyond an exploration into the power of Perl 6 grammars.[6] A robust, rules-based grammar engine was one of the top details for which Perl 6 was designed. Combined with features such as multi-method dispatch and other metaprogramming conveniences, Perl 6 is primed to host `Subtext`. Barriers to entry include a lack of documentation, but at the same time the "scene" around the programming language is small and extremely helpful. Another downside is the current speed of the language, though that is an aspect which is addressed with each monthly release. In general, the idea of Perl 6 is that it presents a mutable interface to its own programming capacities. The sympatico between the two projects is thus too significant to deny.

The configuration files themselves present a slight complication, as they need to be highly parse-able despite potentially containing every reserved character known to any programming language or syntax currently known. Thankfully, there have been many attempts to achieve this robustness. One that fits particularly well into the generative typesetting mindset which `Subtext` exemplifies is YAML (Yet Another Markup Language[8]). YAML is intended to facilitate everything from configuration files to object persistence through a human-friendly syntax. The flexibility of such a system will no doubt provide a solid foundation for implementing `Subtext`.

Additionally, there will be a standard syntax for `Subtext`. That is, there will be a defined pre-format that ships with the system. This standard syntax will include bibliographic functionality that is currently limited or non-existent in most multi-output workflows.

Longer-term goals include a web interface for dealing with the input files. Such a system would likely integrate the newly-open sourced Etherpad software for online editing. This would be tied to a version control interface based on `git` that would fill in the functionality that MS Word's 'Track Changes' system currently provides. Ideally, integrated into this system would be a real-time parser such as exhibited in the AJAX-ified interface of the WMD[7] editor, which renders the HTML output of Markdown text in real-time within the same browser window. This functionality is likely constrained by the speed of the Rakudo Perl 6 imple-

mentation. However, it is conceivable that the standard `Subtext` syntax can be parsed in JavaScript. This means that highly customized workflows would not be able to enjoy a real-time feedback interface in the near-term future. This seems to be a small trade-off for the kind of flexibility this system can enable in generative typesetting, and could easily find itself solved over the course of the continuance of Moore's Law.

Request For Comments

Though `Subtext` aims to be useful for dealing with *n+1* different output formats, initial development will concern itself with simply HTML and ConTEXt outputs. Together these two encompass the primary formats of concern. LaTeX, ePub, and others can easily be added by simply defining a new set of effects.

The standard syntax has yet to be designed. Any comments or suggestions in this regard (or concerning any of what has been discussed) will be very useful. At this early conceptual stage where nothing is locked down except for the core ideas, there is a great potential for shaping the eventual system without worrying about any legacy functionality. Please do not hesitate to send me your thoughts!

Notes

1. The thesis, titled *Grammars of Process: Agency, Collective Becoming, and the Organization of Software* is available at http://mastersofmedia.hum.uva.nl/2010/09/17/grammars-of-process-agency-collective-becoming-and-the-organization-of-software-2/.
2. `Pandoc` is the only text format translation tool that currently translates into ConTEXt. It is written by John MacFarlene and is available at http://johnmacfarlane.net/pandoc/.
3. For an easy example of this, just set `text-align: justify;` in the CSS for <p> tags in an HTML document.
4. A clearly notable exception to this is Adobe InDesign and other WYSIWYG Desktop Publishing tools, in which line-breaking must be taken more seriously. In terms of "end-user" level document creation, however, the statement that linebreaking is lacking in WYSIWYG stands.
5. Markdown: http://daringfireball.net/projects/markdown/.
6. Perl 6: http://perl6.org. For an example of Perl 6 grammars, see http://perl6advent.wordpress.com/2009/12/21/day-21-grammars-and-actions/ from the 'Perl 6 Advent Calendar,' a great place to start learning about the potentials of this language.
7. WMD - The WYSIYWM Markdown Editor: http://wmd-editor.com/.
8. YAML: http://yaml.org.

John C. Haltiwanger
john.haltiwanger@gmail.com

Typesetting in Lua using LuaTEX

Introduction

Sometimes you hear folks complain about the TEX input language, i.e. the backslashed commands that determine your output. Of course, when alternatives are being discussed every one has a favourite programming language. In practice coding a document in each of them triggers similar sentiments with regards to coding as TEX itself does.

So, just for fun, I added a couple of commands to ConTEXt MkIV that permit coding a document in Lua. In retrospect it has been surprisingly easy to implement a feature like this using metatables. Of course it's a bit slower than using TEX as input language but sometimes the Lua interface is more readable given the problem at hand.

After a while I decided to use that interface in non-critical core ConTEXt code and in styles (modules) and solutions for projects. Using the Lua approach is sometimes more convenient, especially if the code mostly manipulates data. For instance, if you process xml files of database output you can use the interface that is available at the TEX end, or you can use Lua code to do the work, or you can use a combination. So, from now on, in ConTEXt you can code your style and document source in (a mixture of) TEX, xml, MetaPost and in Lua.

In this article I will introduce typesetting in Lua, but as we rely on ConTEXt it is unavoidable that some regular ConTEXt code shows up. The fact that you can ignore backslashes does not mean that you can do without knowledge of the underlying system. I assume the user is somewhat familiar with this macro package.

Some basics

To start with, I assume that you have either the so called ConTEXt minimals installed or TEXLive. You only need LuaTEX and can forget about installing pdfTEX or XƎTEX, which saves you some megabytes and hassle. Now, from the user's perspective a ConTEXt run goes like:

```
context yourfile
```

and by default a file with suffix `tex` will be processed. There are however a few other options:

```
context yourfile.xml
context yourfile.rlx --forcexml
context yourfile.lua
context yourfile.pqr --forcelua
context yourfile.cld
context yourfile.xyz --forcecld
```

When processing a Lua file the given file is loaded and just processed. This option will seldom be used as it is way more efficient to let `mtxrun` process that file. However, the last two variants are what we will discuss here. The suffix `cld` is a shortcut for ConTEXt Lua Document.

A simple cld file looks like this:

```
context.starttext()
context.chapter("Hello There!")
context.stoptext()
```

So yes, you need to know the ConTEXt commands in order to use this mechanism. In spite of what you might expect, the codebase involved in this interface is not that large. If you know ConTEXt, and if you know how to call commands, you basically can use this Lua method.
The examples that I will give are either (sort of) standalone, that is, they are dealt with from Lua, or they are run within this document. Therefore you will see two patterns. If you want to make your own documentation, then you can use this variant:

```
\startbuffer
context("See this!")
\stopbuffer

\typebuffer \ctxluabuffer
```

I use anonymous buffers here but you can also use named ones. The other variant is:

```
\startluacode
context("See this!")
\stopluacode
```

This will process the code directly. Of course we could have encoded this document completely in Lua but that is not much fun for a manual.

The main command

There are a few rules that you need to be aware of. First of all no syntax checking is done. Second you need to know what the given commands expects in terms of arguments. Third, the type of your arguments matters:

```
nothing : just the command, no arguments
string  : an argument with curly braces
array   : a list between square backets (sometimes optional)
hash    : an assignment list between square brackets
boolean : when true a newline is inserted
        : when false, omit braces for the next argument
```

In the code above you have seen examples of this but here are some more:

```
context.chapter("Some title")
context.chapter({ "first" }, "Some title")
context.startchapter({ title = "Some title", label = "first" })
```

This blob of code is equivalent to:

```
\chapter{Some title}
\chapter[first]{Some title}
\startchapter[title={Some title},label=first]
```

You can simplify the third line of the Lua code to:

```
context.startchapter { title = "Some title", label = "first" }
```

In case you wonder what the distinction is between square brackets and curly braces: the first category of arguments concerns settings or lists of options or names of instances while the second category normally concerns some text to be typeset.

Strings are interpreted as TEX input, so:

```
context.mathematics("\\sqrt{2^3}")
```

or, if you don't want to escape:

```
context.mathematics([[\sqrt{2^3}]])
```

is okay. As TEX math is a language in its own and a de-facto standard way of inputting math this is quite natural, even at the Lua end.

Spaces and Lines

In a regular TEX file, spaces and newline characters are collapsed into one space. At the Lua end the same happens. Compare the following examples. First we omit spaces:

```
context("left")
context("middle")
context("right")
```

leftmiddleright

Next we add spaces:

```
context("left")
context(" middle ")
context("right")
```

left middle right

We can also add more spaces:

```
context("left ")
context(" middle ")
context(" right")
```

left middle right

In principle all content becomes a stream and after that the TEX parser will do its normal work: collapse spaces unless configured to do otherwise. Now take the following code:

```
context("before")
context("word 1")
context("word 2")
```

```
context("word 3")
context("after")
```

beforeword 1word 2word 3after

Here we get no spaces between the words at all, which is what we expect. So, how do we get lines (or paragraphs)?

```
context("before")
context.startlines()
context("line 1")
context("line 2")
context("line 3")
context.stoplines()
context("after")
```

before

line 1line 2line 3

after

This does not work out well, as again there are no lines seen at the TEX end. Newline tokens are injected by passing `true` to the `context` command:

```
context("before")
context.startlines()
context("line 1") context(true)
context("line 2") context(true)
context("line 3") context(true)
context.stoplines()
context("after")
```

before

line 1
line 2
line 3

after

Don't confuse this with:

```
context("before") context.par()
context("line 1") context.par()
context("line 2") context.par()
context("line 3") context.par()
context("after")  context.par()
```

before
line 1
line 2
line 3
after

There we use the regular \par command to finish the current paragraph and normally you will use that method. In that case, when set, whitespace will be added between paragraphs.

Direct output

The ConTEXt user interface is rather consistent and the use of special input syntaxes is discouraged. Therefore, the Lua interface using tables and strings works quite well. However, imagine that you need to support some weird macro (or a primitive) that does not expect its argument between curly braces or brackets. The way out is to precede an argument by another one with the value false. We call this the direct interface. This is demonstrated in the following example.

```
\unexpanded\def\bla#1{[#1]}

\startluacode
context.bla(false,"***")
context.par()
context.bla("***")
\stopluacode
```

This results in:

[*]**
[***]

Here, the first call results in three * being passed, and #1 picks up the first token. The second call to bla gets {***} passed so here #1 gets the triplet. In practice you will seldom need the direct interface.

In ConTEXt for historical reasons, combinations have the following syntax:

```
\startcombination % optional specification, like [2*3]
  {\framed{content one}} {caption one}
  {\framed{content two}} {caption two}
\stopcombination
```

You can also say:

```
\startcombination
  \combination {\framed{content one}} {caption one}
  \combination {\framed{content two}} {caption two}
\stopcombination
```

When coded in Lua, we can feed the first variant as follows:

```
context.startcombination()
  context.direct("one","two")
  context.direct("one","two")
context.stopcombination()
```

To give you an idea what this looks like, we render it:

one one
two two

So, the direct function is basically a no-op and results in nothing by itself. Only arguments are passed. Equivalent, but a bit more ugly looking, is:

```
context.startcombination()
  context(false,"one","two")
  context(false,"one","two")
context.stopcombination()
```

Catcodes

If you are familiar with TEX's inner working, you will know that characters can have special meanings. This meaning is determined by the characters catcode.

```
context("$x=1$")
```

This gives: $x = 1$ because the dollar tokens trigger inline math mode. If you think that this is annoying, you can do the following:

```
context.pushcatcodes("text")
context("$x=1$")
context.popcatcodes()
```

Now we get: $x=1$. There are several catcode regimes of which only a few make sense in the perspective of the cld interface.

`ctx, ctxcatcodes, context`	the normal ConTEXt catcode regime
`prt, prtcatcodes, protect`	the ConTEXt protected regime, used for modules
`tex, texcatcodes, plain`	the traditional (plain) TEX regime
`txt, txtcatcodes, text`	the ConTEXt regime but with less special characters
`vrb, vrbcatcodes, verbatim`	a regime specially meant for verbatim
`xml, xmlcatcodes`	a regime specially meant for xml processing

In the second case you can still get math:

```
context.pushcatcodes("text")
context.mathematics("x=1")
context.popcatcodes()
```

When entering a lot of math you can also consider this:

```
context.startimath()
context("x")
context("=")
context("1")
context.stopimath()
```

Module writers of course can use `unprotect` and `protect` as they do at the TEX end. As we've seen, a function call to `context` acts like a print, as in:

```
context("test ")
context.bold("me")
context(" first")
```

test **me** first

When more than one argument is given, the first argument is considered a format conforming the `string.format` function.

```
context.startimath()
context("%s = %0.5f",utf.char(0x03C0),math.pi)
context.stopimath()
```

$\pi = 3.14159$

This means that when you say:

```
context(a,b,c,d,e,f)
```

the variables `b` till `f` are passed to the format and when the format does not call for them, they will not end up in your output.

```
context("%s %s %s",1,2,3)
context(1,2,3)
```

The first line results in the three numbers being typeset, but in the second case only the number 1 is typeset.

Why we need functions

In a previous section we introduced functions as arguments. At first sight this feature looks strange but you need to keep in mind that a call to a `context` function has no direct consequences. It generates TEX code that is executed after the current Lua chunk ends and control is passed back to TEX. Take the following code:

```
context.framed( {
    frame = "on",
    offset = "5mm",
    align = "middle"
  },
  context.input("knuth")
)
```

We call the function `framed` but before the function body is executed, the arguments get evaluated. This means that `input` gets processed before `framed` gets done. As a result there is no second argument to `framed` and no content gets passed: an error is reported. This is why we need the indirect call:

```
context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input("knuth") end
)
```

This way we get what we want:

> Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important. But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

The function is delayed till the `framed` command is executed. If your applications use such calls a lot, you can of course encapsulate this ugliness:

```
mycommands = mycommands or { }

function mycommands.framed_input(filename)
  context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input(filename) end
end

mycommands.framed_input("knuth")
```

Of course you can nest function calls:

```
context.placefigure(
  "caption",
  function()
    context.framed( {
      frame = "on",
      align = "middle"
    },
      function() context.input("knuth") end
    )
  end
)
```

Or you can use a more indirect method:

```
function text()
  context.framed( {
      frame = "on",
      align = "middle"
    },
    function() context.input("knuth") end
  )
end

context.placefigure(
  "none",
  function() text() end
)
```

You can develop your own style and libraries just like you do with regular Lua code.

How we can avoid them

As many nested functions can obscure the code rather quickly, there is an alternative. In the following examples we use `test`:

```
\def\test#1{[#1]}

context.test("test 1",context(" test 2a "),"test 3")
```

This gives: test 2a [test 1]test 3. As you can see, the second argument is executed before the encapsulating call to `test`. So, we should have packed it into a function but here is an alternative:

```
context.test("test 1",context.delayed(" test 2a "),"test 3")
```

Now we get: [test 1] test 2a test 3. We can also delay functions themselves, look at this:

```
context.test("test 1",context.delayed.test(" test 2b "),"test 3")
```

The result is: [test 1][test 2b]test 3. This feature also conveniently permits the use of temporary variables, as in:

```
local f = context.delayed.test(" test 2c ")
context("before",f,"after")
```

Of course you can limit the amount of keystrokes even more by creating a shortcut:

```
local delayed = context.delayed

context.test("test 1",delayed.test(" test 2 "),"test 3")
context.test("test 4",delayed.test(" test 5 "),"test 6")
```

So, if you want you can produce rather readable code and readability of code is one of the reasons why Lua was chosen in the first place.
There is also another mechanism available. In the next example the second argument is actually a string.

```
local nested = context.nested

context.test("test 8",nested.test("test 9"),"test 10")
```

There is a pitfall here: a nested context command needs to be flushed explicitly, so in the case of:

```
context.nested.test("test 9")
```

a string is created but nothing ends up at the TEX end. Flushing is up to you. Beware: nested only works with the regular ConTEXt catcode regime.

Trial typesetting

Some typesetting mechanisms demand a preroll. For instance, when determining the most optimal way to analyse and therefore typeset a table, it is necessary to

typeset the content of cells first. Inside ConTEXt there is a state tagged 'trial typesetting' which signals other mechanisms that for instance counters should not be incremented more than once.
Normally you don't need to worry about these issues, but when writing the code that implements the Lua interface to ConTEXt, it definitely had to be taken into account as we either or not can free cached (nested) functions.
You can influence this caching to some extend. If you say

```
function()
  context("whatever")
end
```

the function will be removed from the cache when ConTEXt is not in the trial typesetting state. You can prevent *any* removal of a function by returning `true`, as in:

```
function()
  context("whatever")
  return true
end
```

Whenever you run into a situation that you don't get the outcome that you expect, you can consider returning `true`. However, keep in mind that it will take more memory, something that only matters on big runs. You can force flushing the whole cache by:

```
context.restart()
```

An example of an occasion where you need to keep the function available is in repeated content, for instance in headers and footers.

```
context.setupheadertexts {
  function()
    context.pagenumber()
    return true
  end
}
```

Of course it is not needed when you use the following method:

```
context.pagenumber("pagenumber")
```

Because here ConTEXt itself deals with the content driven by the keyword `pagenumber`.

Variables

Normally it makes most sense to use the English version of ConTEXt. The advantage is that you can use English keywords, as in:

```
context.framed( {
    frame = "on",
  },
  "some text"
)
```

If you use the Dutch interface it looks like this:

```
context.omlijnd( {
    kader = "aan",
  },
  "wat tekst"
)
```

A rather neutral way is:

```
context.framed( {
    frame = interfaces.variables.on,
  },
  "some text"
)
```

But as said, normally you will use the English user interface so you can forget about these matters. However, in the ConTEXt core code you will often see the variables being used this way because there we need to support all user interfaces.

Modes

Context carries a concept of modes. You can use modes to create conditional sections in your style (and/or content). You can control modes in your styles or you can set them at the command line or in job control files. When a mode test has to be done at processing time, then you need constructs like the following:

```
context.doifmodeelse( "screen",
  function()
      ... -- mode == screen
  end,
  function()
      ... -- mode ~= screen
  end
)
```

However, often a mode does not change during a run, and then we can use the following method:

```
if tex.modes["screen"] then
  ...
else
  ...
end
```

Watch how the `modes` table lives in the `tex` namespace. We also have `systemmodes`. At the TEX end these are mode names preceded by a `*`, so the following code is similar:

```
if tex.modes["*mymode"] then
  -- this is the same
elseif tex.systemmodes["mymode"] then
  -- test as this
else
  -- but not this
end
```

Inside ConTEXt we also have so called constants, and again these can be consulted at the Lua end:

```
if tex.constants["someconstant"] then
  ...
else
  ...
end
```

But you will hardly need these and, as they are often not public, their meaning can change, unless of course they *are* documented as public.

Token lists

There is normally no need to mess around with nodes and tokens at the Lua end yourself. However, if you do, then you might want to flush them as well. Say that at the TEX end we have said:

```
\toks0 = {Don't get \inframed{framed}!}
```

Then at the Lua end you can say:

```
context(tex.toks[0])
```

and get: Don't get framed! In fact, token registers are exposed as strings so here, register zero has type `string` and is treated as such.

```
context("< %s >",tex.toks[0])
```

This gives: < Don't get framed! >. But beware, if you go the reverse way, you don't get what you might expect:

```
tex.toks[0] = [[\framed{oeps}]]
```

If we now say `\the\toks0` we will get `Don't get framed!` as all tokens are considered to be letters.

Node lists

If you're not deep into TEX you will never feel the need to manipulate nodelists yourself, but you might want to flush boxes. As an example we put something in box zero (one of the scratch boxes).

```
\setbox0 = \hbox{Don't get \inframed{framed}!}
```

At the TEX end you can flush this box (`\box0`) or take a copy (`\copy0`). At the Lua end you would do:

```
context.copy()
context.direct(0)
```

or:

```
context.copy(false,0)
```

but this works as well:

```
context(node.copy_list(tex.box[0]))
```

So we get: Don't get framed! If you do:

```
context(tex.box[0])
```

you also need to make sure that the box is freed but let's not go into those details now.

Styles

Say that you want to typeset a word in a bold font. You can do that this way:

```
context("This is ")
context.bold("important")
context("!")
```

Now imagine that you want this important word to be in red too. As we have a nested command, we end up with a nested call:

```
context("This is ")
context.bold(function() context.color( { "red" }, "important") end)
context("!")
```

or

```
context("This is ")
context.bold(context.delayed.color( { "red" }, "important"))
context("!")
```

In that case it's good to know that there is a command that combines both features:

```
context("This is ")
context.style( { style = "bold", color = "red" }, "important")
context("!")
```

But that is still not convenient when we have to do that often. So, you can wrap the style switch in a function.

```
local function mycommands.important(str)
    context.style( { style = "bold", color = "red" }, str )
end

context("This is ")
mycommands.important( "important")
context(", and ")
mycommands.important( "this")
context(" too !")
```

Or you can setup a named style:

```
context.setupstyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.style( { "important" }, "important")
context(", and ")
context.style( { "important" }, "this")
context(" too !")
```

Or even define one:

```
context.definestyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.important("important")
context(", and ")
context.important("this")
context(" too !")
```

This last solution is especially handy for more complex cases:

```
context.definestyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.startimportant()
context.inframed("important")
context.stopimportant()
context(", and ")
context.important("this")
context(" too !")
```

This is **important**, and **this** too !

A complete example

One day my 6 year old niece Lorien was at the office and wanted to know what I was doing. As I knew she was practicing calculus at school I wrote a quick and dirty script to generate sheets with exercises. The most impressive part was that the answers were included. It was a rather braindead bit of Lua, written in a few minutes, but the weeks after I ended up running it a few more times, for her and her friends, every time a bit more difficult and also using different calculus. It was that script that made me decide to extend the basic cld manual into this more extensive document. We generate three columns of exercises. Each exercise is a row in a table. The last argument to the function determines if answers are shown.

```
local random = math.random

local function ForLorien(n,maxa,maxb,answers)
  context.startcolumns { n = 3 }
  context.starttabulate { "|r|c|r|c|r|" }
  for i=1,n do
    local sign = random(0,1) > 0.5
    local a, b = random(1,maxa or 99), random(1,max or maxb or 99)
    if b > a and not sign then a, b = b, a end
    context.NC()
    context(a)
```

```
    context.NC()
    context.mathematics(sign and "+" or "-")
    context.NC()
    context(b)
    context.NC()
    context("=")
    context.NC()
    context(answers and (sign and a+b or a-b))
    context.NC()
    context.NR()
  end
  context.stoptabulate()
  context.stopcolumns()
  context.page()
end
```

This is a typical example of where it's more convenient to write the code in Lua that in TEX's macro language. As a consequence setting up the page also happens in Lua:

```
context.setupbodyfont {
  "palatino",
  "14pt"
}
context.setuplayout {
  backspace = "2cm",
  topspace  = "2cm",
  header    = "1cm",
  footer    = "0cm",
  height    = "middle",
  width     = "middle",
}
```

At this point, we need to generate the document. There is a pitfall here: we need to use the same random number for the exercises and the answers, so we freeze and defrost it. Functions in the commands namespace implement functionality that is used at the TEX end but better can be done in Lua than in TEX macro code. Of course these functions can also be used at the Lua end.

```
context.starttext()
  local n = 120
  commands.freezerandomseed()
  ForLorien(n,10,10)
  ForLorien(n,20,20)
  ForLorien(n,30,30)
  ForLorien(n,40,40)
  ForLorien(n,50,50)
  commands.defrostrandomseed()
  ForLorien(n,10,10,true)
  ForLorien(n,20,20,true)
  ForLorien(n,30,30,true)
  ForLorien(n,40,40,true)
  ForLorien(n,50,50,true)
context.stoptext()
```

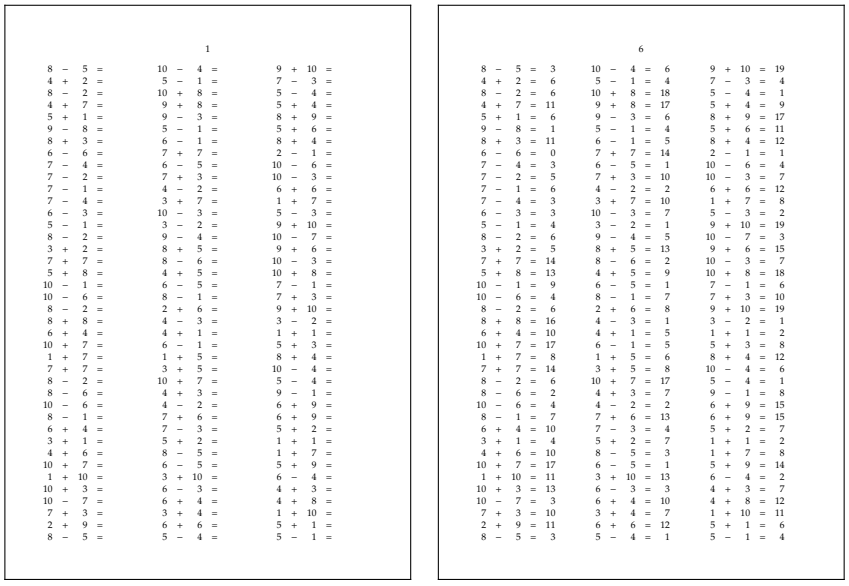


Figure 1 Lorien's challenge.

A few pages of the result are shown in figure 1. In the ConTEXt distribution more advanced version can be found in `s-edu-01.cld` as I was also asked to generate multiplication and table exercises. I also had to make sure that there were no duplicates on a page as she complained that was not good. There a set of sheets is generated with:

```
moduledata.educational.calculus.generate {
  name     = "Bram Otten",
  fontsize = "12pt",
  columns  = 2,
  run      = {
    { method = "bin_add_and_subtract", maxa =   8, maxb =   8 },
    { method = "bin_add_and_subtract", maxa =  16, maxb =  16 },
    { method = "bin_add_and_subtract", maxa =  32, maxb =  32 },
    { method = "bin_add_and_subtract", maxa =  64, maxb =  64 },
    { method = "bin_add_and_subtract", maxa = 128, maxb = 128 },
  },
}
```

Graphics

If you are familiar with ConTEXt, which by now probably is the case, you will have noticed that it integrates the MetaPost graphic subsystem. Drawing a graphic is not that complex:

```
context.startMPcode()
context [[
  draw
    fullcircle scaled 1cm
    withpen pencircle scaled 1mm
    withcolor .5white
```

```
    dashed dashpattern (on 2mm off 2mm) ;
 ]]
context.stopMPcode()
```

We get a gray dashed circle rendered with an one millimeter thick line:

So, we just use the regular commands and pass the drawing code as strings. Although MetaPost is a rather normal language and therefore offers loops and conditions and the lot, you might want to use Lua for anything else than the drawing commands. Of course this is much less efficient, but it could be that you don't care about speed. The next example demonstrates the interface for building graphics piecewise.

```
context.resetMPdrawing()

context.startMPdrawing()
context([[fill fullcircle scaled 5cm withcolor (0,0,.5) ;]])
context.stopMPdrawing()

context.MPdrawing("pickup pencircle scaled .5mm ;")
context.MPdrawing("drawoptions(withcolor white) ;")

for i=0,50,5 do
  context.startMPdrawing()
  context("draw fullcircle scaled %smm ;",i)
  context.stopMPdrawing()
end

for i=0,50,5 do
  context.MPdrawing("draw fullsquare scaled " .. i .. "mm ;")
end

context.MPdrawingdonetrue()

context.getMPdrawing()
```

This gives:

I the first loop we can use the format options associated with the simple `context` call. This will not work in the second case. Even worse, passing more than one argument will definitely give a faulty graphic definition. This is why we have a special interface for MetaFun. The code above can also be written as:

```
local metafun = context.metafun
```

```
metafun.start()

metafun("fill fullcircle scaled 5cm withcolor %s ;",
    metafun.color("darkblue"))

metafun("pickup pencircle scaled .5mm ;")
metafun("drawoptions(withcolor white) ;")

for i=0,50,5 do
  metafun("draw fullcircle scaled %smm ;",i)
end

for i=0,50,5 do
  metafun("draw fullsquare scaled %smm ;",i)
end

metafun.stop()
```

Watch the call to color, this will pass definitions at the TEX end to MetaPost. Of course you really need to ask yourself "Do I want to use MetaPost this way?" Using Lua loops instead of MetaPost ones makes much more sense in the following case:

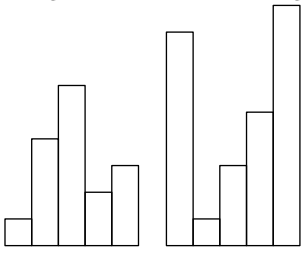
```
local metafun = context.metafun

function metafun.barchart(t)
  metafun.start()
  local t = t.data
  for i=1,#t do
    metafun("draw unitsquare xyscaled(%s,%s) shifted (%s,0);",
      10, t[i]*10, i*10)
  end
  metafun.stop()
end

local one = { 1, 4, 6, 2, 3, }
local two = { 8, 1, 3, 5, 9, }

context.startcombination()
  context.combination(metafun.delayed.barchart { data = one }, "one")
  context.combination(metafun.delayed.barchart { data = two }, "two")
context.stopcombination()
```

We get two barcharts alongside:



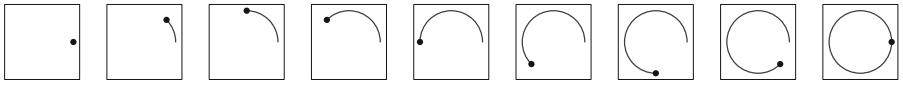
one two

```
local template = [[
  path p, q ; color c[] ;
  c1 := \MPcolor{darkblue} ;
  c2 := \MPcolor{darkred} ;
  p := fullcircle scaled 50 ;
```

```
  l := length p ;
  n := %s ;
  q := subpath (0,%s/n*l) of p ;
  draw q withcolor c2 withpen pencircle scaled 1 ;
  fill fullcircle scaled 5 shifted point length q of q withcolor c1 ;
  setbounds currentpicture to unitsquare shifted (-0.5,-0.5) scaled 60 ;
  draw boundingbox currentpicture withcolor c1 ;
  currentpicture := currentpicture xsized(1cm) ;
]]

local function steps(n)
  for i=0,n do
    context.metafun.start()
      context.metafun(template,n,i)
    context.metafun.stop()
    if i < n then
        context.quad()
    end
  end
end

context.hbox(function() steps(8) end)
```



To some extent we fool ourselves with this kind of Luafication of MetaPost code. Of course we can make a nice MetaPost library and put the code in a macro instead. In that sense, doing this in ConTEXt directly often gives better and more efficient code. Of course you can use all relevant commands in the Lua interface, like:

```
context.startMPpage()
  context("draw origin")
  for i=0,100,10 do
    context("..{down}(%d,0)",i)
  end
  context(" withcolor \\MPcolor{darkred} ;")
context.stopMPpage()
```

to get a graphic that has its own page. Don't use the `metafun` namespace here, as it will not work here. This drawing looks like:

Hans Hagen

Processing "Computed" Texts

Abstract
This article is a comparison of methods that may be used to derive texts to be typeset by a word processor. By 'derive', we mean that such texts are extracted from a larger structure, which can be viewed as a database. The present standard for such a structure uses an XML-like format, and we give an overview of the available tools for this derivation task.

Keywords
Typesetting computed texts, TEX, LaTEX, ConTEXt, XƎTEX, LuaTEX, XML, XSLT, character maps, XQuery, XSL-FO.

Introduction

Formats based on the TEX typesetting engine—e.g., *Plain TEX* [27], or LaTEX [30], or ConTEXt [8], or LuaTEX [9]—are known as wonderful tools to get high-quality print outputs. Of course, they have been initially designed to typeset texts directly written by end-users. But other texts may be *generated dynamically*, in the sense that they result from some *computation* applied to more data, in particular, items belonging to databases. A very simple example is given by a bill computed by means of a spreadsheet program like Microsoft Excel: the master file is an .xls or .xlsx file—that is, all information about data is centralised into this file—but we may wish such a bill to be typeset nicely using a word processor comparable with LaTEX[1].

We personally experienced a more significant example: at the University of Franche-Comté, we manage the projects proposed to Computer Science students in several degrees, this curriculum being located at Besançon, in the East of France. That is, we collect projects' proposals, control the assignment of student groups to projects. Then, at the semester's end, we organise the projects' oral defences, and rate students from information transmitted by jurys. During projects, information is transmitted to students and projects' supervisors either on the Web, or by means of printed documents. Managing only a list of project specifications and enriching it progressively is insufficient: it is better for oral defences' announcement to be shown with respect to defences' chronological order, and this order is unknown when projects are proposed. Likewise, we may wish to present the grades received by students according to the decreasing order of these grades, or according to students' alphabetical order, other criteria being possible, too. In these cases, we have to perform a sort operation before typesetting the result. These examples are not limitative: other operations related to 'classical' programming may be needed if we are only interested in a subset of the information concerning projects, for example, extracting projects proposed by companies, not by people working at our university.

It is well-known that TEX's language is not very suitable for tasks directly related to programming[2]. A better idea is to use a format suitable for information management, with interface tools serving several purposes. Database formats could be used, but presently, the indisputable standard to model such formats is XML[3], providing a rich toolbox for this kind of task. In particular, this toolbox provides the XPath language [38], this language's expressions allow parts of an XML document to be addressed precisely. But these tools related to XML have advantages and drawbacks: we are going thoroughly into these points in this article, which is a revised, updated, and extended version of [19]. Reading this article requires only basic knowledge about (LA)TEX and the tools related to XML.

A simple example

The examples given in the introduction are 'real' applications and belong to our framework: Microsoft Excel can generate XML files[4], and we personally manage students' projects by means of a master file using XML-like syntax. However, for sake of simplicity, we consider an easier example for the present article, pictured in Figure 1. This XML text—a file ds.xml—describes some items of a series of stories—*Doc Savage*—first published as *pulps* in the 1930's, then republished as pocket books in the 1960's[5]. As it can be noticed in the given example, the original publication order—for pulps—was not followed by the series of pocket books[6]. In addition, some stories were unpublished as pulps (e.g., *The Red Spider*) or retitled when published as pocket books (e.g., *The Deadly Dwarf*, previously entitled *Repel*), in which case, the pulp's title is given as the `pulp` element's contents[7]. More precisely, if a `pulp` element's contents is empty, this means that the pulp's title was the same as

```
<story-list>
  <story>
    <title>The Deadly Dwarf</title>
    <pulp nb="56">Repel</pulp>
    <pocket-book nb="28"/>
  </story>
  <story>
    <title>The Land of Terror</title>
    <pulp nb="2"/>
    <pocket-book nb="8"/>
  </story>
  <story>
    <title>The Lost Oasis</title>
    <pulp nb="7"/>
    <pocket-book nb="6"/>
  </story>
  <story>
    <title>The Man of Bronze</title>
    <pulp nb="1"/>
    <pocket-book nb="1"/>
  </story>
  <story>
    <title>The Red Spider</title>
    <pocket-book nb="95"/>
  </story>
  <story>
    <title>World&apos;s Fair Goblin</title>
    <pulp nb="74"/>
    <pocket-book nb="39"/>
  </story>
</story-list>
```

Figure 1. Master file using XML-like syntax.

the pocket book's. Fig. 2 gives the schema modelling our taxonomy[8], written using XML Schema [42]. Let us recall that this language provides a *datatype library*: for example, '`xsd:string`' for strings, the prefix '`xsl:`' allows us to get access to XML Schema's constructs[9].

Now we propose to search the information given in Fig. 1, extract the items published as pulps, sort them according to the publication order[10]. The title given is the pulp's; if the corresponding pocket book has been retitled, a footnote must give the 'new' title. Of course, we wish the result to be typeset nicely, as LaTeX or ConTeXt is able to do. To be more precise, a good solution processable by *Plain TeX* could look like the source text given in Fig. 3. As mentioned above, a TeX-based solution:

```
\story-list{%
 \story{\title{...}\pulp{...}...}%
 ...%
}
```

could use TeX commands for dealing with the elements `story-list`, `story`, `title`, `pulp`, and `pocket-book`, but would lead to complicated programming.

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="story-list">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="story"
                     maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title"
                           type="xsd:string"/>
              <xsd:element name="pulp"
                           minOccurs="0">
                <xsd:complexType>
                  <xsd:simpleContent>
                    <xsd:extension
                      base="xsd:string">
                      <xsd:attribute
                        ref="nb"
                        use="required"/>
                    </xsd:extension>
                  </xsd:simpleContent>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="pocket-book">
                <xsd:complexType>
                  <xsd:attribute
                    ref="nb"
                    use="required"/>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:attribute name="nb"
                 type="xsd:positiveInteger"/>
</xsd:schema>
```

Figure 2. Our organisation expressed in XML Schema.

Using tools related to XML

XSLT producing TeX sources

XSLT[11] [41] is the language designed for transformations of XML texts. By 'transformations', we mean that we can build printed documents as well as online documents to be put on the Web from the source file. 'Simple' texts are possible, too. We also can perform some computation from data stored in the original XML file. Let us come to our example, the text of a stylesheet that may be used to get Fig. 3 from Fig. 1 is given *in extenso* in Fig. 4. This stylesheet takes as much advantage as possible of the features introduced by XPath's and XSLT's new version[12] (2.0): for example, we have made precise the types of the used variables, by means of `as` attributes,

```
\item{1} The Man of Bronze
\item{2} The Land of Terror
\item{7} The Lost Oasis
\item{56} Repel\footnote*{Book's title of pulp \#56:
 The Deadly Dwarf}
\item{74} World's Fair Goblin

\end
```

Figure 3. Stories' titles sorted by pulp numbers.

these types being provided by XML Schema's library[13]. Likewise, we have made precise the types of templates' results. As it can be noticed in Fig. 4, these types belong to XML Schema's namespace, whereas XSLT constructs are prefixed by the namespace associated with '`xsl:`'.

Since we are interested in deriving texts processable by (LA)TEX, an important new feature introduced by XSLT 2.0 is the possible use of *character maps* [41, § 20.1]. In particular, they allow TEX's special characters to be replaced by commands producing them:

`#` ⟶ `\#` `\` ⟶ `$\backslash$`[14]

whenever they appear within a text node to be put by XSLT. More precisely, a single character can be replaced by a string, as shown by the character map `some-special-characters` given in Fig. 4. To distinguish an 'actual' backslash character, belonging to a string, and a command's beginning, a solution is to use a character belonging to a private area of Unicode [34] for the latter. For sake of readabiblity, we define this fictive character by means of an entity[15]—`start-command`—[32, p. 48–49]. Introducing this entity leads us to put a dummy `DOCTYPE` tag, since XSLT stylesheets do not refer to a DTD. In other words, specifying these additional characters is a 'trick', but that allows us to process strings extracted from the original XML file systematically. As shown in Fig. 4, the same technique can be used for opening and closing a group: we use fictive characters the character map transforms into braces. The same for a delimited fragment in (LA)TEX's math mode. Another solution could be the direct generation of Unicode texts and the use of a Unicode-compliant TEX-like engine[16], e.g. XETEX [26] or LuaTEX [9].

As abovementioned, XSLT is not limited to texts' generation, the `xsl:output` element's `method` attribute may also be set to `html` or `xhtml`[17] [41, § 20], in which case it allows Web pages to be generated. Likewise, this `method` attribute set to `xml` means that XML texts are to be generated. Using these output methods provides a great advantage: since any XSLT stylesheet is an XML text, an XSLT processor checks that the final document is well-formed, in particular, opening and closing tags must be balanced. The generation of (LA)TEX texts lacks an analogous check: an XSLT processor cannot ensure that opening and closing braces are balanced, as in TEX; likewise, when LaTEX texts are generated, it is impossible to ensure that environments like:

`\begin{document}` … `\end{document}` (1)

are balanced. Since such errors are not detected statically, they just appear when generated texts are processed. Let us notice that such a check would be more difficult to apply to the texts generated for the ConTEXt format, because the opening and closing commands for ConTEXt's environments are '`\start...`' and '`\stop...`', e.g., the equivalent formulation for (1) in LaTEX is:

`\starttext` … `\stoptext`

in ConTEXt. Concerning the delimiters of a command's arguments, we can ensure that opening and closing braces—more precisely, the two character entities `start-group` and `end-group`, before their replacement by braces—are balanced by using only the XSLT function `ntg:make-group` to build a TEX group from a sequence of strings. Let us remark that XSLT functions have been introduced in XSLT 2.0, so there is an additional reason to use this version. Another solution could be the use of an XML dialect whose architecture would reflect TEX's markup. From our point of view, that would complicate the process since an additional step—a translation from this dialect into 'actual' TEX-like syntax—would be performed. In addition, some versions of such dialects have already been proposed—e.g., in [7, § A.1] for LaTEX—but it seems to us that none is actually used.

XSLT producing XSL-FO texts

Since deriving XML texts by means of XSLT provides a better check level about syntax, an alternative idea is to get XSL-FO[18] [37] texts. Such texts use an XML-like syntax that aims to describe high-quality print outputs. As shown in [16], there is some similarity between LaTEX and XSL-FO, the latter providing, of course, more systematic markup[19]. This language is verbose, but it is not devoted to direct use: XSL-FO texts usually result from applying an XSLT stylesheet. The use of several namespaces—usually denoted by the prefixes '`xsl:`' and '`fo:`'—clearly distinguish elements belonging to XSLT and XSL-FO.

This approach's advantage is clear: generated texts are well-formed. However, XSL-FO lacks *document classes*, as in LaTEX. Some elements allow the description of *page models*, but end-users are entirely responsible for this definition. XSL-FO provides much expressive power about placement of *blocks*[20], but is very basic on other points. For example, let us consider footnotes, end-users are responsible of choosing each footnote

```
<!DOCTYPE stylesheet [<!ENTITY start-command "&#xE000;"> <!ENTITY start-group "&#xE001;">
                      <!ENTITY end-group "&#xE002;">]>

<xsl:stylesheet version="2.0" id="pulps-plus" xmlns:maps="http://www.ntg.nl"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                extension-element-prefixes="xsd">

  <xsl:output method="text" encoding="ISO-8859-1" use-character-maps="some-special-characters"/>

  <xsl:strip-space elements="*"/>

  <xsl:character-map name="some-special-characters">
    <xsl:output-character character="#" string="\#"/>
    <xsl:output-character character="%" string="\%"/>
    <xsl:output-character character="$" string="\$"/>
    <xsl:output-character character="&amp;" string="\&"/>       <!-- &. Plain TeX's commands -->
    <xsl:output-character character="\" string="$\backslash$"/> <!-- '\{' and '\}' are only -->
    <xsl:output-character character="{" string="$\{$"/>   <!-- usable in math mode -->
    <xsl:output-character character="}" string="$\}$"/>   <!-- (cf. [27, Exercise 16.12]). -->
    <xsl:output-character character="~" string="{\char'7E}"/> <!-- Using hexadecimal code. -->
    <xsl:output-character character="&start-command;" string="\"/>
    <xsl:output-character character="&start-group;" string="{"/>
    <xsl:output-character character="&end-group;" string="}"/>
  </xsl:character-map>

  <xsl:variable name="eol" select="'&#xA;'" as="xsd:string"/> <!-- (End-Of-Line character.) -->

  <xsl:template match="story-list" as="xsd:string">
    <xsl:variable name="pulps" as="xsd:string+">
      <xsl:apply-templates select="story[pulp]">
        <xsl:sort select="xsd:integer(pulp/@nb)"/>     <!-- Numerical sort. -->
      </xsl:apply-templates>
    </xsl:variable>
    <xsl:value-of select="$pulps,$eol,'&start-command;end',$eol" separator=""/>
  </xsl:template>

  <xsl:template match="story" as="xsd:string">
    <xsl:variable name="pulp-0" select="pulp" as="element(pulp)"/>
    <xsl:variable name="pulp-nb-0-string" select="xsd:string($pulp-0/@nb)" as="xsd:string"/>
    <xsl:variable name="pulp-title-0" select="data(pulp-0)" as="xsd:string"/>
    <xsl:variable name="title-processed" as="xsd:string">
      <xsl:apply-templates select="title"/>
    </xsl:variable>
    <xsl:value-of select='"&start-command;item",maps:mk-group($pulp-nb-0-string)," ",
                          if ($pulp-title-0) then
                            $pulp-title-0,"&start-command;footnote*",
                            maps:mk-group(("Book&apos;s title of pulp #",$pulp-nb-0-string,": ",
                                           $title-processed)) else
                            $title-processed,
                          $eol'
                  separator=""/>
  </xsl:template>

  <xsl:template match="title" as="xsd:string"><xsl:apply-templates/></xsl:template>

  <xsl:function name="maps:mk-group" as="xsd:string+">
    <xsl:param name="string-seq" as="xsd:string*"/>
    <xsl:sequence select="'&start-group;',$string-seq,'&end-group;'"/>
  </xsl:function>

</xsl:stylesheet>
```

Figure 4. Getting a source text for *Plain TEX* by means of an XSLT stylesheet.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>...</fo:layout-master-set>
  <fo:page-sequence master-reference="simple-page" font-family="serif" font-size="medium" text-align="left">
    <fo:flow flow-name="xsl-region-body">
      <fo:list-block provisional-distance-between-starts="20mm" provisional-label-separation="3mm">
        ...
        <fo:list-item>
          <fo:list-item-label start-indent="10mm" end-indent="label-end()">
            <fo:block>56</fo:block>
          </fo:list-item-label>
          <fo:list-item-body start-indent="body-start()">
            <fo:block>
              Repel<fo:footnote>
                <fo:inline font-size="x-small" vertical-align="super">*</fo:inline>
                <fo:footnote-body>
                  <fo:block text-align-last="justify"><fo:leader leader-pattern="rule"/></fo:block>
                  <fo:block font-size="xx-small">
                    <fo:inline font-size="xx-small" vertical-align="super">*</fo:inline>Book's title for
                    pulp #56:  The Deadly Dwarf
                  </fo:block>
                </fo:footnote-body>
              </fo:footnote>
            </fo:block>
          </fo:list-item-body>
        </fo:list-item>
        ...
      </fo:list-block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Figure 5. How to put a footnote in XSL-FO (see the equivalent *Plain TEX* source text in Fig. 3).

mark. Fig. 5 provides the result of applying an XSLT stylesheet providing an XSL-FO text for our example. The first child of the `fo:footnote` element gives the footnote reference, the second child is the actual footnote's contents [37, § 6.12.3]. This `fo:footnote` element seems to be low-level in comparison with LaTeX's `\footnote` command[21]. Of course, if you want footnotes to be numbered automatically, XSLT addresses this problem. However, another point seems to us to be more debatable: end-users are responsible for putting a leader separating footnotes from the text body[22] (we show how to proceed in Fig. 5). So some footnotes may be preceded by a leader, some may not. This point may seem anecdotal, but for LaTeX users, some features of XSL-FO can be viewed as low-level and be difficult to handle since they are already programmed in LaTeX classes.

Last but not least, most current XSL-FO processors do not implement the whole of this language, even if they can successfully process most of XSL-FO texts used in practice[23]. So some features may be unusable, whereas an equivalent construct will work in (LA)TeX. XSL-FO has been designed to deal with the whole of Unicode, so it shows how the Unicode bidirectional algorithm [35] is put into action [18], but this point may also be observed with XeTeX.

XQuery producing TEX sources

XQuery [40] is a query language for data stored in XML form, as SQL[24] does for relational data bases[25]. XQuery can be used to search documents and arrange the result, as an XML structure or a simple text (possibly suitable for a TeX-like engine). An XQuery program processing our example in order to get Fig. 3's text is given in Fig. 6. Like XSLT 2.0, XQuery uses XPath 2.0 expressions and the datatype library provided by XML Schema. As we did in XSLT, we systematically put type declarations using the `as` keyword, for sake of clarity and for taking as much advantage as possible of XQuery's type-checker. Such programs, using FLWOR[26] expressions, are more compact than equivalent ones in XSLT.

However, XQuery is suitable only for generating simple texts: advanced features like character maps in XSLT are provided by some XQuery processors, but are not portable. You have to use the `replace` function [39, § 7.6.3] to deal with TeX's special characters:

```
replace($s,"(#|%)","\\$1")
```

substitutes each occurrence of '#' (resp. '%') by '\#'

```
declare namespace maps = "http://www.ntg.nl" ;
declare namespace saxon = "http://saxon.sf.net/" ;
declare namespace xsd = "http://www.w3.org/2001/XMLSchema" ;

declare option saxon:output "omit-xml-declaration=yes" ;

declare variable $eol as xsd:string := "&#xA;" ;
declare variable $filename as xsd:string external ;

declare function maps:mk-group($string-seq as xsd:string*) as xsd:string+ {
  "{",$string-seq,"}"
} ;

if (doc-available($filename)) then
  string-join((for $story-0 as element(story) in doc($filename)/story-list/story[pulp]
               let $pulp-nb-0 as xsd:untypedAtomic := data($story-0/pulp/@nb),
                   $pulp-nb-0-int as xsd:integer := xsd:integer($pulp-nb-0),
                   $pulp-nb-0-string as xsd:string := xsd:string($pulp-nb-0),
                   $pulp-title-0 as xsd:string := xsd:string(data($story-0/pulp)),
                   $story-title-0 as xsd:string := xsd:string(data($story-0/title))
               order by $pulp-nb-0-int
               return ("\item",maps:mk-group($pulp-nb-0-string)," ",
                       if ($pulp-title-0) then
                         $pulp-title-0,"\footnote*",
                         maps:mk-group(("Book's title of pulp \#",$pulp-nb-0-string,": ",
                                        $story-title-0)) else
                         $story-title-0,$eol),
               $eol,"\end",$eol),
              "") else
  ()
```

Figure 6. Getting a source text for *Plain TEX* by means of XQuery.

(resp. '\%') within the string `$s`. Let us come back to implementation-dependent features, a simple example is given in Fig. 6: we declare that the result is not an XML text by a non-portable option, `saxon:output`[27]. Of course, if XQuery is used to generate XML texts, they are well-formed, but no analogous check can be done about texts generated for a TEX-like engine. In other words, XQuery has the same drawback as XSLT.

A curiosity: DSSSL

DSSSL[28] [21] was initially designed as the stylesheet language for displaying SGML[29] texts. DSSSL includes a core expression language that is a side-effect free subset of the Scheme programming language [25]. XML being a subset of SGML, stylesheets written using DSSSL can be applied to XML texts. DSSSL is rarely used now, but the example we cite illustrates how a functional programming language can be suitable for our requirements. Fig. 7 gives a stylesheet that produces a result equivalent to Fig. 3. In fact, end-users do not write (LA)TEX commands when they develop a stylesheet, the jade[30] program can generate TEX-like texts[31]:

```
jade -d pulps.dsl -t tex ds.sgml
```

—we have to specify a predefined *backend*, here '`tex`'— the typesetting engine usable to process `jade`'s results being JadeTEX [7, § 7.5.2]. Deriving texts directly processable by LaTEX or ConTEXt is impossible.

As shown in Fig. 7, processing a *`name`* element uses pattern-matching:

```
(element name E)
```

the *E* expression consists of assembling *literals* by means of the `make` form, using types predefined in DSSSL: `paragraph`, `sequence`, ... The generic type of such results is called *sosofo*[32] w.r.t. DSSSL's terminology.

Enriched TEX engines

If we go back to programs based on TEX-like typesetting engines, there are two other possible methods, based on TEX-engines 'enriched' by using a 'more classical' programming language. In both cases, XML texts are preprocessed by procedures belonging to a programming language, and the result is sent to a TEX-engine. Of course, such a *modus operandi* is suitable only if we want to generate (LA)TEX texts, it would be of little interest to get XML texts or pages written using (X)HTML[33].

PyTEX [6] is written using Python [28] and uses TEX as a daemon. Getting the components of a 'computed'

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">

<style-sheet>
  <style-specification id="pulps-plus">
    <style-specification-body>
      (declare-flow-object-class page-footnote
        "UNREGISTERED::Sebastian Rahtz//Flow Object Class::page-footnote")

      (root (let ((margin-size 1in))
              (make simple-page-sequence
                page-width: 210mm page-height: 297mm left-margin: margin-size right-margin: margin-size
                top-margin: margin-size bottom-margin: margin-size header-margin: margin-size
                footer-margin: 12mm center-footer: (page-number-sosofo)
                (process-children))))
      (element story-list-sgml
        (make sequence
          (let ((get-pulp (lambda (node-list) (select-elements (children node-list) "pulp"))))
            (process-node-list
             (apply node-list
                    (list-stable-sort
                     (node-list->list (node-list-filter (lambda (node-list)
                                                          (not (node-list-empty? (get-pulp node-list))))
                                                        (children (current-node))))
                     <
                     (lambda (story-node-list)
                       (string->number (attribute-string "nb" (get-pulp story-node-list))))))))))
      (element story
        (let* ((story-indent 20pt)
               (current-children (children (current-node)))
               (pulp-node-list (select-elements current-children "pulp"))
               (pulp-processed (process-node-list pulp-node-list))
               (pulp-title-string (data pulp-node-list))
               (title-processed (process-node-list (select-elements current-children "title"))))
          (make paragraph
            first-line-start-indent: (- story-indent) font-size: 12pt space-before: 10pt
            start-indent: story-indent pulp-processed space-literal
            (if (string-null? pulp-title-string)
                title-processed
                (let* ((footnote-marker-sosofo (literal "*"))
                       (footnotemark-sosofo (make-superscript footnote-marker-sosofo)))
                  (make sequence
                    (literal pulp-title-string) footnotemark-sosofo
                    (make page-footnote
                      footnotemark-sosofo (literal "Book's title of pulp #") pulp-processed (literal ": ")
                      title-processed)))))))
      (element title (process-children-trim))
      (element pulp (literal (attribute-string "nb")))
      (define space-literal (literal " "))
      (define make-superscript
        (let ((shift-factor 0.4)
              (size-factor 0.6))
          (lambda (sosofo-0)
            (make sequence
              font-size: (* (inherited-font-size) size-factor)
              position-point-shift: (* (inherited-font-size) shift-factor)
              sosofo-0))))

      (define (string-null?  string-0) ...)
      (define (list-stable-sort list-0 rel-2? key-f1)
        ;; Sorts list-0 according to the order relation rel-2?. The argument key-f1 gives a key for each element.
        ...)
    </style-specification-body>
  </style-specification>
</style-sheet>
```

Figure 7. DSSSL stylesheet generating a text to be printed.

text is left to the Python functions dealing with XML texts and successive results are sent to TEX, in turn.

LuaTEX [9] is able to call functions written using Lua [20]. On another point, this TEX-engine can process texts using XML-like syntax, as shown in [10]. Fig. 8 gives an implementation of our example in ConTEXt MkIV: it uses Lua to define an interface with sorting functions, the other functionalities being put into action using TEX-like commands. As in ConTEXt, the layout is controlled by set-up commands:

```
\start...setup  ...  \stop...setup
```

—for example, `title` elements' contents are just displayed, processing `pulp` elements displays the number or the title, depending on a mode—then our XML file is processed 'atomically', by means of the `\xmlprocessfile` command. This approach is promising, but let us recall that LuaTEX and ConTEXt MkIV have not yet reached stable state: that is planned for the year 2012. Another important drawback: as shown by the examples using the `\xmlfilter` command in Fig. 8, this command uses path expressions, very close to XPath expressions, but not identical. For example, `command`—used to connect a selected item to the set-up command that processes it—obviously does not belong to XPath. On the contrary, some XPath expressions are not recognised, even if 'simple' paths are processed. Some tricks may be used as workarounds, but we personally think that complete compatibility with XPath should be attained.

Conclusion

If we sum up the approaches shown throughout this article, those that seem suitable are XQuery for simple examples, XSLT for more ambitious ones, provided that Version 2.0 is used. The use of LuaTEX could be interesting in a near future, too.

However, we think that there are two directions that should be explored. The first would be a modern implementation of XSL-FO using a TEX-like typesetting engine. Such an implementation has begun: Passive TEX [4], but this project is presently stalled. We think that processing XSL-FO could re-use the experience accumulated by (LA)TEX developers, even if syntaxes are very different[34]. The second direction would be the definition and implementation of an output mode of XSLT suitable for (LA)TEX; some additional services could be performed: for example, checking that braces and environments are balanced. Such an output mode already exists in nbst[35], the language of bibliography styles close to XSLT and used by MlBibTEX[36] [11], but it concerns only the way to process LaTEX's special characters. If the `method` attribute of the `nbst:output` element is set to `text`, the result of:

```
\enablemode[ds:pulp]

\startxmlsetups xml:ds:base
 \xmlsetsetup{#1}{%
  story-list|story|title|pulp|pocket-book}{%
  xml:ds:*}
\stopxmlsetups

\xmlregisterdocumentsetup{ds}{xml:ds:base}

\startxmlsetups xml:ds:story-list
 \xmlresetsorter{story}
 \xmlfilter{#1}{%
  /story/command(xml:story-list:getkeys)}
 \subject{sortkeys}
 \xmlshowsorter{story}\blank
 \xmlsortentries{story}
 \xmlflushsorter{story}{xml:story-list:flush}
\stopxmlsetups

\startxmlsetups xml:story-list:getkeys
 \xmladdsortentry{story}{#1}{%
  \xmlattribute{#1}{/pulp}{nb}}
\stopxmlsetups

\startxmlsetups xml:story-list:flush
 \startitemize\xmlfirst{#1}{.}\stopitemize
\stopxmlsetups

\startxmlsetups xml:ds:story
 \sym{\xmlfirst{#1}{pulp}}
 \xmldoifelsetext{#1}{pulp}{
  {\disablemode[ds:pulp] \xmlfirst{#1}{pulp}
   \footnote{%
    Book's title: \xmlfirst{#1}{title}}}{%
  \xmlfirst{#1}{title}}
\stopxmlsetups

\startxmlsetups xml:ds:title
 \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:ds:pulp
 \doifmodeelse{ds:pulp}{\xmlatt{#1}{nb}}{%
  \xmlflush{#1}}
\stopxmlsetups

\starttext
 \xmlprocessfile{ds}{ds.xml}{}
\stoptext
```

Figure 8. Processing a master file with ConTEXt Mk IV.

```
<nbst:text>#60 The Maji</nbst:text>
```

is '`#60 The Maji`'. If this attribute is set to `LaTeX`, the result is '`\#60 The Maji`'.

Acknowledgements

I wish to thank Hans Hagen who greatly helped me debug and improve LuaTEX source texts. Thanks also to Karl Berry, who clarified some terminology notions.

Last but not least, thanks to Taco Hoekwater for his patience when he was waiting for this final version.

References

[1] *Apache FOP*. November 2010. `http://xmlgraphics.apache.org/fop/`.

[2] Frédéric Boulanger : « LaTeX au pays des tableurs ». *Cahiers GUTenberg*, Vol. 39–40, p. 7–16. In *Actes du Congrès GUTenberg 2001*, Metz. Mai 2001.

[3] Neil Bradley: *The Concise SGML Companion*. Addison-Wesley. 1997.

[4] David Carlisle, Michel Goossens et Sebastian Rahtz : « De XML à PDF avec xmltex, XSLT et PassiveTeX ». *Cahiers GUTenberg*, Vol. 35–36, p. 79–114. In *Actes du congrès GUTenberg 2000*, Toulouse. Mai 2000.

[5] James Clark *et al.*: *Relax NG*. `http://www.oasis-open.org/committees/relax-ng/`. 2002.

[6] Jonathan Fine: "TeX Forever!". In: *Proc. EuroTeX 2005*, pp. 150–158. Pont-à Mousson, France. March 2005.

[7] Michel Goossens and Sebastian Rahtz, with Eitan M. Gurari, Ross Moore and Robert S. Sutor: *The LaTeX Web Companion*. Addison-Wesley Longman, Inc., Reading, Massachusetts. May 1999.

[8] Hans Hagen: *ConTeXt, the Manual*. November 2001. `http://www.pragma-ade.com/general/manuals/cont-enp.pdf`.

[9] Hans Hagen: "The Luafication of TeX and ConTeXt". In: *Proc. BachoTeX 2008 Conference*, pp. 114–123. April 2008.

[10] Hans Hagen: "Dealing with XML in ConTeXt MkIV". *MAPS*, Vol. 37, pp. 25–39. 2008.

[11] Jean-Michel Hufflen: "MlBibTeX's Version 1.3". *TUGboat*, Vol. 24, no. 2, pp. 249–262. July 2003.

[12] Jean-Michel Hufflen: "Introduction to XSLT". *Biuletyn GUST*, Vol. 22, pp. 64. In *BachoTeX 2005 conference*. April 2005.

[13] Jean-Michel Hufflen: "Advanced Techniques in XSLT". *Biuletyn GUST*, Vol. 23, pp. 69–75. In *BachoTeX 2006 conference*. April 2006.

[14] Jean-Michel Hufflen: "Introducing LaTeX users to XSL-FO". *TUGboat*, Vol. 29, no. 1, pp. 118–124. EuroBachoTeX 2007 proceedings. 2007.

[15] Jean-Michel Hufflen: "XSLT 2.0 vs XSLT 1.0". In: *Proc. BachoTeX 2008 Conference*, pp. 67–77. April 2008.

[16] Jean-Michel Hufflen : « Passer de LaTeX à XSL-FO ». *Cahiers GUTenberg*, Vol. 51, p. 77–99. Octobre 2008.

[17] Jean-Michel Hufflen: "Introduction to XQuery". In: Tomasz Przechlewski, Karl Berry and Jerzy B. Ludwichowski, eds., *TeX: at a Turning Point, or at the Crossroads? Proc. BachoTeX 2009 Conference*, pp. 17–25. April 2009.

[18] Jean-Michel Hufflen: "Multidirectional Typesetting in XSL-FO". In: Tomasz Przechlewski, Karl Berry and Jerzy B. Ludwichowski, eds., *TeX: at a Turning Point, or at the Crossroads? Proc. BachoTeX 2009 Conference*, pp. 37–40. April 2009.

[19] Jean-Michel Hufflen: "Processing 'Computed' Texts". *ArsTeXnica*, Vol. 8, pp. 102–109. In GUIT 2009 meeting. October 2009.

[20] Roberto Ierusalimschy: *Programming in Lua*. 2nd edition. Lua.org. March 2006.

[21] International Standard ISO/IEC 10179:1996(E): *DSSSL*. 1996.

[22] ISO/IEC 19757: *The Schematron. An XML Structure Validation Language Using Patterns in Trees*. `http://www.ascc.net/xml/resource/schematron/schematron.html`. June 2003.

[23] Michael H. Kay: *XSLT 2.0 Programmer's Reference*. 3rd edition. Wiley Publishing, Inc. 2004.

[24] Michael H. Kay: *Saxon. The XSLT and XQuery Processor*. October 2010. `http://saxon.sourceforge.net`.

[25] Richard Kelsey, William D. Clinger, and Jonathan A. Rees, with Harold Abelson, Norman I. Adams iv, David H. Bartley, Gary Brooks, R. Kent Dybvig, Daniel P. Friedman, Robert Halstead, Chris Hanson, Christopher T. Haynes, Eugene Edmund Kohlbecker, Jr, Donald Oxley, Kent M. Pitman, Guillermo J. Rozas, Guy Lewis Steele, Jr, Gerald Jay Sussman and Mitchell Wand: "Revised^5 Report on the Algorithmic Language Scheme". *HOSC*, Vol. 11, no. 1, pp. 7–105. August 1998.

[26] Jonathan Kew: "XeTeX in TeX Live and beyond". *TUGboat*, Vol. 29, no. 1, pp. 146–150. EuroBachoTeX 2007 proceedings. 2007.

[27] Donald Ervin Knuth: *Computers & Typesetting. Vol. A: The TeXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1984.

[28] Alex Martelli: *Python in a Nutshell*. 2nd edition. O'Reilly. July 2006.

[29] Jim Melton and Alan R. Simon: *Understanding the new SQL*. Morgan Kaufmann. 1993.

[30] Frank Mittelbach and Michel Goossens, with Johannes Braams, David Carlisle, Chris A. Rowley, Christine Detig and Joachim Schrod: *The LaTeX Companion*. 2nd edition. Addison-Wesley Publishing Company, Reading, Mas-

sachusetts. August 2004.
[31] Chuck Musciano and Bill Kennedy: *HTML & XHTML: The Definitive Guide*. 5th edition. O'Reilly & Associates, Inc. August 2002.
[32] Erik T. Ray: *Learning XML*. O'Reilly & Associates, Inc. January 2001.
[33] Denis B. Roegel : « Anatomie d'une macro ». *Cahiers GUTenberg*, Vol. 31, p. 19–27. Décembre 1998.
[34] The Unicode Consortium: *The Unicode Standard Version 5.0*. Addison-Wesley. November 2006.
[35] The Unicode Consortium, http://unicode.org/reports/tr9/: *Unicode Bidirectional Algorithm*. Unicode Standard Annex #9. March 2008.
[36] Eric van der Vlist: *Comparing XML Schema Languages*. http://www.xml.com/pub/a/2001/12/12/schemacompare.html. December 2001.
[37] W3C: *Extensible Stylesheet Language (XSL). Version 1.1*. W3C Recommendation. Edited by Anders Berglund. December 2006. http://www.w3.org/TR/2006/REC-xsl11-20061205/.
[38] W3C: *XML Path Language (XPath) 2.0*. W3C Recommendation Draft. Edited by Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael H. Kay, Jonathan Robie and Jérôme Siméon. January 2007. http://www.w3.org/TR/2007/WD-xpath20-20070123.
[39] W3C: *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Recommendation. Edited by Ashok Malhotra, Jim Melton, and Norman Walsh. January 2007. http://www.w3.org/TR/2007/REC-xpath-functions-20070123.
[40] W3C: *XQuery 1.0: an XML Query Language*. W3C Recommendation. Edited by Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie and Jérôme Siméon. January 2007. http://www.w3.org/TR/xquery.
[41] W3C: *XSL Transformations (XSLT). Version 2.0*. W3C Recommendation. Edited by Michael H. Kay. January 2007. http://www.w3.org/TR/2007/WD-xslt20-20070123.
[42] W3C: *XML Schema*. December 2008. http://www.w3.org/XML/Schema.
[43] Larry Widen and Chris Miracle: *Doc Savage: Arch Enemy of Evil*. Fantasticon Press, Milwaukee, Wisconsin. 1993.

Notes

1. Let us mention that [2] shows—in French—how to use LaTeX to put spreadsheets' functionalities into action.
2. As an example of using TeX's language for programming purposes, readers interested in putting a sort procedure into action can refer to [33]: this *modus operandi* may be viewed as a worthwhile exercise, but is unusable in practice, especially when it is not trivial to obtain sort keys from items to be sorted.
3. e**X**tensible **M**arkup **L**anguage. Readers interested in a general introductory book to this formalism can refer to [32].
4. The XML format used by Microsoft Excel is OOXML (**O**ffice **O**pen XML).
5. All the source texts mentioned throughout [19] and this article—including new versions realised for this present article, in which case the corresponding file names are suffixed with '`-plus`'—can be downloaded *in extenso* from the Web page http://lifc.univ-fcomte.fr/home/~jmhufflen/texts/guit-2009/.
6. If you are interested in the story of *Doc Savage* series and its successive editions, you can find more information in [43].
7. When several titles have been used, such a story is more commonly known under the pocket book's title, because pocket books are easier to get than pulps, which are very rare. That is why our `title` elements always refer to pocket books', the contents of `pocket-book` elements being always empty.
8. *Schemas* allow specifiers to define *document types*, which can be viewed as some taxonomy common to a family of XML texts. There exist several schema languages, and the Web page abovementioned gives several versions, using a DTD (**D**ocument **T**ype **D**efinition) [32, Ch. 5], XML Schema [42], Relax NG (**N**ew **G**eneration) [5], and Schematron [22]. A discussed comparison among these schema languages can be found in [36].
9. Readers interested in more details about XML namespaces can consult [32, pp. 41–45].
10. This example seems to us to be pertinent, because there is no order 'better' than others: the original order is based on pulps, but—as mentioned above—some stories are unpublished as pulps, whereas sorting stories according to pocket books' order allows us to sort all the stories, but this is not really chronological.
11. e**X**tensible **S**tylesheet **L**anguage **T**ransformations. Introductions to this language have been given in some BachoTeX conferences, held in Poland: [12, 13, 15].
12. A study of XPath 2.0's and XSLT 2.0's new features, in comparison with XPath 1.0 and XSLT 1.0, can be found in [15].
13. In addition, let us mention that when an XML text is processed by XSLT 2.0, the information put into a DTD or an XML Schema text can be exploited [23, p. 58]. That is not true about other schema languages.
14. In text mode, LaTeX 2_ε's modern versions provide the `\textbackslash` command [30, Table 7.33].
15. If we name this character by introducing a variable by means of an `xsl:variable` element—as we did in Fig. 4 for the end-of-line character—we cannot use this variable's value within the `character` attribute of the `xsl:output-character` element.
16. Engines are not formats: a *format* is a set of pre-loaded definitions based on primitives of a TeX-like engine, whereas an *engine* is TeX or is derived from TeX by adding or redefining

some primitives. *Plain TEX* and LaTEX are formats, XeTEX and LuaTEX are engines.
17. (e**X**tensible) **H**yper**T**ext **M**arkup **L**anguage. XHTML is a reformulation of HTML—the original language of *Web* pages—using XML conventions. [31] is a good introduction to these languages.
18. e**X**tensible **S**tylesheet **L**anguage–**F**ormatting **O**bjects.
19. [16] is written in French. If you would like a similar text in English, [14] is an abridged version.
20. Roughly speaking, a *block* in XSL-FO is analogous to a *minipage* in LaTEX [16, § 1.2].
21. XSL-FO's footnotes can be compared with *Plain TEX*'s `\footnote` command, as shown in Fig. 3.
22. That is done in standard LaTEX class, but not universal: as an example related to TEX's community, the arstexnica class, used for articles of the ArsTEXnica journal—published by GUIT (*Gruppo Utilizzatori Italiani di TEX*), the Italian-speaking TEX users group—does not put leaders just above footnotes.
23. We personally use Apache FOP (**F**ormatting **O**bjects **P**rocessor) [1]:

```
fop pulps-result.fo pulps-result.pdf
```

generates a PDF (*P*ortable **D**ocument **F**ormat) file from a source text in XSL-FO.
24. **S**tructured **Q**uery **L**anguage. A good introductory book about it is [29].
25. A short introduction to XQuery is given in [17].
26. '**F**or, **L**et, **W**here, **O**rder by, **R**eturn', the keywords used throughout such expressions.
27. The XQuery processor we have used for this example is Saxon [24]. Fig. 6's text can be processed by:

```
java net.sf.saxon.Query pulps-plus.xq \
  filename="ds.xml"
```

We also use Saxon as an XSLT 2.0 processor and the stylesheet of Fig. 4 can be processed by:

```
java net.sf.saxon.Transform -s:ds.xml \
  -xsl:pulps-plus.xsl
```

28. **D**ocument **S**tyle **S**emantics **S**pecification **L**anguage.
29. **S**tandard **G**eneralised **M**arkup **L**anguage. Now it is only of a historical interest. Readers interested in this metalanguage can refer to [3].
30. **J**ames Clark's **A**wesome **D**SSSL **E**ngine.
31. We use a different file and a different name for the root element (`story-list-sgml`) because of syntactic reasons: empty tags' syntax was different in SGML [3, p. 259].
32. **S**pecification **O**f a **S**equence **O**f **F**low **O**bjects.
33. Unless a converter to (X)HTML is used, of course.
34. Besides, it is well-known that TEX recognises only its own formats, which complicates cooperation between TEX and other programs.
35. **N**ew **B**ibliography **STy**les.
36. **M**ulti**L**ingual BibTEX.

Jean-Michel Hufflen
LIFC (EA CNRS 4157),
University of Franche-Comté, 16, route de Gray,
25030 Besançon Cedex, France

à la Mondrian

Abstract
Mondriaan has worked most of his live as an abstract painter, influenced by the magic realism of Jan Toorop, and by Cubism and Pointillism. He was member of De Stijl and has lived in Paris and in New York. Some of his work seems to have been composed randomly, though he was very precise, as witnessed by the overpainting of various squares in his Victory Boogie-Woogie. Mondriaan's `random' work `Composition in Line 1916`, is emulated and varied in MetaPost and PostScript, in color, with the lines (position and size) randomly chosen. He was the first painter to frame work by Lozenges. Division of the sides of his `Lozenge with 2 lines` is near to the golden ratio. Emulated Lozenges obeying the golden ratio have been included. The variations look nevertheless Mondriaan-esque.

Keywords
Art, color, cubism, De Stijl, golden ratio, Mondrian, pointillism, pseudo-random numbers, MetaPost, PostScript, Toorop

Introduction

My applied math professor Hans Lauwerier[1] published in 1987, long after I finished my math education, *Fractals—meetkundige figuren in eindeloze herhaling*. It contained the following picture, which he called 'à la Mondriaan'[2]

Lauwerier used this picture to illustrate a very simple algorithm[3] for generating a pseudo-random number sequence in BASIC

> "... start with a four digit number, the seed, square it and delete the first and last two numbers, and repeat the process of squaring and deleting ..."

He randomly positioned (place, horizontally or vertically) the line pieces of random length. I'll improve on his picture with the use of a shade of color, and variation in line thickness. Moreover, I'll cadre by different frames.

Mondriaan

Mondriaan was born in 1872 at Amersfoort and started drawing à la nature.

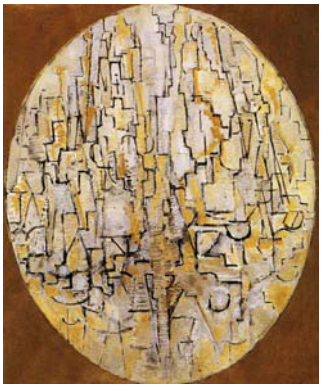
Meisje 1890 Winterswijk 1895

Jan Toorop has influenced Mondriaan by his new realism. Mondriaan was often a guest at Toorop's in Zeeland in the beginning of the 20^{th} century; the (light at the) seashore inspired Mondriaan.

Zeeland farmer 1909 Red Cloud 1907

Arum Lilies 1909 Lighthouse WC 1908/9

Mondriaan exercised an oval as a boundary, not as a frame. The `Composition in Oval 1913` reminds me of the work of George Braque.



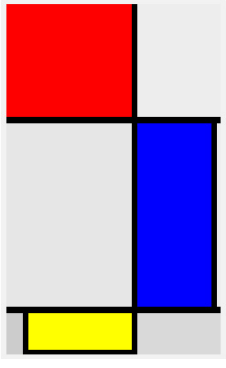
`Composition in Oval 1913`

`Composition in Oval 1914`

He has lived in Paris, 1919-1938, and was influenced by cubism and pointillism.

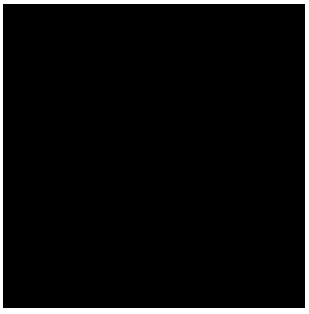
`Blossoming Apple Tree 1912` `Beach with one Pier 1909`

I was familiar with some of Mondriaan's works from the 'De Stijl'[4] period, where he exercised the use of primary color panes and straight lines, which reminds me of stained glass windows.

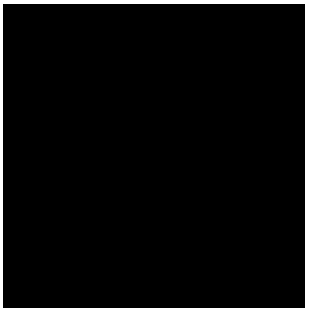


emulation: `Large Composition with Red, Blue and Yellow 1928`

His series of Lozenge's from the twenties, ended in a minimal one, given below at the right.

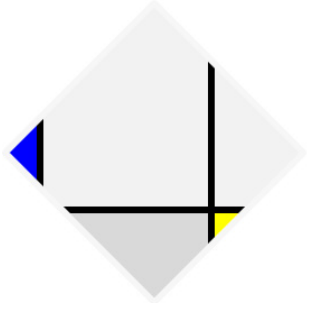


Lozenge with 3 lines 1925

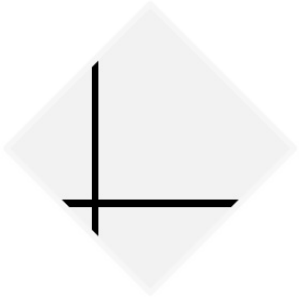


Lozenge with 2 lines 1931

I measured his Lozenges and was suprised that he missed the divison of the sides by the golden ratio: his sides are divided in $3.9 : 5.6$; the golden ratio Φ does not hold, ie $3.9 : 5.6 \neq 5.6 : 9.5$. A tiny difference, but nonetheless.



Φ emulation



Φ emulation

```
%Mondriaan-like Lozenge with 3 lines. Fall 2009 kisa1@xs4all.nl
beginfig(0)
s=200;
z1=(0,.5s); z2=(.5s,s); z3=(s,.5s); z4=(.5s,0);
z5=0.618/1.618[z2,z3]; z6=0.618/1.618[z3,z4];
z7= 1/1.618[z3,z4]; z8=1/1.618[z4,z1];
z9=0.382/1.618[z1,z4]; z10=0.382/1.618[z1,z2];
z11= (z5--z7) intersectionpoint (z6--z8);
path p; p = z1--z2--z3--z4--cycle;
pw=4; pickup pencircle scaled pw;
fill p withcolor .95white;
fill z1--z10--z9--cycle withcolor blue;
fill z7--z11--z6--cycle withcolor red+green;
fill z8--z11--z7--z4--cycle withcolor .85white;
draw z5--z7;
draw z6--z8;
draw z9--z10;
clip currentpicture  to p;
draw p withcolor .97white;
endfig
end
```

For the right Lozenge the main part of the above reads

```
fill p withcolor .95white;
draw z5--z7;
draw z6--z8;
clip currentpicture  to p;
draw p withcolor .97white;
```

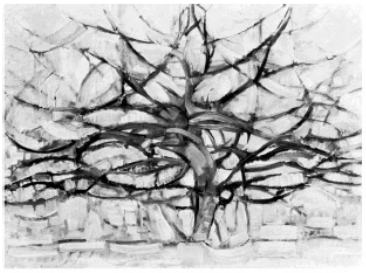
Remark. In contrast with PostScript we don't have to translate the origin in MetaPost to somewhere in the middle.
For my last year's birthday Sveta and I composed the following à la Mondriaan invitation starting from the left Φ-Lozenge.

Not so long ago, I used one of his Apple trees, along with a little poetic proza, for our local gardeners bulletin. It is about one of our Apple trees, which in every season has something to offer:

- spring, blossoming beauty
- summer, healthy shade
- autumn, fruits to enjoy
- winter, beautiful silhouet à la Mondriaan.

Appelweelde

Bruingetinte
het is november,
zeldzaam zacht.
De appelboom
majestueus,
met zijn takken
armen vol met
Wat een weelde,

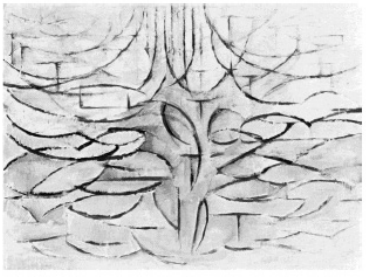


bladeren op het gras
herfst, en

staat fier en
stevig geworteld,
als uitgespreide
appels.
wat een pracht!

Van de vroege, aangetaste appels heeft
deze reus zich al in juli en augustus ontdaan.
Nu hangen er nog een en al gave ponders,
tjokvol met sap.

Het is een
appelboom op en
tuin, een
jaar door:
bloesempracht in
schaduw in de
oogst in de
een Mondriaan
silhouet in de

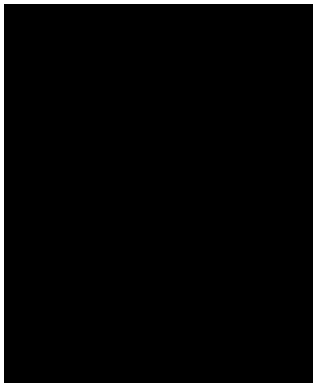


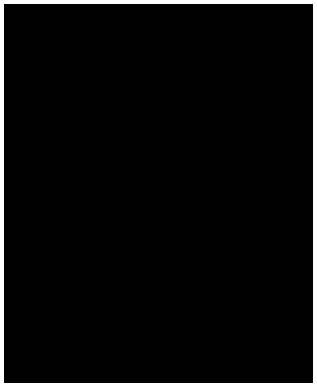
rijkdom zo'n
ruime plek in de
genoegen het hele

de lente,
zomer,
herfst, en

winter.

His unfinished Victory Boogie-Woogie is shown on the cover of the biography about Mondriaan.





Mondriaan has the reputation to be very precise, and I was curious whether some of his works

> "... could be generated randomly in color and nevertheless convey a Mondriaan impression ... "

although at the time he did not use color for the 'random' pictures given on the next page.

Composition in Line, 1916

Composition 10, 1915

Variation of Mondriaan's 'random' art

Is it random?

As far as I understand it, his work was composed precisely, very precisely, as witnessed by his unfinished Victory Boogie-Woogie, where several squares have been painted over.

Square, Oval and Lozenge cadres. I generated an abstract picture of lines, with a spread in size, thickness and with a color shade, such that the impression is blue-like, green-like,...

I intended these pictures as presents to my friends. Made them unique by using the date of birth (month and day) as seed for the pseudo-random number generator in MetaPost.[5] Colored the frame with their favorite color, which also biases the color shade.

Below is Sveta's one in pink, with 1007 as seed for the pseudo-random number generator.

How to?

First, we have to decide on the size of the rectangle. Next the number of randoms. I chose 500, but that can be adjusted at will. The lines are drawn within a rectangle of 180 by $180 \cdot 1.618$, obeying the golden ratio. For the position in the rectangle the

random number generator must generate numbers between 0 0 180 ≈291, conform PostScript's BoundingBox convention to denote a rectangle.

`uniformdeviate` delivers a number between 0 and its argument. We have the intervals (0, u) and (0,v), so the invokes for the (x, y)-position read

```
hx:= uniformdeviate u;  hy:= uniformdeviate v;
```

I alternate between vertical and horizontal lines, where the lines have maximum length `size=10`, and maximum (pen)width `pw=5pt`. The actual size, horizontally as well as vertically, is determined by an invoke of `uniformdeviate`

```
draw (hx, hy)--(hx + uniformdeviate size, hy) withcolor rgb%vertical line
```

Line thickness is varied as follows:

```
pickup pencircle scaled ((uniformdeviate 1)*pw);
```

The most difficult part is the color, which I let vary a little around the chosen value, in order to generate a shade but ... gives the main color impression. I chose a multiplicative factor `uniformdeviate 2` and called this `spread`.

To finish it up, I clip and border first to the rectangle, next to the oval, and finally to the diamond; all border increasingly the same pattern.

```
%Mondriaan-alike. Fall 2009 CGL
if scantokens(mpversion) > 1.005:  outputtemplate :=
else:                               filenametemplate
fi                                         "%j.eps";
prologues:=3;
beginfig(0)
u:=180;    %rectangle u x v, golden ratio proportion
v:=1.618u;
size=10; pw=5;
n=500;     %number of randoms
path p.r,   %rectangle
       p.o,  %oval
       p.d;  %diamond
picture  pic.r, pic.o, pic.d, signature;
defaultfont:="ptmr8r";
label.rt("cgl okt09", (2.75u+4pw, 2pw));%seed may be shown instead
signature:=currentpicture; currentpicture:=nullpicture;
%
color rgb, colorofchoice;
%parameters for Sveta
randomseed:=1007; colorofchoice:= .8*red +.2*green .7*blue;%roze
%
for j=0 upto n:
 pickup pencircle scaled ((uniformdeviate 1)*pw*pt); linecap:=squared;
 hx:= uniformdeviate u; hy:= uniformdeviate v;
 rgb:= .8*(uniformdeviate 2)*red + .2*(uniformdeviate 2)*green
       + .7*(uniformdeviate 2)*blue;
 draw (hx,hy)--(hx,hy+ uniformdeviate size) withcolor rgb;
 hx:= uniformdeviate u; hy:= uniformdeviate v;
 rgb:= .8*(uniformdeviate 2)*red + .2*(uniformdeviate 2)*green
       + .7*(uniformdeviate 2)*blue;
```

```
draw (hx,hy)--(hx+uniformdeviate size, hy) withcolor rgb;
endfor;
p.r= ((0, v)--(u,v)--(u,0)--(0,0)--cycle) shifted (2pw,2pw);
p.o= ((.5u,v){right}...(u,.5v){down}...(.5u,0){left}...(0,.5v){up}
       ...cycle) shifted (u+4pw,2pw);
p.d= ((.5u,v)--(u,.5v)--(.5u,0)--(0,.5v)--cycle) shifted (2u+6pw,2pw);
currentpicture:=currentpicture shifted (2pw,pw);
clip currentpicture to p.r;
pic.o:=currentpicture shifted (u+4pw,0);
clip pic.o to p.o;%clip more
pic.d:= currentpicture shifted (2u+6pw,0);
clip pic.d to p.d;%clip even more
addto currentpicture also pic.o;%add shifted pictures
addto currentpicture also pic.d;
%
pickup pencircle scaled pw; linecap:=rounded;
drawoptions (withcolor .8*red +.2*green .7*blue);
draw p.r; draw p.o; draw p.d;
addto currentpicture also signature;
endfig end
```

I reused this program for generating personalized presents several times, but I am happier with the PostScript operator given below. When the picture was for a man, I copied the Square or the Lozenge, and when it is for a woman I copied the Oval.[6] Frame the picture in a physical frame and ready it is, apart from a wrap-around paper.

Note that all three frames have the same pattern: the pattern is clipped by frame variations. Subtle is the printing of the signature outside the clipping area.

PostScript operator variant

Why a PostScript variant of the MetaPost program?

- allows larger numbers (seed eg 22121943)
- more convenient clipping
- operator (library)
- no invoke of MP
- easier experimenting with color shade
- improved code.

In this variant I generate only one framed picture triggered by the value 0, 1, or 2 for the last parameter on the stack, where 0=Square 1=Oval 2=Lozenge. The size is 420 by $420 \cdot 1.618$ and positioned with the lower left corner at

```
100 50 translate
```

for a centered result. The complete birthday date, ddmmyyyy, can be supplied. The code is not a direct translation, because of the language differences and because I matured in coding and was after a library operator. The thickness of the strokes are obtained as follows

```
/w  {maxwidth unifrmdev} def
... w setlinewidth
```

The line pieces are positioned symmetrically

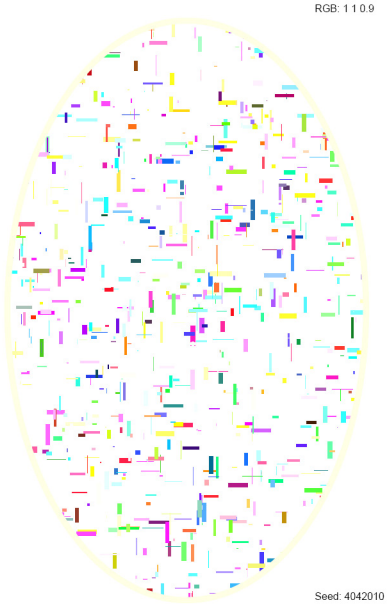
```
/laux l 2 div def
xaux laux sub yaux moveto xaux laux add yaux lineto w setlinewidth
color setrgbcolor stroke%h-line
```

The spread in color has been implemented as: if an rgb-value = 0, it is changed into eps=0.1 and the resulting value c.q. the original non-zero value is multiplied by the spread factor.

```
/spread {2 unifrmdev  mul }def%interval (0, 2)
/color{r 0 eq {eps}{r} ifelse spread
          g 0 eq {eps}{g} ifelse spread
          b 0 eq {eps}{b} ifelse spread} def
```

The strokes are in a shade of the desired color. The rgb-values are printed at the top, and the birthday date, the seed for the pseudo-random number generator, at the bottom.

The most Mondrian-like is maybe the Easter Egg.




```
/Mondrian
%birthday: ddmmyyyy, a number as seed for srand
%three numbers from the closed interval [0, 1], for the rgb-color values:
% red green blue
%number for the kind of frame (0=Square 1=Oval 2=Lozenge}: 0, 1 or 2;
%==>
%generated Mondrian-alike
{0 begin gsave %savety for not changing the graphics state outside
/form exch def
/b exch def /g exch def /r exch def /date exch def
date srand% start random generator with (birthday date) seed
100 50 translate
%wired-in parameters
/u 420 def /v u 1.618 mul def /hu u 2 div def
/hv v 2 div def% 0 0 u v BB of rectangle
```

```
/maxrandom 500 def /maxlength 20 def /maxwidth 3 def /eps 0.1 def
/hx  {u unifrmdev}            def
/hy  {v unifrmdev}           def
/l    {maxlength unifrmdev}def
/w  {maxwidth unifrmdev} def
/spread {2 unifrmdev mul }def                 %(0, 2)
%/spread {2 unifrmdev 1 add 2 div mul }def%(0.5, 1.5)
%/spread {2 unifrmdev 3 add 4 div mul }def%(0.75, 1.25)
/color{r 0 eq {eps}{r} ifelse spread
          g 0 eq {eps}{g} ifelse spread
          b 0 eq {eps}{b} ifelse spread} def
form 0 eq {/contour
  {0 0 moveto u 0 lineto u v lineto 0 v lineto closepath} def} if %Square
form 1 eq {/contour
  {hu hv hu hv 0 360 ellipse} def} if                              %Oval
form 2 eq {/contour
  {hu 0 moveto u hv lineto hu v lineto 0 hv lineto closepath}
  def} if                                                       %Lozenge
%
gsave contour clip%random pattern will only show up in (is clipped to)
   contour maxrandom{%draw pattern in loop confined to contour
/xaux hx def /yaux hy def%position in u x v rectangle
/laux l 2 div def
xaux laux sub yaux moveto xaux laux add yaux lineto w setlinewidth
  color setrgbcolor stroke%h-line
/xaux hx def /yaux hy def/laux l 2 div def
xaux yaux laux sub moveto xaux yaux laux add lineto w setlinewidth
  color setrgbcolor stroke%v-line
}repeat
grestore %end clipping path
contour 7 setlinewidth r g b setrgbcolor stroke%original color od choice
H12pt setfont /nstr 8 string def
u 85 sub v 10 add moveto (RGB: ) show
                    r nstr cvs show ( ) show
                    g nstr cvs show ( ) show
                    b nstr cvs show
u 85 sub 0 moveto (Seed: ) show  date nstr cvs show
grestore end}def%end Mondrian
/Mondrian load 0 26 dict put
```

A variant suited for cmyk-color values, also some 50 lines, took me about 10 minutesje . When $k = 0$ I did not let it contribute to the spread. A brainfag is that cyan absorbs red, etc, an approach different from the rgb-model.

Example of use.

```
%!!PS Mondriaan-achtig. CGL April 2010
%BoundingBox: 0 0 620 790
(C:\\PSlib\\PSlib.eps) run
22121943   1  0  0 0 Mondrian showpage
22121943   0  1  0 1 Mondrian showpage
22121943   0  0  1 2 Mondrian showpage
22121943  .5 .5 .5 2 Mondrian showpage
%%EOF
```

No invoke of the MetaPost preprocessor is needed, just the use of distiller (as part of Adobe's Acrobat) or the `ps2pdf` batch program, or ... will yield the visual result.

Acknowledgements

Thank you Hans Lauwerier for your inspiring material, and thank you Piet Mondriaan for your great art.

For a discussion of the differences in color on various devices, I refer to the LaTeX Graphics Companion or to Siep Kroonenberg's *Color in professional print production*, MAPS 20, spring 1998.

Thank you Jos Winnink for your suggestion to reorder the material.

Conclusion

Mondriaan was undoubtedly precise, but some of his 'random'-like works can be created by the use of a pseudo-random number generator and yield nevertheless a Mondriaan impression.

The positioning of the lines in his `Lozenge with 2 lines` and `Lozenge with 3 lines` is also precise, but ... division of the sides by the golden ratio Φ yields equally-well artistic results.

I have input his Victory Boogie-Woogie in MetaPost, tedious work, and also scanned the picture. Don't have any use for it as yet.

My case rests, have fun and all the best.

Notes

1. For a survey of biographies of Dutch mathematicians http://bwnw.cwi-incubator.nl/cgi-bin/uncgi/alf.
2. Born as Piet Mondriaan. At the end of his live, in the USA, he used to call himself Mondrian.
3. Which obeys statistical tests for randomness.
4. A Dutch art movement which began in 1917. Its characteristics are the use of straight lines and primary colors. The artists were generally after utmost simplicity and abstraction. Main representatives were Theo van Doesburg, Piet Mondriaan, Bart van der Leck, Gerrit Rietveld, and J.J.P. Oud, active in various art forms, among others architecture.
5. Adding the year exceeds MetaPost number capacity for the moment.
6. This extra work is superfluous in the PostScript variant.

Kees van der Laan
March 2010
Hunzeweg 57, 9893PB, Garnwerd, NL
kisa1@xs4all.nl

NTG Najaarsbijeenkomst 2010

Tijdens een van haar presentaties op de ConTEXt meeting in Brejlov dit jaar vertelde de Finse Mari Voipio dat ze altijd wat waardevols oppikt tijdens TEX-conferenties. “Veel van wat er wordt verteld gaat langs me heen of het vliegt hoog boven mijn hoofd voorbij, maar er is ook altijd iets wat mijn ogen opent, een ontdekking waarmee ik rijker naar huis terugkeer.”

Ik verheug me ook altijd op een ntg-dag. TEX gebruikers onder elkaar, die geïnteresseerd zijn in zoveel dat direct of zijdelings te maken heeft met het typesetten van tekst, en die daarover wat te vertellen hebben.

De lokaties waar we bijeenkomen zijn vaak de moeite waard. Plekken waar je anders niet komt. Een mooie zaal van een restaurant in een klein stadje, een vergaderkamer van een kerk, een statige kamer in een voormalige watertoren. En dankzij uitstekende contacten bij Defensie zijn we met zekere regelmaat onder dak bij het leger. Dit keer op het terrein van het Kasteel van Breda waar sinds 1826 de Koninklijke Militaire Academie is gevestigd. Mooie gebouwen en zelfs een onderaardse gang. Het terrein is slechts een dag per jaar toegankelijk voor het publiek.

Als ik voorbij de slagbomen ben waar mijn paspoort is gecontroleerd, denk ik dat ik op afstand kan zien wie militair is (atletische stap) en wie van de ntg.

Binnen zijn de eersten bezig met de beamer die wel aan staat maar niet wil luisteren naar onze laptops. Gelukkig heeft Hans zijn eigen beamer mee, een klein dapper ding in een handzaam koffertje.

John Haltiwanger zou als eerste spreken maar hij heeft zijn paspoort niet bij zich en moet dus terug naar Amsterdam om die te halen ... ’Middags geeft hij zijn presentatie over een manier om poëzie te typesetten. Samen met een Spaanse vriend maakt hij muziek, de een rapt in het Engels en de ander in het Spaans en in het boekje bij hun cd moeten beide talen correct naast elkaar komen te staan. Omdat je vandaag de dag als beginnende musicus je muziek zowat weg moet geven om publiek te werven, moet je drukwerk goed verzorgd zijn om net die extra aandachtswaarde te bereiken, vertelt hij.

Tijdens zijn demonstratie gaat er wat mis op zijn computer en zo’n moment vind ik erg spannend. Iedereen reageert anders. Ik heb eens een van de sprekers zich zien verliezen in kalme concentratie. In serene rust ging ze haar setup na tot alles werkte en de zaal wachtte het in stilte af.

John Haltiwanger

Arthur Reutenauer

“I have to kill it, I guess” zegt later in de middag Arthur Reutenauer tijdens zijn live demonstratie van een TEX applicatie op de iPad. Een van de processen is uit de hand gelopen en het systeem heeft te kampen met een reusachtig lek aan geheugenruimte. Hoe dan ook is het indrukwekkend om te zien dat het in principe mogelijk is een TEX document te typen en te compileren op de iPad. Logischer zou het zijn, vertelt Arthur, als je zo'n tablet online kon gebruiken als invoer voor een TEX server elders op internet, waarna je de pdf zou kunnen downloaden of laten afdrukken.

De ‘verplichte’ groepsfoto

Hans Hagen vertelt (ook tijdens de lunch) over het typesetten van tekst die anders dan wij gewend zijn van rechts naar links loopt of van boven naar beneden op het papier. De techniek gaat me ver boven de pet maar ik vind het altijd interessant om Hans zijn manier van redeneren te volgen, zijn nuchtere aanpak te zien.

Taco Hoekwater, Hans van der Meer, en Hans Hagen

Willi, die veel weet van typografie maar ook van machines, beschrijft enthousiast wat hij zag op een drukkerij van bijbels waar twee katernen tegelijk werden geproduceerd. Het dunne papier wordt door de grote machines ongelooflijk snel gedrukt, gevouwen en genaaid, wat vooral mogelijk is doordat het papier zich zo soepel laat verwerken, ‘lebendig’ is zoals men in Zwitserland zegt.

Willi Egger

Willi Egger besluit de dag met een demonstratie van een manier die hij ontwikkelde om een A4-tje zo te printen en te vouwen dat je meteen een katern hebt. Dat vind ik een prachtige vondst. Op A4 kan het, maar het kan ook groter en als ik een betaalbare A3 printer koop, kan ik op die manier simpel een originele presentatie maken van tekst en fotografie. Mijn ‘vondst van de dag’ om thuis wat mee te gaan doen!

Frans Goddijn