TUG 2019 program

(* = presenter)

Friday	8:15 am	*registration*	
August 9	8:55 am	Boris Veytsman, TEX Users Group	*Welcome*
	9:00 am	Erik Braun, CTAN	*Current state of CTAN*
	9:30 am	Arthur Reutenauer, Uppsala, Sweden	*The state of XeTeX*
	10:00 am	Frank Mittelbach, LaTeX3 Project	*The LaTeX "dev" format*
	10:30 am	*break*	
	10:45 am	Frank Mittelbach	*Taming UTF-8 in pdfTeX*
	11:15 am	Uwe Ziegenhagen, Cologne, Germany	*Combining LaTeX with Python*
	11:45 am	Henri Menke, University of Otago	*Parsing complex data formats in LuaTeX with LPEG*
	12:15 pm	*lunch*	
	1:30 pm	Dick Koch, University of Oregon	*Big changes in MacTeX, and why users should never notice*
	1:45 pm	Nate Stemen, Overleaf, Inc.	*A few words about Overleaf*
	2:00 pm	Aravind Rajendran*, Rishikesan Nair T, Rajagopal C.V., STM Doc. Eng. Pvt Ltd	*Neptune — a proofing framework for LaTeX authors*
	2:30 pm	Pavneet Arora, Bolton, ON	*Rain Rain Go Away: Some thoughts on rain protection, and its uses*
	3:00 pm	*break*	
	3:15 pm	Shreevatsa R, Sunnyvale, CA	*What I learned from trying to read the TeX program*
	3:45 pm	Petr Sojka, Masaryk University & CSTUG	*TeX in Schools? Just Say Yes, given that ...*
	4:15 pm	Shakthi Kannan, Chennai, India	*XeTeX Book Template*
	4:45 pm	Jim Hefferon, St Michael's College	*What do today's newcomers want?*
	5:15 pm	*TUG Annual General Meeting*	
Saturday	8:55 am	*announcements*	
August 10	9:00 am	Petr Sojka, Ondřej Sojka	*The unreasonable effectiveness of pattern generation*
	9:30 am	Arthur Reutenauer	*Hyphenation patterns in TeX Live and beyond*
	10:00 am	David Fuchs	*What six orders of magnitude of space-time buys you*
	10:30 am	*break*	
	10:45 am	Tomas Rokicki, Palo Alto, CA	*Searchable PDFs with Type 3 bitmap fonts*
	11:15 am	Martin Ruckert, Hochschule Muenchen	*The design of the HINT file format*
	11:45 am	Doug McKenna, Mathemaesthetics, Inc.	*An interactive iOS math book using a new TeX interpreter library*
	12:15 pm	*lunch*	
	1:15 pm	*group photo*	
	1:30 pm	Jennifer Claudio, Sally Ha, Oak Grove H.S.	*A brief exploration of artistic elements in lettering*
	2:30 pm	William Adams, Mechanicsburg, PA	*Design into 3D: A system for customizable project designs*
	3:00 pm	*break*	
	3:15 pm	Boris Veytsman, Sch. Systems Biology, GMU	*Creating commented editions with TeX*
	3:45 pm	Behrooz Parhami, UC Santa Barbara	*Evolutionary changes in Persian and Arabic scripts to accommodate the printing press, typewriting, and computerized word processing*
	4:15 pm	Amine Anane, Montréal, QC	*Arabic typesetting using a Metafont-based dynamic font*
	4:45 pm	Takuto Asakura, National Institute of Informatics, Japan	*A TeX-oriented research topic: Synthetic analysis on mathematical expressions and natural language*
	5:15 pm	Herb Schulz, Naperville, IL	*Optional workshop: TeXShop tips & tricks*
	6:30 pm	*banquet*	Sheraton Palo Alto
Sunday	8:55 am	*announcements*	
August 11	9:00 am	Antoine Bossard, Kanagawa University	*A glance at CJK support with XeTeX and LuaTeX*
	9:30 am	Jaeyoung Choi, Saima Majeed, Ammar Ul Hassan, Geunho Jeon, Soongsil Univ.	*FreeType MF Module 2: Integration of METAFONT and TeX-oriented bitmap fonts inside FreeType*
	10:00 am	Jennifer Claudio, Emily Park, Oak Grove H.S.	*Improving Hangul to English translation*
	10:30 am	*break*	
	10:45 pm	Rishikesan Nair T*, Rajagopal C.V., Radhakrishnan C.V.	*TeXFolio — a framework to typeset XML documents using TeX*
	11:15 pm	Sree Harsha Ramesh, Dung Thai, Boris Veytsman*, Andrew McCallum, UMass-Amherst, (*) Chan Zuckerberg Initiative	*BibTeX-based dataset generation for training citation parsers*
	11:15 pm	Didier Verna, EPITA	*Quickref: A stress test for Texinfo*
	12:15 am	*lunch*	
	1:30 pm	Uwe Ziegenhagen	*Creating and automating exams with LaTeX & friends*
	2:00 pm	Yusuke Terada, Tokyo Educational Institute	*Construction of a digital exam grading system using TeX*
	2:30 pm	Chris Rowley*, Ulrike Fischer, LaTeX3 Project	*Accessibility in the LaTeX kernel — experiments in tagged PDF*
	3:00 pm	*break*	
	3:15pm	Ross Moore, Macquarie Univ.	*LaTeX 508 — creating accessible PDFs*
	≈3:45 pm	*end*	

William Adams

Design into 3D: A system for customizable project designs

Design into 3D is a system for modeling parametric projects for manufacture using CNC machines. It documents using OpenSCAD to allow a user to instantly see a 3D rendering of the result of adjusting a parameter in the Customizer interface, parameters are then saved as JSON files which are then read into a LuaLaTeX file which creates a PDF as a cut list/setup sheet/assembly instructions and MetaPost to create SVG files which may be loaded into a CAM tool. A further possibility is using a tool such as TPL (Tool Path Language) to make files which are ready to cut.

This was initially a (funded) Kickstarter `https://kickstarter.com/projects/designinto3d/design-into-3d-a-book-of-customizable-project-desi` and is being developed as a wiki page on the Shapeoko project (`https://wiki.shapeoko.com/index.php/Design_into_3D`) with code on GitHub (`https://github.com/WillAdams/Design_Into_3D` and a number of sample files and project have already been made: `https://cutrocket.com/p/5c9fb998c0b69/`, `https://cutrocket.com/p/5cb536396c281/`, `https://cutrocket.com/p/5cba77918bb4b/`; and this is tied into a Thingiverse project (`https://www.thingiverse.com/thing:3575705` and an on-line box generator (`http://chaunax.github.io/projects/twhl-box/twhl.html`).

Amine Anane

Arabic typesetting using a Metafont-based dynamic font

Arabic script is a cursive script where the shape and width of letters are not fixed but vary depending on the context and the justification needs. A typesetter must consider those dynamic properties of letters to achieve a high-quality text comparable to Arabic calligraphy.

In this talk I will present a parametric font that has been designed as a first step towards such high-quality typesetter. The font is based on Metafont language which can generate a glyph with a given width dynamically and respecting the curvilinear nature of Arabic letters. It uses an extended version of OpenType to support the varying width of the glyphs. I will demonstrate a graphical tool which has been developed specifically to facilitate the design of such dynamic fonts. As a case study, I will compare a handwritten Quranic text with a one generated with this dynamic font and I will conclude by highlighting some future works towards a complete high-quality Arabic typesetter.

Takuto Asakura

A TeX-oriented research topic: Synthetic analysis on mathematical expressions and natural language

Since mathematical expressions play fundamental roles in Science, Technology, Engineering and Mathematics (STEM) documents, it is beneficial to extract meanings from formulae. Such extraction enables us to construct databases of mathematical knowledge, search for formulae, and develop a system that generates executable codes automatically.

TeX is widely used to write STEM documents and provides us with a way to represent *meanings* of elements in formulae in TeX by macros. As a simple example, we can define a macro `\def\inverse#1{#1^{-1}}`, and use it as `$\inverse{A}$` in documents to make it clear that the expression means "the inverse of matrix A" rather than "value A to the power of -1". Using such meaningful representations is useful in practice for maintaining document sources, as well as converting TeX sources to other formal formats such as first-order logic and content markup in MathML. However, this manner is optional and not forced by TeX. As a result, many authors neglect it and write messy formulae in TeX documents (even with wrong markup).

To make it possible to associate elements in formulae and their meanings automatically instead of requiring it of authors, recently I began research on detecting or disambiguating the meaning for each element in formulae by conducting synthetic analyses on mathematical expressions and natural language text. In this presentation, I will show the goal of my research, the approach I'm taking, and the current status of the work.

Antoine Bossard

A glance at CJK support with XeTeX and LuaTeX

From a typesetting point of view, the Chinese and Japanese writing systems are peculiar in that their characters are concatenated without ever using spaces to separate them or the meaning units (i.e., "words" in our occidental linguistic terminology) they form. And this is also true for sentences: although they are usually separated with punctuation marks such as periods, spaces remain unused. Conventional typesetting approaches, TeX in our case, thus need to be revised in order to support the languages of the CJK group: Chinese, Japanese and, to a lesser extent, Korean. While more or less complete solutions to this issue can be found, in this article we give and pedagogically discuss a minimalistic implementation of CJK support with the Unicode-capable XeTeX and LuaTeX typesetting systems.

Erik Braun

Current state of CTAN

The "Comprehensive TEX Archive Network" is the authoritative place where TEX-related material is collected.

Developers can upload their packages, and the distributions use it to pick up their packages. The TEX Catalogue's entries can be accessed via the website, and all the data can be accessed from mirror servers all over the world.

The talk will give an overview of the current state of CTAN, recent developments, and most common problems. In further discussion, feedback from users and developers is very welcome.

Jaeyoung Choi, Saima Majeed, Ammar Ul Hassan, Geunho Jeon

FreeType MF Module 2: Integration of METAFONT and TEX-oriented bitmap fonts inside FreeType

METAFONT is the structured font definition language that can generate variants of different font styles by changing parameter values. It doesn't require creating a new font file for every distinct font design. It generates as output a Generic Font file (GF) bitmap and (if requested) its corresponding TEX Font Metric file (TFM). These fonts can be utilized on devices of any resolution without creating new font files, according to the preferred size. These benefits can also be applied to complex characters such as CJK (Chinese-Japanese-Korean) glyphs rather than only applying to alphanumeric characters. However, METAFONT, GF, and Packed Fonts (PK compressed form of GF) cannot be utilized beyond the TEX environment as they require additional conversion overhead. Furthermore, existing font engines, notably FreeType, do not support such fonts.

In this paper, FreeTypeMFModule2 is proposed for the FreeType font engine. The proposed module can directly support METAFONT, GF, and PK fonts inside FreeType in a GNU/Linux environment. To utilize such fonts, users are not required to preconvert bitmaps into outlines: the proposed module automatically performs such conversions without relying on other libraries. By using the proposed module, users can generate variants of font styles (via METAFONT) and use it on the desired resolution devices (via GF). The proposed font module reduces the creation time and cost for creating the distinct fonts styles. Furthermore, it reduces the conversion and configuration overhead for TEX-oriented fonts.

Jennifer Claudio, Sally Ha

A brief exploration of artistic elements in lettering

This non-technical talk explores the stylistic elements of letter forms as used in arts and culture through an examination of elongations and decorations with a focus on the letter E. Samples discussed are derived from the calligraphy of Don Knuth's *3:16*, in samples of street art, and in typographic branding.

Jennifer Claudio, Emily Park

Improving Hangul to English translation

Real time translation of languages using camera input occasionally results in awkward failures. As a proposed method of assisting such tools for Korean (Hangul) to English translation, an optical method is proposed to help translation algorithms first assess whether the Korean text has been written as English syllables in Korean or in true Korean vocabulary before producing translated phrases.

David Fuchs

What six orders of magnitude of space-time buys you

TEX and METAFONT were designed to run acceptably fast on computers with less than 1/1000th the memory and 1/1000th the processing power of modern devices. Many of the design trade-offs that were made are no longer required or even appropriate.

Federico Garcia-De Castro

An algorithm for music slurs in METAFONT

This paper describes an algorithm that draws beautiful slurs around given notes (or other points to avoid). I have been working on such an algorithm on and off since around 2004 — when commercial music typesetting software did not provide for automatic, let alone beautiful, slurs. Along the way I tried many kinds of approaches, some of them inspired by METAFONT routines such as **superellipse**, the **flex** macro, and the **transform** infrastructure (which, for example, is what slants the `\textsc` font out of a vertical design). The usual fate of these attempts was one of promise followed by interesting development leading to collapse — there usually were too many independent and variables interacting chaotically. Earlier this year I finally found a robust, elegant algorithm. I will present all of the attempts and describe what makes the final algorithm unique, and compare it to the way commercial software does slurs today. This is a graphic presentation, rather than musical.

Jim Hefferon

What do today's newcomers want?

The reddit LATEX subgroup gets questions from many beginners. Whereas StackExchange edits the questions, here you see a less filtered version of what people are working on. We will examine an archive of past postings to get some data about what it is that today's newcomers are trying to do, how they are trying to do it, and where they are struggling.

Shakthi Kannan

XeTeX Book Template

The XeTeX Book Template is a free software framework for authors to publish multilingual books using XeTeX. You can write the content in GNU Emacs Org-mode files along with TeX, and the build scripts will generate the book in PDF. The Org-mode files are exported to TeX files, and Emacs Lisp post-processing is done prior to PDF generation. Babel support with Org-mode TeX blocks allows one to selectively export content as needed. The framework separates content from presentation.

A style file exists for specifying customized page titles, setting margins, font specification, chapter title and text formatting, page style, spacing etc. The framework has been used to publish books containing Tamil, Sanskrit and English. It is released under the MIT license and available at `https://gitlab.com/shakthimaan/xetex-book-template`. In this talk, I will explain the salient features of the XeTeX Book Template, and also share my experience in creating and publishing books using the framework.

Dick Koch

Big changes in MacTeX, and why users should never notice

Just before the TeX Live 2019 release, Apple developers received a notice about changes coming in macOS 10.15 for install packages and the programs they contain. After some panic, experiments convinced us that we can accommodate the changes. But if you wrote one of the binaries in TeX Live, we could use your help, even if you never use a Mac.

Doug McKenna

An interactive iOS math book using a new TeX interpreter library

The current TeX ecosystem is geared towards creating only static PDF or other output files. Using a re-implementation of a TeX language interpreter as a library linked into an iOS client program that simulates a document on a device with a touch screen, the author will demonstrate a new PDF-free ebook, *Hilbert Curves*, that typesets itself each time the application launches. The library maintains all TeX data structures for all pages in memory after the typesetting job is done, exporting pages as needed while the user reads the book and interacts with its dynamic illustrations. This design also allows text-searching the document's TeX data structures while the ebook is "running".

Henri Menke

Parsing complex data formats in LuaTeX with LPEG

Although it is possible to read external files in TeX, extracting information from them is rather difficult. Ad-hoc solutions tend to use nested if statement or regular expressions provided by several macro packages. However, these quick hacks don't scale well and quickly become unmaintainable.

LuaTeX comes to the rescue with its embedded LPEG library for Lua. LPEG provides a Domain Specific Embedded Language (DSEL) that allows to write grammars in a natural way. In this talk I will give a quick introducing to Parsing Expression Grammars (PEG) and then show how to write simple parsers in Lua with LPEG. Finally we will build a JSON parser to demonstrate how easy it is to even parse complex data formats.

Frank Mittelbach

The LaTeX "dev" format

What prevents banana software (gets ripe at the customer site)? Proper testing! But this is anything but easy.

The talk will give an overview about the efforts made by the LaTeX Project Team over the years to provide high-quality software and explains the changes that we intend to make this summer to improve the situation further.

Frank Mittelbach

Taming UTF-8 in pdfTeX

To understand the concepts in `pdflatex` for processing UTF-8 encoded files it is helpful to understand the models used by the TeX engine and earlier models used by LaTeX on top of TeX. This talk gives a short historical review of that area and explains

— how it is possible in a TeX system that only understands 8-bit input to nevertheless interpret and process UTF-8 files successfully;
— what the obstacles are that can be and have been overcome;
— what restrictions remain if one doesn't switch to a Unicode-aware engine such as LuaTeX or XeTeX.

The talk will finish with an overview about the improvements with respect to UTF-8 that will be activated in LaTeX within 2019 and how they can already be tested right now.

Ross Moore

LaTeX 508 – creating accessible PDFs

Authoring documents that are accessible to people with disabilities is not only the morally correct thing to be doing, but is now required by law, at least for U.S. Government offices and agencies, through the revised Section 508 of the U.S. Disabilities Act (2017). It is likely to eventually become so also for any affiliated institutions, such as universities, colleges and many schools.

For mathematics and related scientific fields, it thus becomes imperative that we be able to produce documents using LaTeX that conform to the accessible standard ANSI/AIIM/ISO 14289-1:2016 (PDF/UA-1). This is far more rigorous than standard PDF, in terms of capturing document structure, as well as all content associated with each particular structural element.

In this talk we show an example of a research report produced as PDF/UA for the U.S. National Parks Service. We illustrate several of the difficulties involved with creating such documents. This is due partly to the special handling required to encode the structure of the technical information such as appears on the title page, and inside-cover pages, as well as tabular material and images throughout the body of the document. But there are also difficulties that are due to the nature of TeX itself, and the intricacy of LaTeX's internal programming.

Behrooz Parhami

Evolutionary changes in Persian and Arabic scripts to accommodate the printing press, typewriting, and computerized word processing

The Persian script has presented difficulties for printing ever since printing presses were introduced in Iran in the 1600s. The appearance of typewriters created additional problems and the introduction of digital computers added to the design challenges. The Arabic script presented nearly identical complications. These difficulties persisted until high-resolution dot-matrix printers and display devices offered greater flexibility to font designers and the expansion of the computer market in the Middle East attracted investments to help solve the problems.

Nevertheless, certain peculiarities of Persian and Arabic scripts have led to legibility and aesthetic quality issues to persist in many cases. In this presentation, I will enumerate some of the features of modern Persian and Arabic scripts that made implementation on modern technologies quite challenging and review the issues presented by, and some of the solutions proposed for, each new generation of computer printers and display devices.

Interestingly, the same features that make legible and pleasant printing/displaying difficult also lead to challenges in automatic text recognition. I will conclude with an overview of current state of the art and areas that still need further work.

Aravind Rajendran (presenter), Rishikesan Nair T, Rajagopal C.V.

Neptune — a proofing framework for LaTeX authors

In academic publishing, LaTeX authors may be considered problem mongers since they insist on better typography, adherence to conventions (particularly in math equations), usage of their finely crafted LaTeX sources for final output and other myriad benefits offered by LaTeX. In recent times, galley proofs are provided to authors as editable sources rendered as a web page in XML or HTML format. Authors who have submitted their articles in LaTeX format are seldom comfortable viewing and editing the output as a web page since math equations do not provide the original LaTeX sources to edit, Ti*k*Z graphics, XY-pic and commutative diagrams, `prooftree` math, and the like have been replaced with their respective graphics, thus precluding the opportunity to edit in case a mistake is found; source listing with packages like `listings` suffer a similar fate, the woes are many and much more.

Hence, LaTeX authors are not without cause when they complain of publishers' lack of academic and semantic sensibilities. Our program Neptune is an answer for all these problems, wherein a LaTeX author can be provided with copy-edited LaTeX sources and corresponding PDF output in the final print format side by side with enough facilities to navigate between source and PDF, a navigable list of track changes showing copy edits that can be accepted/rejected, a navigable list of author edits during the proofing session, comparison of pre- and post-proof LaTeX sources side by side with ability to discard any edit, comparison of pre- and post-PDF versions, navigable query lists, multiple sessions for proofing, standard editor features, etc. This presentation will show all these features with the help of a live demonstration.

Sree Harsha Ramesh, Dung Thai, Boris Veytsman (presenter), Andrew McCallum

BibTeX-based dataset generation for training citation parsers

A human can relatively easily read a bibliography list and parse each entry, i.e., recognize authors' names, item title, venue, year and volume information, pagination, urls and doi numbers, etc., using such cues as punctuation and font changes. This is even more impressive since there is no universal standard of bibliography typesetting; virtually all publishers and journals use their own "house styles".

It has been a challenge to make a machine to do this task, which is important for digitization of the scientific literature. One of the problems is the lack of labeled data: parsed bibliography items, suitable to train the algorithms. It is cumbersome and expensive to translate a large number of bibliographies into a machine readable format. Thus the largest dataset published so far has 2479 entries.

In this work (it has been partially presented at AKBC 2019) we describe a way to overcome the problem. We start with a bibliography already in machine readable format, and typeset it using BibTeX and LaTeX. The resulting labeled dataset is suitable for training algorithms. We used Nelson Beebe's archive of 1.41 million BibTeX entries and typeset it with 275 bibliography styles from the recent TeX Live collection. After deleting some problematic entry-style combinations that did not compile (mostly due to non-standard fields), we obtained 185 million labeled samples, improving the state of the art by five orders of magnitude.

Arthur Reutenauer

Hyphenation patterns in TeX Live and beyond

Hyphenation has always been one of the strengths of TeX, which since its TeX82 version has had an efficient algorithm to describe how words should be divided across linebreaks. This algorithm was successfully used to produce *hyphenation patterns* for many languages, as testified to by the many pattern sets present in TeX distributions.

In 2008 Mojca Miklavec and myself overhauled all the pattern sets in the TeX Live distribution in order to rationalise them and prepare them for LuaTeX, which expects Unicode input, while keeping them usable by earlier TeX engines. The result of this effort was the somewhat misleadingly named package `hyph-utf8`, and was soon adopted by the MiKTeX distribution also.

We have since then been maintaining the package, which now supports about 80 languages and language variants. We also initiated many collaborations with other actors in the free software world, as TeX's hyphenation algorithm is used in some form by OpenOffice, LibreOffice, and Firefox, as well as lesser-known programs such as Apache FOP.

I will discuss some of the challenges we encountered, and our future plans.

Arthur Reutenauer

The state of XeTeX

I will present the current state of XeTeX and describe our plans for the future.

Rishikesan Nair T (presenter), Rajagopal C.V., Radhakrishnan C.V.

TeXFolio — a framework to typeset XML documents using TeX

XML is now accepted as the *de facto* archival standard in the world of academic publishing. As a consequence, publishers insist on XML as one of the deliverable output formats when typesetters are contracted. The primary aim is to archive the content so that future use of the same without surprises is ensured.

However, some publishers want a bit more than this, insisting on generating PDF output, both for print and web delivery, directly from XML sources which is considered to be the definitive version of the content, instead of from author-provided sources. The key objective is to keep the fidelity between the printed and archived versions of content. This is termed as "XML-first" workflow. There are many typesetting systems available to undertake the job for text, but not for sources with heavy math content.

TeXFolio is a web-based framework on the cloud that will fill this gap to generate standards-compliant, hyperlinked, bookmarked PDF output directly from XML sources with heavy math content using TeX. TeXFolio is a complete journal production system as well. It can produce strip-ins which are alternate GIF/SVG images of MathML content. In addition, DOI look-up, HTML rendering of XML/MathML, and the whole dataset generation according to the customer's specification; running customer-specific validation tools is also possible. This presentation will demonstrate this and various other features of TeXFolio.

Tomas Rokicki

Searchable PDFs with Type 3 bitmap fonts

Searching and copying text in PDF documents generated by TeX works well when Type 1 versions of the fonts are used, but works poorly, if at all, when Type 3 (bitmap) fonts are used. This is due to decisions made in 1986 by your author. We improve this situation with some relatively small changes to dvips.

Martin Ruckert

The design of the HINT file format

The HINT file format is intended as a replacement of the DVI or PDF file format for on-screen reading of TeX output. Its design should therefore support the following features: reflow of text to fill a window of variable size, convenient navigating of text with links in addition to paging forward and backward, efficient rendering on mobile devices, simple generation from existing TeX input files, and an exact match of traditional TeX output if the window size matches the paper size.

In this talk, I will present the key elements of the design, discuss past and future design decissions, and demonstrate the new file format using a working prototype.

Shreevatsa R

What I learned from trying to read the TEX program

As we know, TEX is written in a system called `WEB` that exemplifies the idea of literate programming (or programs as literature), and has been published as a book. Indeed, many good and experienced programmers have read the program with interest. But what if the reader is neither good nor experienced? Here we discuss some (more or less superficial) obstacles that stymie the novice modern programmer trying to make sense of the TEX program, and how they can be overcome.

Petr Sojka, Ondřej Sojka

The unreasonable effectiveness of pattern generation

Languages are constantly evolving organisms and so are the hyphenation rules and needs. The effectiveness and utility of TEX's hyphenation has been proven by its usage in almost all typesetting systems in use today. The current Czech hyphenation patterns were generated in 1995 and no hyphenated word database is freely available.

We have developed a new Czech word database and have used `patgen` to efficiently generate new effective Czech hyphenation patterns and evaluated its generalization qualities.

We have achieved almost full coverage on the training dataset of 3,000,000 words and validated the patterns on the testing database of 100,000 words approved by Czech Academy of Science linguists.

Our pattern generation case study serves as an example of an effective solution of widespread dictionary problem. The study has shown the versatility of Liang's approach. The unreasonable effectiveness of TEX's pattern technology has lead to applications that are and will be used even more widely 40 years after its inception.

Petr Sojka

TEX in Schools? Just Say Yes, given that ...

TEX is used in schools, such as my own Masaryk University, for many purposes: for writing theses, essays and papers by students, for teaching electronic publishing, literate programming, writing scientific papers, quizzes, slides by faculty, and generating documents and web pages from university databases by university information systems and staff. TEX and related technologies are systematically supported and deployed at the Faculty of Informatics for more than a quarter century. In the talk we recall the support and projects we have realized in the past, evaluate the outcomes, and discuss possible future deployment of TEX-related technologies.

Yusuke Terada

Construction of a digital exam grading system using TEX

At our school in Japan, large-scale paper exams are held on a regular basis. The number of examinees is enormous, and the grading must be finished within a short period of time. Improving efficiency was strongly needed. So I developed a digital exam grading system using TEX. TEX and related software play a core role in the system, co-operating with iPad and Apple Pencil.

In this presentation, I would like to present how TEX can be effectively applied to constructing the digital exam grading system. I will also mention the unexpected difficulties that I faced in the actual large-scale operations and the way I have overcome them.

Didier Verna

Quickref: A stress test for Texinfo

Quickref is a global documentation project for the Common Lisp ecosystem. It creates reference manuals automatically by introspecting libraries and generating a corresponding documentation in Texinfo format. The Texinfo files may subsequently be converted into PDF or HTML. Quickref is non-intrusive: software developers do not have anything to do to get their libraries documented by the system.

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Common Lisp ecosystem at once. The result is a website containing almost two thousand reference manuals. Quickref provides a Docker image for an easy recreation of this website, but a public version is also available and kept up to date at `quickref.common-lisp.net`.

Quickref constitutes an enormous (and successful) stress test for Texinfo, and not only because of the number of files generated and processed. The Texinfo file sizes range from 7K to 15M (double that for the generated HTML). The number of lines of Texinfo code in those files extend from 364 to 285,020, the indexes may contain between 14 and 44500 entries, and the processing times vary from .3s to 1m 38s per file.

In this talk, I will make a real-time demonstration of the system, give an overview of its design and architecture, describe the challenges and difficulties in generating valid Texinfo code automatically, and finally put some emphasis on the currently remaining problems and deficiencies.

Boris Veytsman

Creating commented editions with TEX

There are many ways to create critical editions, more general "commented" editions, with layers of commentary accompanying the main text. Packages like `bigfoot`, `manyfoot`, `*edmac` and others help to typeset different variants of them. Recently Instituto Nacional de Matemática Pura e Aplicada, Brazil (IMPA) requested yet another version for its series of mathematics textbooks for Brazilian schools. This series is designed as combinations of students' textbooks and the teacher's book. The teacher's book reproduces pages from the students' book, and adds a layer of comments around them. The package `commedit`, commissioned by IMPA, hacks the output routine for the production of students' and teachers' version from a single source.

Uwe Ziegenhagen

Creating and automating exams with LATEX & friends

LaTeX offers sophisticated document classes and packages to create exam materials. In this talk we show how to create exams using the `exam` class and how the document creation can be fine-tuned to allow individualized exams.

Uwe Ziegenhagen

Combining LATEX with Python

Even older than Java, Python has achieved a lot of popularity especially in recent years. It is an easy-to-learn general purpose programming language, with strong abilities not only in fancy topics such as machine learning and artificial intelligence.

In this talk we want to present scenarios where LaTeX documents can be enhanced by Python scripts. We will show examples where LaTeX documents are automatically generated by Python or receive content from Python scripts.

A glance at CJK support with XeTeX and LuaTeX

Antoine Bossard

Abstract

From a typesetting point of view, the Chinese and Japanese writing systems are peculiar in that their characters are concatenated without ever using spaces to separate them or the meaning units (i.e., "words" in our occidental linguistic terminology) they form. And this is also true for sentences: although they are usually separated with punctuation marks such as periods, spaces remain unused. Conventional typesetting approaches, TeX in our case, thus need to be revised in order to support the languages of the CJK group: Chinese, Japanese and, to a lesser extent, Korean. While more or less complete solutions to this issue can be found, in this article we give and pedagogically discuss a minimalistic implementation of CJK support with the Unicode-capable XeTeX and LuaTeX typesetting systems.

1 Introduction

The Chinese, Japanese and Korean writing systems are conventionally gathered under the CJK appellation. The Chinese writing system consists of the Chinese characters, which can be in simplified or traditional form, amongst other character variants [1]. The (modern) Japanese writing system is made of the Chinese characters and the kana characters. The Chinese and Japanese writing systems concatenate characters without ever separating them with spaces. The Korean writing system consists mainly of hangul characters, with in addition the Chinese characters, which are however rarely used nowadays. Although modern Korean separates words with spaces, traditionally, the Korean writing system does not (as an illustration, see for instance Sejong the Great's 15th century manuscript Hunminjeongeum[1]).

Notwithstanding other critical issues such as fonts, by not relying on spaces between characters or words, the CJK scripts are a challenge to conventional typesetting solutions such as TeX. In fact, the algorithms for word-breaking, which conventionally occurs at spaces (plus hyphenation), become inapplicable.

On a side note, even though we consider hereinafter the CJK writing systems, this discussion can be extended to related scripts such as Tangut and Vietnam's Chữ Nôm.

[1] King Sejong (世宗) introduced hangul in the *Hunminjeongeum* (訓民正音) manuscript (1443–1446).

In this paper, we provide a glance at CJK support with XeTeX and LuaTeX by giving a minimalistic implementation for these oriental scripts. This work is both a proof of concept and a pedagogical discussion on how to achieve CJK support as simply as possible with the aforementioned typesetting solutions. Both XeTeX and LuaTeX support Unicode, which enables us to focus on typesetting issues, leaving encoding and font considerations aside.

The rest of this paper is organised as follows. Technical discussion of the proposed implementation is conducted in Section 2. The state of the art and paper contribution are summarised in Section 3. Finally, this paper is concluded in Section 4.

2 A minimalistic implementation

We describe here the proposed minimalistic implementation of CJK support with XeTeX and LuaTeX step by step in a pedagogical manner: paragraph management (Step 1) is addressed in Section 2.1, Latin text mingling (Step 2) in Section 2.2, Latin text paragraph (Step 3) in Section 2.3, Korean text paragraph (Step 4) in Section 2.4 and sophisticated line-breaking (Step 5) in Section 2.5. "Latin text" here designates text written with the Latin alphabet, or similar; for instance English and French text.

A handful of TeX commands are used hereinafter without being detailed; see [4] for those that are not self-explanatory. The document preamble specifies nothing particular. The fontspec package [10] is loaded for facilitated font manipulation, and, as detailed in the rest of this section, since it is considered without loss of generality that the document consists of Chinese or Japanese paragraphs by default, the main font of the document is set accordingly (e.g., `\setmainfont{Noto Serif CJK JP}` [3]).

2.1 Paragraph management

A conventional approach to break long character sequences (i.e., Chinese or Japanese characters in our case) is to insert between each two glyphs a small amount of horizontal space so that TeX can split the sequence across multiple lines (see for instance [13]). Without such extra space, line breaks could still occur thanks to hyphenation, but this is not applicable in the CJK case. We rely on a "scanner" macro to transform a paragraph by interleaving space between its characters. In practice, according to the TeX terminology, this extra space will be a horizontal skip of 0pt width and ±1pt stretch.

The scanner macro is a recursive process that takes one token (e.g., a character) as single parameter and outputs it with on its right extra horizontal space. The recursion stops when the parameter token is the stop signal (more on this later); in this case, the macro outputs `\par`, thus materialising the end of the paragraph. The scanner macro `\cjk@scan` is defined as follows.

```
\def\cjk@scan#1{% #1: single token
 \ifx#1\cjk@stop% stop signal detected
  \par% so, complete the paragraph
 \else
  #1% display the current character
  \hskip 0pt plus 1pt minus 1pt\relax% space
  \expandafter\cjk@scan% recursive call
 \fi
}
```

The above scanner is started by the `\cjk@scanstart` macro whose primary objective is to append the stop signal `\cjk@stop` at the end of the paragraph that is about to be transformed. This initial macro takes one parameter: the paragraph to transform. In a pattern matching fashion, a paragraph is taken as a whole by setting `\par` as delimiter for the parameter of the `\cjk@scanstart` macro. This will require inserting `\par` once the paragraph has been transformed though, since the `\par` command that ends the paragraph is treated as a delimiter by the macro and thus skipped. In addition, each paragraph needs to be ended by a blank line (i.e., `\par`) to content this pattern matching. The scanner starting macro is given below.

```
\def\cjk@scanstart#1\par{% #1: paragraph
 \cjk@scan#1\cjk@stop% append \cjk@stop
}
```

In this work, paragraphs are considered to be written in Chinese or Japanese by default. Hence, paragraph typesetting mode selection by means of a command such as `\CHJP{text}` is not suitable. We rely on the `\everypar` token parameter so as to trigger the transformation of each paragraph with the scanner previously described. This is simply done as follows:

```
\everypar={\cjk@scanstart}
```

or, in a safer manner [2]:

```
\everypar=\expandafter{\the\everypar
 \cjk@scanstart}
```

人民身體之自由應予保障。除現行犯之逮捕由法律另定外，非經司法或警察機關依法定程序，不得逮捕拘禁。非由法院依法定程序，不得審問處罰。非依法定程序之逮捕、拘禁、審問、處罰，得拒絕之。

人民因犯罪嫌疑被逮捕拘禁時，其逮捕拘禁機關應將逮捕拘禁原因，以書面告知本人及其本人指定之親友，並至遲於二十四小時內移送該管法院審問。本人或他人亦得聲請該管法院，於二十四小時內向逮捕之機關提審。

(a) (b)

Figure 1: Before (a) and after (b) paragraph transformation: line breaking now enabled (traditional Chinese text example).

An illustration of the result of this paragraph transformation is given in Figure 1 (Chinese and Japanese paragraphs).

2.2 Latin text mingling

It is often the case that Latin text such as English words, expressions or sentences is mingled within Chinese or Japanese paragraphs. In the previously described paragraph transformation method, spaces, if any, are "gobbled" and never passed as parameter for the scanner macro `\cjk@scan`. This is not really a problem for Chinese and Japanese text since as explained they do not rely on spaces. But now that we are considering Latin text mingling in such paragraphs, spaces need to be retained since Latin text, such as English, rely on spaces for instance to separate words.

Without going too much into details, to force TEX to also pass spaces as parameters to the scanner macro, spaces need to be made *active*, as per the TEX terminology. Hence, it suffices to call the `\obeyspaces` macro, whose purpose is exactly to make spaces active, at the beginning of the document. In addition, the scanner macro is refined to avoid adding extra space when the current character is a space; see below.

```
\def\cjk@scan#1{%
 \ifx#1\cjk@stop
  \par
 \else
  #1%
  \if#1\space% no extra space if #1 is a space
  \else
   \hskip 0pt plus 1pt minus 1pt\relax
  \fi
  \expandafter\cjk@scan
 \fi
}
```

Two remarks are made next. First, it should be noted that Latin text mingled within Chinese or Japanese paragraphs is treated just as Chinese

日本国民は、正当に選挙された国会における代表者を通じて行動し、われらとわれらの子孫のために、諸国民との協和による成果と、わが国全土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存することを宣言し、この憲法を確定する。そもそも国政は、国民の厳粛な信託によるものであつて、その権威は国民に由来し、その権力は国民の代表者がこれを行使し、その福利は国民がこれを享受する。これは人類普遍の原理であり、この憲法は、かかる原理に基くものである。われらは、これに反する一切の憲法、法令及び詔勅を排除する。We,theJapanesepeople...

(a)

日本国民は、正当に選挙された国会における代表者を通じて行動し、われらとわれらの子孫のために、諸国民との協和による成果と、わが国全土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存することを宣言し、この憲法を確定する。そもそも国政は、国民の厳粛な信託によるものであつて、その権威は国民に由来し、その権力は国民の代表者がこれを行使し、その福利は国民がこれを享受する。これは人類普遍の原理であり、この憲法は、かかる原理に基くものである。われらは、これに反する一切の憲法、法令及び詔勅を排除する。We, the Japanese people...

(b)

Figure 2: Before (a) and after (b) making spaces active: Latin text mingling now retains spaces (Japanese text example).

or Japanese text: extra space is inserted between glyphs. Therefore, line and word breaking for mingled Latin text can occur anywhere, and thus no word-breaking by hyphenation will ever happen since never necessary due to the extra space added between glyphs. Second, even though no extra space is added after a space character, extra space is still added before a space character. This issue will be tackled in a subsequent section.

An illustration of the result of this refined paragraph transformation is given in Figure 2.

2.3 Latin text paragraph

Because the `\obeyspaces` macro has been called so as to typeset Chinese and Japanese paragraphs, Latin text paragraphs would be typeset just as those, that is, with extra space added between consecutive glyphs (except after spaces). As a result, and as explained above, line and word breaking would not be satisfactory.

Hence, we next enable the proper typesetting of Latin text paragraphs, that is, paragraphs that include spaces between words. To this end, we define the `\iflatin` conditional statement that will be used to distinguish Latin text paragraphs from others. The flag command `\latinfalse` is called at the beginning of the document to reflect that Chinese and Japanese paragraphs are by default. Latin text paragraphs are marked as such by calling the flag command `\latintrue` at the beginning of the paragraph. The scanner starting macro `\cjk@scanstart` is adjusted so as to not start the scanner in case the Latin flag is set.

As the `\obeyspaces` macro has been previously called, spaces are active characters; this setting needs to be reverted in the case of Latin text paragraph in order to have proper line and word breaking. Hence, the scanner starting macro in addition reverts spaces from the active state back to their

日本国民は、正当に選挙された国会における代表者を通じて行動し、われらとわれらの子孫のために、諸国民との協和による成果と、わが国全土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存することを宣言し、この憲法を確定する。

We, the Japanese people, acting through our duly elected representatives in the National Diet, determined that we shall secure for ourselves and our posterity the fruits of peaceful cooperation with all nations and the blessings of liberty throughout this land, and resolved that never again shall we be visited with the horrors of war through the action of government, do proclaim that sovereign power resides with the people and do firmly establish this Constitution.

(a)

日本国民は、正当に選挙された国会における代表者を通じて行動し、われらとわれらの子孫のために、諸国民との協和による成果と、わが国全土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存することを宣言し、この憲法を確定する。

We, the Japanese people, acting through our duly elected representatives in the National Diet, determined that we shall secure for ourselves and our posterity the fruits of peaceful cooperation with all nations and the blessings of liberty throughout this land, and resolved that never again shall we be visited with the horrors of war through the action of government, do proclaim that sovereign power resides with the people and do firmly establish this Constitution.

(b)

Figure 3: Before (a) and after (b) Latin mode enabling: Latin text now properly typeset (Japanese and English text example).

default state in the case of a Latin text paragraph. The refined code is given below.

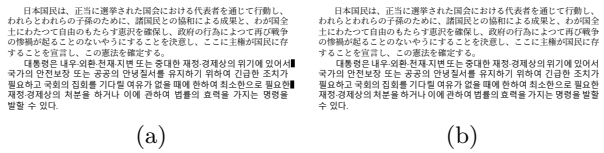
```
\newif\iflatin % flag to detect whether to scan
\latinfalse % flag initially set to false

\def\cjk@scanstart#1\par{
 \iflatin% if Latin text paragraph, don't scan
  \catcode`\ =10% revert \obeyspaces
  #1\par% display the paragraph normally
  \latinfalse
 \else
  \cjk@scan#1\cjk@stop
 \fi
}
```

An illustration of the result of this refined paragraph transformation is given in Figure 3.

2.4 Korean text paragraph

Let us now discuss the case of Korean text paragraph typesetting. As mentioned in introduction, modern Korean relies on spaces to separate words. Hence, such paragraphs are treated as a Latin text paragraph, concretely being marked with the `\latintrue` flag. Yet, because Korean glyphs (i.e., hangul or hanja) are wider than Latin ones, the width of spaces is adjusted. In addition, a font switch is also used to select a Korean font since it is frequent that Korean glyphs are not included inside the default font used for Chinese and Japanese paragraph typesetting.

Such settings are applied at the beginning of the paragraph, thus embedding the paragraph into a group for font selection and the adjusted space setting. Hence, the paragraph starts with



(a) (b)

Figure 4: Before (a) and after (b) space width adjustment for Korean text: no more overfull hboxes (Japanese and Korean text example).

a '{' token and it is required to leave the vertical mode for proper parsing of the paragraph when used as parameter of the scanner starting macro `\cjk@scanstart`. Precisely, the problem with starting the paragraph with a command like `{\malgun` (e.g., font switch) is that TEX is still in vertical mode when it processes it. Switching to horizontal mode starts a new paragraph and thus triggers `\everypar`, but then with an unmatched '}' remaining (i.e., the one corresponding to, say, the font switch) at the end of the paragraph, and thus the parsing error.

For convenience, these Korean text paragraph settings are gathered under the `\korean{}` macro as defined below.

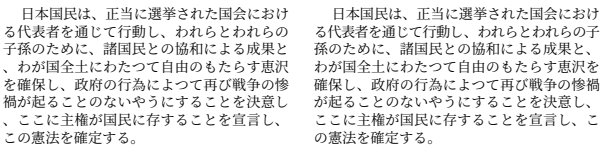
```
\def\korean#1{
 \latintrue% activate the Latin mode
 \leavevmode% leave the vertical mode
 {% Adjust the space size:
  \spaceskip=\fontdimen2\font plus
   3\fontdimen3\font minus
   3\fontdimen4\font% ×3 stretch and shrink
  \malgun #1% Korean font switch
 }
}
```

It should be noted that this redefinition of `\spaceskip` for the current paragraph would also be applied for Latin text mingled within a Korean paragraph. Furthermore, this font selection process – yet without activating the Latin mode and adjusting the space width – could also be used in the case where distinct fonts for Chinese and Japanese text are required.

An illustration of the result of this paragraph typesetting is given in Figure 4. Before space width adjustment, the overfull hboxes materialised by the two black boxes should be noticed.

2.5 Sophisticated line-breaking

Just as, say, in French, where line breaks are not allowed before the punctuation marks ':', ';', '!' and so on – even though these need to be preceded by a space and are thus typical usages of non-breaking spaces – CJK typesetting expectedly forbids line breaks before punctuation marks such as commas and periods.



(a) (b)

Figure 5: Paragraph transformation by the original (a) and new (b) scanner macro: no more line break before a comma (Japanese text example).

We derive in this section a new scanner macro, `\cjk@scanbis`, to tackle this remaining problem. The approach is simple: refrain from adding extra space after the current character when the next one is a punctuation mark. And at the same time, this new scanner allows us to solve the aforementioned incongruity of extra space being added before a space character in Latin text paragraphs.

In practice, the new scanner takes two tokens as parameters instead of one: the first parameter is the currently processed token and the second one is the next token in line. The recursive call is also updated since now expecting two tokens as parameters instead of one; see below.

```
\def\cjk@scanbis#1#2{% two tokens passed
 #1%
 \ifx#2\cjk@stop
  \par
 \else
  \if#2、% no extra space before character '、'
  \else\if#2。% idem before character '。'
  \else\if#2\space% idem before a space
  \else\if#1\space% idem after a space
  \else\hskip 0pt plus 1pt minus 1pt\relax
  \fi\fi\fi\fi
  \expandafter\cjk@scanbis\expandafter#2%
 \fi
}
```

Note that similar additional conditions for other CJK punctuation marks can be appended if needed. Besides, in the scanner macro `\cjk@scanstart`, the expression `\cjk@scan#1\cjk@stop` is naturally changed to `\cjk@scanbis#1\cjk@stop`.

An illustration of the effect of this new scanner is shown in Figure 5.

3 State of the art and contribution

Early solutions for CJK support within the TEX ecosystem include the CJK package [5] and the

Japanese TEX system pTEX [7]. While the former provides some support for Unicode, the latter does not. Notably, the CJK package has partial support for vertical typesetting. Regarding Korean, the hlatex package [12] enables the processing by LATEX of KS X 1001 encoded files, and of UTF-8 files via the obsolete TEX extension Omega [9]. Omega also has some support for multi-directional CJK typesetting.

More recent alternatives include the xeCJK package [6], which is dedicated to XETEX (i.e., no LuaTEX support). This package is very large as it consists of more than 14,000 lines of macro code. As of 2018, it is only documented in Chinese. Another extensive package, LuaTEX-ja [11], is available, this time restricted to the support for Japanese with LuaTEX. Finally, upLATEX [8], another system dedicated to Japanese can also be cited; it is based on pLATEX, which is in turn based on pTEX.

Even if the above are more or less complete solutions to the CJK typesetting issue with TEX, we have presented in this paper a very simple solution, which neither requires a separate TEX system such as pTEX nor advanced TEX capacities such as xtemplate, LATEX3, etc., unlike, for instance, xeCJK. With only a few lines of macro code, we have described how to add basic yet arguably competent support for CJK to both XETEX and LuaTEX, indistinctly. The XETEX, LuaTEX flexibility has been retained: no extra layer has been piled as, for instance, with xeCJK (e.g., the `\setCJKmainfont` command). Moreover, the complexity induced by packages such as xeCJK is likely to be a threat to the compatibility with other packages, as well as with online compilation systems such as those employed by scientific publishers.

4 Conclusions

It is well known that the Chinese, Japanese and Korean writing systems are challenging for the typesetting solutions such as TEX that were originally designed for Latin text. Various extensions and packages were proposed to support CJK in TEX, with uneven success. Such solutions are in most cases, if not all, extensive – not to say invasive – additions to the TEX ecosystem. In this paper, relying on the Unicode-capable XETEX and LuaTEX systems, we have presented and pedagogically discussed a minimalistic solution to this CJK typesetting issue. With only a few lines of macro code, we have shown that satisfactory CJK support can be achieved: paragraph management, Latin text mingling and sophisticated line-breaking are examples of the addressed typesetting issues.

Regarding future works, given its still rather frequent usage, right-to-left horizontal typesetting would be a useful addition to this pedagogical discussion on CJK typesetting. Furthermore, even though a complex issue for TEX, right-to-left vertical typesetting is a meaningful objective as it is ubiquitous for the CJK writing systems.

Acknowledgements

The author is thankful towards Keiichi Kaneko (Tokyo University of Agriculture and Technology, Japan) and Takeyuki Nagao (Chiba University of Commerce, Japan) for their insightful advices.

References

[1] A. Bossard. *Chinese Characters, Deciphered*. Kanagawa University Press, Yokohama, Japan, 2018.

[2] S. Checkoway. *The everyhook package*, 11 2014. Package documentation. `https://ctan.org/pkg/everyhook` (last accessed November 2018).

[3] Google. Google Noto fonts, 2017. `https://google.com/get/noto` (last accessed December 2018).

[4] D. E. Knuth. *The TEXbook*. Addison-Wesley, Boston, MA, USA, 1986.

[5] W. Lemberg. *CJK*, 4 2015. Package documentation. `https://ctan.org/pkg/cjk` (last accessed November 2018).

[6] L. Liu and Q. Lee. *xeCJK 宏包 (in Chinese)*, 4 2018. Package documentation. `https://ctan.org/pkg/xecjk` (last accessed November 2018).

[7] K. Nakano, Japanese TEX Development Community, and TTK. *About pLATEX 2ε*, 9 2018. Package documentation. `https://ctan.org/pkg/platex` (last accessed November 2018).

[8] K. Nakano, Japanese TEX Development Community, and TTK. *About upLATEX 2ε*, 4 2018. Package documentation. `https://ctan.org/pkg/uplatex` (last accessed November 2018).

[9] J. Plaice and Y. Haralambous. The latest developments in Ω. *TUGboat* 17(2):181–183, 6 1996.

[10] W. Robertson. *The fontspec package – Font selection for XETLATEXand LuaLATEX*, 7 2018. Package documentation. `https://ctan.org/pkg/fontspec` (last accessed December 2018).

[11] The LuaTEX-ja project team. *The LuaTEX-ja package*, 11 2018. Package documentation. `https://ctan.org/pkg/luatexja` (last accessed November 2018).

[12] K. Un. 한글라텍 길잡이 *(in Korean)*, 4 2005. Package documentation. `https://ctan.org/pkg/hlatex` (last accessed December 2018).

[13] B. Veytsman. *Splitting Long Sequences of Letters (DNA, RNA, Proteins, Etc.)*, 8 2006. Package documentation. `https://ctan.org/pkg/seqsplit` (last accessed November 2018).

Appendix

The placeholder text used in the various illustrations of this article is in the public domain as detailed below. Figure 1: the placeholder text is the two first paragraphs of Article 8 of the Chinese constitution (1947), written with traditional Chinese. Figure 2: the placeholder text is the first paragraph of the Japanese constitution (1946), followed by the first few words of the corresponding official English translation. Figure 3: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946), followed by the corresponding official English translation. Figure 4: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946), followed by the first paragraph of Article 76 of the South Korean constitution (1988). Figure 5: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946).

◇ Antoine Bossard
Graduate School of Science
Kanagawa University
2946 Tsuchiya, Hiratsuka,
Kanagawa 259-1293, Japan
abossard@kanagawa-u.ac.jp

FreeType_MF_Module2: Integration of METAFONT, GF, and PK inside FreeType

Jaeyoung Choi, Saima Majeed, Ammar Ul Hassan and Geunho Jeong

Abstract

METAFONT is the structured font definition that has the ability to generate variants of different font styles by changing its parameter values. It doesn't require to create a new font file for every distinct font design. It generates the output fonts such as Generic Font (GF) and its relevant TeX Font Metric (TFM) file on demand. These fonts can be utilized on any size of the resolution devices without creating new font file according to the preferred size. However, METAFONT (MF), Generic Fonts (GF), and Packed Fonts (PK compressed form of GF) cannot be utilized beyond the TeX environment as it requires the additional conversion overhead. Furthermore, existing font engine such as FreeType doesn't support such fonts.

In this paper, we have proposed a module for FreeType which not only adds the support of METAFONT, but also adds the support of GF and PK font under Linux environment. The proposed module automatically perform such conversions without relying on other libraries. By using the proposed module, users can generate variants of font styles (by MF) and use it on the desired resolution devices (by GF). The proposed font module reduces the creation time and cost for creating the distinct fonts styles. Furthermore, it reduces the conversion and configuration overhead for TeX-oriented fonts.

1 Introduction

In the recent era, development in technology is increasing rapidly. In such environments, there is always a need of better and reliable medium for communication. Traditionally, fonts were used as means of communication. A font was collection of small pieces of metal which has particular size and style of the typeface. With the enhancement, modern fonts were introduced which were expected to sum up both the letter shape as it is presented on the metal and the ability of the typesetter by providing information that how to set position and replace the character as appropriate. Such technique was not reliable as the concept of pen and paper was considered the slow and inefficient way of communication. This traditional technique was replaced by the modern fonts. A new concept of digital systems aroused where these modern fonts are utilized which replaced the pen and paper usage. Therefore, modern font is implemented as digital data file which contains set of graphically related characters, symbols or glyphs.

The ability of science and technology to improve human life is known to us. With the rapid increase in development of science and technology, world is becoming "smart". People will automatically be served by smart devices. In such smart devices, digital fonts are commonly used than analog fonts. As font is the representation of text in a specific style and size, therefore, designers can use various font setting to give meaning to their ideas in text. Text is still considered the most appropriate and an elective source to communicate and gather information, respectively. Although a different styles of digital fonts have been created but still they do not meet the requirements of all the users and users cannot alter digital font styles easily [1]. A perfect application for the satisfaction of users' diversified requirements concerning font styles does not exist [2].

Currently, popular digital fonts, either bitmap or outline, have limits on changing font style [3]. These limitations are removed by another type of fonts such as parameterized fonts e.g. METAFONT which will be discussed later in depth. METAFONT provides the opportunity to the font designers to create a different font styles by just changing some of its parameter values. It generates the TeX-oriented bitmap font such as Generic Font (GF) and its equivalent TeX Font Metric (TFM) file. However, the usage of METAFONT directly in the digital environment is not easy as its specific to TeX oriented environment and the current font engines, the FreeType rasterizer doesn't support the METAFONT, Generic Font (GF), and Packed Font (PK). In order to use the METAFONT, GF, and PK font, users have to specifically convert them into its equivalent outline font. When METAFONT was created, the hardware of the PCs was not fast enough to perform the runtime conversion of METAFONT into outline font. Therefore, users are not able to get advantage from the METAFONT to get different font styles. Currently, the hardware which are being utilized in system are fast enough to perform such conversions on runtime. If such fonts will be supported by the current font engines, then the workload of the font designers will be reduced. As the font designers have to create a separate font file for every distinct style. Such recreation task is time taken especially in case of designing the CJK (Chinese-Japanese-Korean) characters due to its complex letters and shape. Therefore, such benefits given by METAFONT can be applied to the CJK font to produce high quality font in an efficient manner. Our previous work, FreeType_MF_Module[10] have been

accomplished for the direct usage of METAFONT excluding TeX-based bitmap fonts, inside FreeType rasterizer. But the work was somehow based on the external library such as mftrace during the internal conversion. Therefore, such library has disadvantages related to the performance and quality. Hence, the purpose of this research is to present a module inside the FreeType that will directly use the METAFONT, GF, and PK font in Linux environment.

In Section 2, the primary objective of this work is discussed. In Section 3, the METAFONT processing with its compiler/interpreter such as mf program are explained. In Section 4, the related research regarding the conversion of METAFONT is discussed along with their drawbacks. The implementation of the proposed module is discussed in Section 5. The experiments on the proposed module and performance evaluation along with other modules of FreeType rasterizer is presented in Section 6. Section 7, describes the concluding remarks.

2 Objective of the Research

With the enhancement in development and technology, typography also get the fame. The primary focus of this work is to understand the digital fonts, TeX-oriented bitmap fonts and find out the ways how to utilize it in Linux environment using the current font engines. Hence, the objective of this research is:

1. To save time of the designer to study the details of each font design from scratch and then create font file for every distinct design
2. To generate variants of different font styles using parametrized font such as METAFONT
3. To utilize the TeX-based bitmap fonts such as GF which is specific to TeX environment inside Freetype font engine
4. To increase the performance by using compact form of GF such as Packed Font (PK)
5. To set the automatic magnification and resolution according to the display in case of Generic Font

3 METAFONT processing with mf program

METAFONT, a TEX font system, had been introduced by D. E. Knuth [4] is an organized font definition which allows the font designers to change the style of font as per their requirements by changing values of parameters. METAFONT benefits the user in a way that they don't need to write the different font file for every unique style. It is considered as programming language which contain lines and curves drawing guidelines which are later interpreted by the interpreter/compiler of METAFONT such as mf program to draw the glyphs into a bitmaps and keeping the bitmaps into a file when done. Mf program determines the exact shapes by solving mathematical equations imposed by METAFONT. To process the METAFONT using mf program, users must have the knowledge of mf invocations [5]. Figure 1 shows the proper way of processing the METAFONT using mf program. It can accept plenty of other commands. Therefore, in order to get the correct GF file, these commands are provided e.g. mode, mag, and METAFONT file to process. The mode command specify the printed mode, if leave this out the default will be used such as proof mode where METAFONT will outputs at a resolution of 2602dpi; this is not usually required without TFM. The mag command takes the font resolution in pixels per inch along with the METAFONT file. In result, mf program generates the TeX-oriented bitmap font file such as GF, its relevant Font metric file named: TFM, and log file.

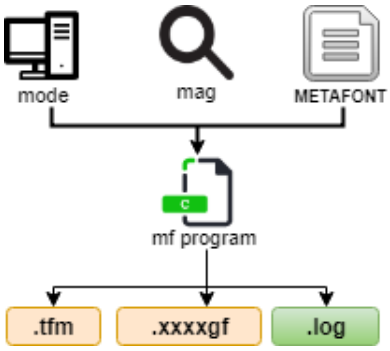


Figure 1: mf invocations

For example, if the device is 600dpi and specify the magnification 3 along with mode then mf program will perform calculations internally and will generate the output in the form of GF at 1800dpi, along with its corresponding tfm and log file.

Generic Font (GF) is TeX-oriented bitmap font generated by the mf program by taking METAFONT as an input along with other information related to the output device. GF font files are generated for each output device with specific scaled sizes. Such font files contain the character shapes in a bitmap form. However, the information relevant to the characters shape are stored in the TeX font metric (TFM) file. To give meaning to the GF font, its corresponding TFM is required as TeX only reads the font metric file instead of GF. These fonts are utilized in TeX typesetting systems. To view or print, these fonts are converted into device-independent (.dvi) files. Later, DVI drivers are required to interpret the data given in device independent files as .dvi files

cannot be read directly by the TeX. Such conversions are performed by the utility named; gftodvi. It reads binary generic font and convert them into device-independent files. In order to preview, utility named xdvi is being utilized. As GF files are unreadable, therefore, such conversions are required in order to view.

The Packed Font (PK) is bitmap font format utilized in the TeX typesetting systems. It can be obtained by compressing the GF font. As GF files are larger in size, therefore, the size of the PK is half of their GF counterparts. The content stored in PK files are same as GF. Such file format is intended to be easy to read and interpreted by the device drivers. It reduces the overhead of loading the font on memory. Due to its compression nature, it reduces the memory requirements for those drivers that loads and stores the each font file on memory. They are also easier to convert into a raster representation. (This also makes it possible for a driver to skip a particular character quickly if it knows that the character is unused).

4 Related Works

4.1 Existing Font Systems

VFlib [6] is a virtual font system that can handle the variety of font formats e.g. TrueType, Type1, and TeX-bitmap fonts. It handles the library itself and the database font file where it defines the implicit and explicit fonts. Although it supports different font formats but for some fonts it make use of the external libraries, as shown in Figure 2. Furthermore, it doesn't support the METAFONT but it has the ability to handle the TeX-bitmap fonts. The font searching mechanism utilized in VFlib is time consuming, if the font doesn't appear in the database. Therefore, to handle such fonts, various font drivers will be called to check whether the requested font can be opened or not. Hence, such font systems are not suitable to add the METAFONT support because of reliance and taking care of database.

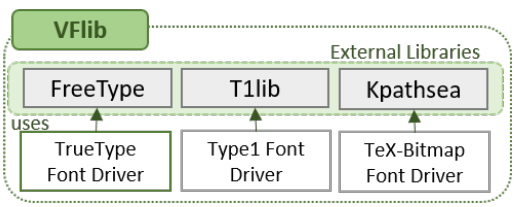


Figure 2: VFlib Reliance

An alternative to such font engines is FreeType [7] font rasterizer. It has the ability to handle different font styles regardless of platform dependency unlike T1lib [8] font rasterizer. However, it doesn't support the TeX-oriented bitmap fonts and METAFONT. But it provides the intuitive interfaces which allows the end-users to add the new font module to enhance the functionality of the engine. Therefore, selection of the FreeType font engine is the best choice for adding the TeX-oriented bitmap fonts because it has no dependency and database issues. If there is a module inside Freetype which will support the TeX-oriented bitmap fonts such as GF and PK, then, users can get advantage of such fonts that are only specific to TeX-environment. No pre-conversion by utilizing the DVI drivers will be required to preview TeX-oriented fonts.

4.2 Researches on adding METAFONT support in existing font systems

As mentioned in Section 4.1, FreeType font engine provides the capability to add the new font module. MFCONFIG [2] added an indirect support of METAFONT inside FreeType. It provides an intuitive way to use METAFONT on Linux environment. As shown in Figure 3, it allows the users to utilize the METAFONT but it has some dependency problem as it is built on high-level font libraries such as FONTCONFIG [9] and Xft. Due to such dependencies it affects the performance of the module compared to font driver modules of FreeType. It is unable to handle the TeX-oriented bitmap fonts such as GF and PK. Therefore, adding the functionality of TeX-bitmap fonts is inadequate as it's not directly implemented inside Freetype.

FreeType_MF_Module [10], a METAFONT module inside the FreeType font engine resolves the dependency and performance issues which were stimulated in MFCONFIG. Its performance is relatively faster than MFCONFIG as it is implemented inside the FreeType. In order to use the METAFONT, it requires to transform it into outline font. Hence, FreeType_MF_Module performs such conversions but relying on mftrace. Although, it generates a high-quality output but during conversion font file information is vanished due to reliance on mftrace. As shown in Figure 4, when the request of METAFONT is received by the FreeType, it sends it to FreeType_MF_Module. When it comes to its sub-module named: Transformation Module, it utilizes mftrace. Mftrace has its own drawbacks. It was specifically designed for translating METAFONT fonts to Type1 or TrueType formats by internally utilizing the autotrace and potrace libraries for vectorization purpose. Approximate conversion gives approximate outline and lost information about nodes and other control points [11]. Although, it processes

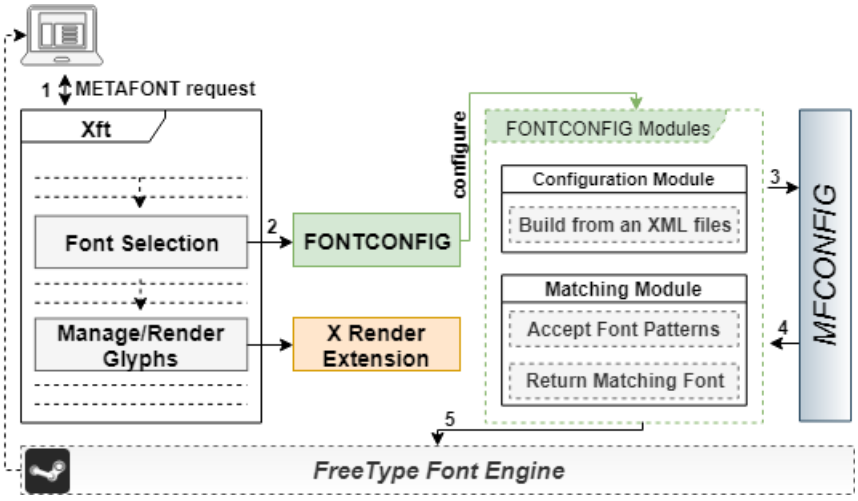


Figure 3: MFCONFIG Internal Architecture

the METAFONT but is unable to process TeX-based bitmap fonts such as Packed Font (PK) and Generic Fonts (GF). Therefore, to add a support of GF or PK inside FreeType_MF_Module is inconvenient due to dependency on external library which slower down the performance of the module.

The proposed FreeType_MF_Module2 intends to resolve the problems of FreeType_MF_Module, and is able to support TeX-bitmap fonts along with METAFONT. The module can process METAFONT and GF independently without relying on external library e.g. mftrace. It can be easily installed and removed, as it is implemented just like the default FreeType driver module. Therefore, METAFONT and TeX-oriented bitmap fonts can be used as the existing digital font formats using the proposed module.

5 Implementation of the Module

To use the digital fonts, FreeType is a powerful library to render the text on screen. It is capable of producing the high quality glyph images of the bitmap and outline font formats. When FreeType receives a request of font from the client application, it sends the font file to the responsible module driver for the manipulation. Otherwise, it displays an error message to the client if the requested font file is not supported. Similarly, the proposed module is directly installed inside the FreeType to process the request of METAFONT and TeX bitmap font such as Generic Font (GF) and Packed Font (PK). As shown in Figure 5, when FreeType receives the METAFONT or GF request it directs into FreeType_MF_Module2.

5.1 METAFONT (MF) Request

When FreeType sends the METAFONT request to FreeType_MF_Module2, its submodule Request Analyzer API analyzes the font file. It analyzes that the requested font file is the exact METAFONT file or the wrong one by analyzing its style parameters. After analyzing, it checks whether the requested font is already manipulated by the font driver or the new request is arrived via Cache. If the requested font is found in the Cache, it sends directly to the engine for manipulation. But if the font is not found in the Cache, it sends the METAFONT request to the Conversion Module. After receiving the request, it utilizes its submodule named: Script Handler. The core functionality of the module is performed in this module. It calls the scripting module based on the request. On METAFONT request, it calls the MF Script module by passing the METAFONT file.

As shown in Figure 6, MF Script Module calls its submodule named: Font Style Extractor Module. It extract the font style parameters from the METAFONT file. For example, the METAFONT request given to the module with the italic style, this will extract the italic style parameters from the METAFONT file and apply into it. Once it extracts the font style parameters, its corresponding outline will be generated with the requested style by utilizing Vectorization Module. After extracting the characters outline, it is necessary to remove the redundant nodes from the characters shapes to make the better quality. Therefore, Node Redundancy Analysis will receive the transformed METAFONT and analyze the outline contours and remove the redundant nodes from the font to create the simplified outline. Once simplification task is done, auto-hinting will be

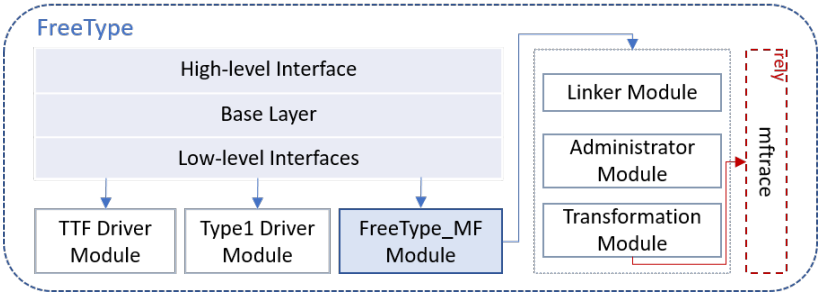


Figure 4: FreeType_MF_Module Architecture

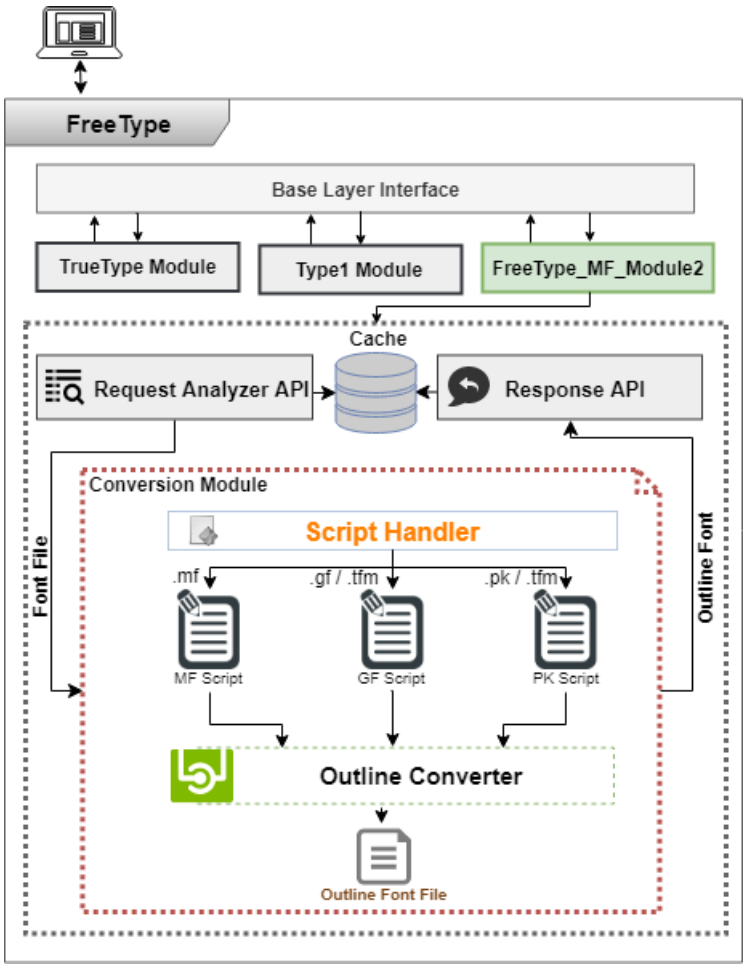


Figure 5: FreeType_MF_Module2 Architecture

performed on the font using Hinting Module. After hinting, the corresponding outline font will be generated with the Outline Converter module and sends the outline font file to the module named: Response API. It updates the Cache with the newly generated outline font for reusability and high performance. After updating, FreeType renders this outline font that was created from the METAFONT with the requested style parameter values.

5.2 Generic Font (GF) Request

When FreeType sends the GF request to the proposed module, it sends the requested font to the Request Analyzer API module. It checks whether the requested GF font is converted with correct use of mf compiler or not by analyzing the device specific information. If the requested GF file is not generated by the correct use of mf compiler, then Request Analyzer API module will not proceed as it has to

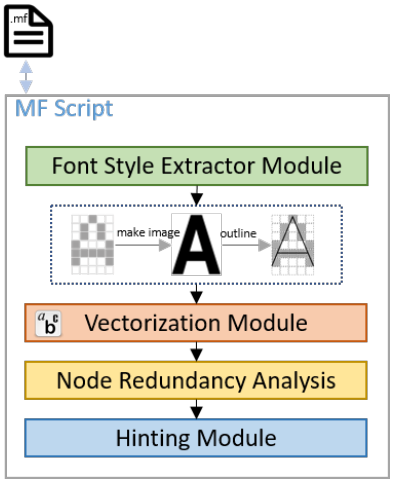


Figure 6: MF Script Internal Architecture

compute file name by using font parameters such as device resolution and magnification. But if the GF font is generated by the correct use of mf compiler, then its TeX font metric file must exist.

On GF request, its TFM must be provided for internal computation related to character shapes. Furthermore, TeX only reads the TFM instead of GF as all the font relevant information is provided by the TFM. Once Request Analyzer API module analyzes the GF request, then it checks in the Cache to get the manipulated font if exist. If requested font doesn't exist in the Cache, then the request will be forwarded to the Conversion Module where its submodule named: Script Handler handles the GF request along with its relevant tfm file.As shown in Figure 7, when GF Script receives the GF file, its submodule Extractor Module plays the main functionality. Its internal module of Font Info Extractor extracts the font related information from the TeX font metric file and extracts a sequence of bitmaps at a specified resolution from GF file.

After extraction, it merges the extracted information and gives meaning to the unreadable gf file in the form of characters images via Merge Extracted Info Module. From such bitmap relevant font, it makes character images. After merging and creating the vectorize kind of images, it extracts the outline of the characters via Outline Extractor Module. After extracting the outline, it sends the extracted outlined characters to the Simplify Module, which is capable of analyzing the font and removes the redundant nodes from the font in order to make the good quality outline. As a result, it outputs the simplified outline using the Outline Converter module internally. The newly created outline font is sent to the Response API, which updates the Cache with the generated outline font for later reusability. Once Cache updated, it sends back the response to the core FreeType module for further processing. Lastly, FreeType renders this outline font that was made from the requested GF with the styled parameter values at a specified resolution.

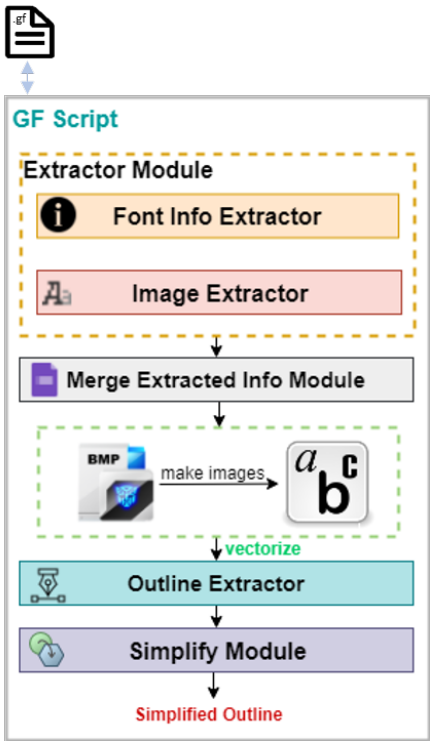


Figure 7: GF Script Internal Architecture

5.3 Packed Font (PK) Request

On PK font request, FreeType performs the same functionality till Conversion Module as it performs in Sections 5.1 and 5.2. Once Script Handler receives the requested PK font, it utilizes PK Script. As shown in Figure 8, Extractor Module extracts the raster information from the packed file. It internally utilizes the GF Script for extracting the font information from the relevant tfm file using it submodule Font Info Extractor. After extraction, it performs the autotracing on the merged font via Autotracing Module, which outputs the character images. Once done, it sends the transformed output to the Outline Extractor Module where it obtains the outline of the characters. After getting the outlined character images, it performs the outline contour analysis and remove the nodes redundancy from the outlines using the submodule named: Outline Contour Analysis Module. It sends the simplified output to the Outline Converter which creates the good quality outline font

file. The generated outline font file is send to the Response API which updates the Cache and sends to the corresponding FreeType module for rendering.

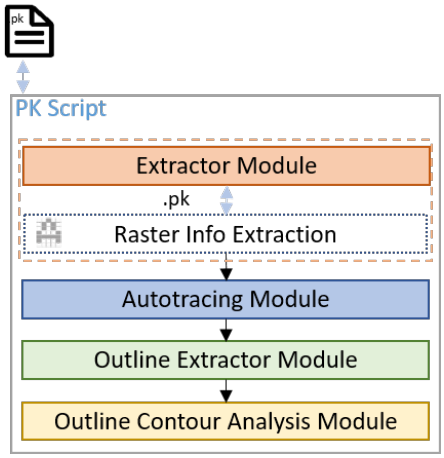


Figure 8: PK Script Internal Architecture

The proposed module provides the direct support of METAFONT, GF, and PK. It is perfectly compatible with the default module drivers of the FreeType. It can manipulate the request with the desired style parameters and scale size. In result, it provides the better quality outline font without utilizing the external libraries.

6 Experiments and Performance Evaluation

In order to test the proposed module, an application server is being utilized. The application server is responsible for rendering the text on the screen by taking the font file from the FreeType along with the requested text to be printed. FreeType can only process those fonts which are supported by it. When the client application sends the METAFONT, GF, or PK request to the FreeType, it internally processes the requested font using the proposed module and sends the newly generated outline font file along with the input text to the application server to display it on screen.

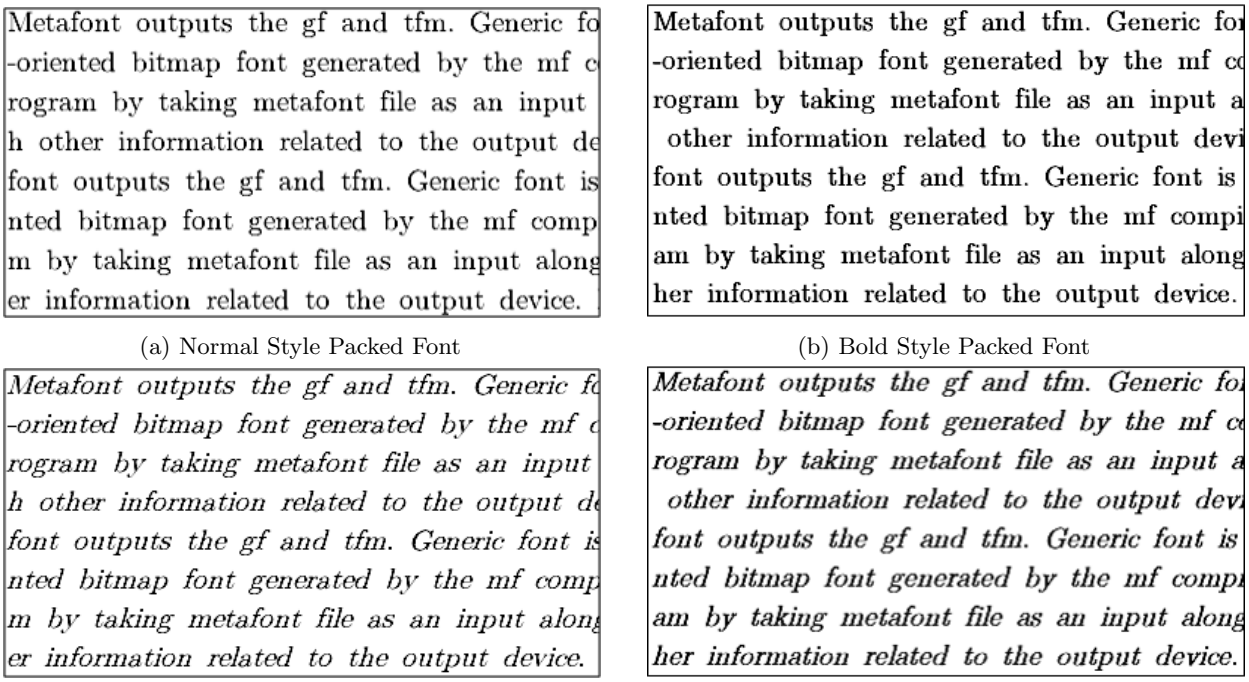
For testing purpose, the METAFONT font Computer Modern is used. The Computer Modern fonts are examined with the four unique styles: Normal, Italic, Bold, and Bold+Italic. These styles are generated by tweaking the METAFONT parameters. In order to verify the quality of the proposed module results, authors used the same four styles of another font family named: FreeSerif. The sample text comprises of words and characters, including the space characters.

The same font family is utilized to test the FreeType_MF_Module with the same four font styles. Changing the parameter values and generating new styles are explained in [10]. The same concept is applied on the proposed module for experiments. The only difference comes in case of GF and PK fonts. In order to manipulate such fonts, information of the printer device and font resolutions of the specific device is required. Furthermore, such TeX-oriented bitmap fonts cannot be directly viewed on the screen, it requires DVI drivers which makes proof sheets from a GF bitmap file where characters from the gf appear one per page in the form of .dvi file. Therefore, in the proposed module the GF and PK fonts are directly manipulated by the module without requiring the DVI drivers and previewers. It accepts the input text by the client application and internally calculates the font resolution in pixels per inch. Afterwards, it internally processes the GF and PK file as described in Sections 5.2 and 5.3, and generates the resultant output with the desired style.

When FreeType sends the METAFONT request to the proposed module, it internally manipulates the request by extracting the styled parameters from the source file. Default style of Computer Modern METAFONT is generated by extracting the default parameters. The four font styles such as Normal, Bold, Italic, and Bold+Italic are generated by the module, and it generates the similar output in Figure. 9(a), (b), (c), (d). Using one Computer Modern METAFONT file, user can generate different font styles based on the desire and requirement.

When FreeType receives the Generic Font request by the client application server, it sends it to the proposed module along with the input text, where it extracts the font related information from the TFM file and resolution information from the GF file. After that it internally calculates the font resolution in pixels per inch by referring to a device definition. Later, it generates the output on the resulted font resolution, as similar to as shown in Figure 9. The default style of Generic Font is generated by extracting the default style parameters at 1200dpi. The remaining font styles such as Bold, Italic, and Bold+Italic are generated by the module at the calculated resolution similar to the results in Figure 9(b), (c), (d). The GF results differ slightly due to the variations in resolution than METAFONT. The authors tested the GF font with different magnifications at the time of manipulation.

Once GF font is obtained by the METAFONT, it has a larger size which takes a lot of memory during the manipulation. In order to reduce the memory consumption, it's converted into packed form using



(a) Normal Style Packed Font

(b) Bold Style Packed Font

(c) Italic Style Packed Font

(d) Bold-Italic Style Packed Font

Figure 9: Text printed with Packed Font (PK)

Table 1: Average time of Rendering (in milliseconds)

Style	Time efficiency of font modules (Average Time)			
	FreeType_MF_Module	*FreeType_MF_Module2*		
		METAFONT	GF	PK
Normal	6 ms	5 ms	6 ms	4 ms
Bold	7 ms	6 ms	6 ms	6 ms
Italic	6 ms	6 ms	5 ms	4 ms
Bold+Italic	8 ms	8 ms	7 ms	6 ms

the utility gftopk. It contains the same information and style parameters which were utilized at the time of GF experiment. Therefore, their resultant output only differs at the performance level rather than quality. The resultant output for the PK request is similar like GF at the same font resolution. As shown in Figure 9,font styles such as Normal, Bold, Italic, and Bold+Italic are generated by the module. The authors compared the obtained results with the FreeType_MF_Module. Therefore, it is concluded that the results are quite similar and proposed module handles the TeX-oriented bitmaps fonts along with the METAFONT inside the FreeType without reliance related to the conversions.

The authors have not only considered the quality factor of the generated font using the proposed module, but also the performance factor. As shown in Table 1, the performance of FreeType_MF_Module is relatively slower on processing the Bold and Bold+Italic font style of METAFONT. It takes time due to the dependency on the external library such as mftrace. Therefore, the proposed module overcomes such performance and dependency issues and added the dual functionality by integrating the TeX-oriented fonts. The GF font takes a little more time compared to PK font but less time than METAFONT font as it is already in the compiled form. The PK font takes a less time than METAFONT and GF, as it is the compressed and compiled form of GF.

The proposed FreeType_MF_Module2 provides the parameterized font support to the users. The proposed module doesn't require the preconversion before giving it to the FreeType rasterizer. The client applications which utilizes the FreeType internally can utilize the METAFONT and TeX-oriented bitmap fonts such as GF and PK using the proposed module. Users can utilize such fonts as it utilizes the TrueType fonts using the FreeType. The proposed module can be utilized in the FreeType font engine as a default driver module. The proposed module will work the same as the other driver modules works in the FreeType. It is able to support the real time conversions on a modern Linux environment.

7 Conclusion

In this paper, a module is proposed for the FreeType font rasterizer which enhanced its functionality by adding the parameterized and TeX-oriented bitmap fonts. FreeType supports many different font formats but doesn't support the fonts which are utilized only in the TeX environment such as GF and PK. It is unable to support the parametrized font such as METAFONT. Although the recent studies provided a way to utilize the METAFONT inside FreeType, however, it has the dependency issues which effects the performance of the module. Furthermore, it can only handle the METAFONT request. Therefore, the proposed module overcome these issues and added the TeX-oriented bitmap support as well. Using the proposed module, users can use the METAFONT, GF, and PK fonts without using other drivers for conversion purpose. Such fonts are specific to the TeX environment, therefore, using the proposed module users can utilize these fonts outside the TeX environment.

Furthermore, the proposed module overcome the disadvantages of the outline fonts which limits the users to change the font style using the existing font file. It requires to create the different font file for every distinct font style with the different sizes as well. Therefore, for creating a new font style in outline fonts for the CJK fonts consumes time and cost, as these are complicated in shapes as compared to the alphabet-based fonts. A various studies have been conducted to implement the CJK fonts, such as Hongzi[14], including the use of a structural font generator using METAFONT for Korean and Chinese[15]. It might take a longer time to process CJK METAFONT fonts, which have complicated shapes and have more than several thousands of phonemes. The proposed module optimization and utilization for the CJK fonts will be considered in future.

References

[1] Donald E. Knuth, *S. Song. Development of Korea Typography Industry,* Appreciating Korean Language, 2013.

[2] Jaeyoung Choi, Sungmin Kim, Hojin Lee, Geunho Jeong, *MFCONFIG:* METAFONT plugin module for Freetype rasterizer TUG 2016 (TUGboat, 2016): 163170.

[3] Y. Park., *Current status of Hangeul in 21th century.* Type and Typography magazine The T, 7th.

[4] Donald E.Knuth, *Computers and typesetting.* Volume c: The Metafontbook. TUGboat, 1986.

[5] Web2c: *A TeX implementation.* http://tug.org/texinfohtml/web2c.html

[6] H. Kakugawa, M. Nishikimi, N. Takahashi, S. Tomura, and K. Handa. *A general purpose font module for multilingual application programs.* Software: Practice and Experience, 31(15):1487–1508, 2001. dx.doi.org/10.1002/spe.424

[7] David Turner, Robert Wilhelm, Werner Lemberg, *FreeType,* www.freetype.org.

[8] Rainer Menzner, *A library for generating character bitmaps from Adobe Type 1 fonts.* http://www.fifi.org/doc/t1lib-dev/t1lib_doc.pdf.gz

[9] Donald E.Knuth Metafont: The Program. Addison-Wesley, 1986.K. Packard, *Fontconfig,* Gnome User's and Developers European 2002.

[10] Jaeyoung Choi, Ammar Ul Hassan, Geunho Jeong, *FreeType_MF_Module,* 2016. https://tug.org/tug2018/preprints/choi-freetype.pdf

[11] Scalable Fonts for MetaFont, *mftrace* http://lilypond.org/mftrace/

[12] Autotrace library. http://lilypond.org/mftrace/

[13] Karel Piska, *Creating Type 1 Fonts from METAFONT Sources, Comparison of Tools, Techniques and Results,* 2004.

[14] Javier Rodr'ıguez Laguna: *Hong-Zi: A Chinese METAFONT,* Communications of the TEX Users Group, TUGboat, Vol 26, No.2 pp.125-141,2005.

[15] Jaeyoung Choi, Gyeongjae Gwon, Minju Son, Geunho Jeong, *"Next Generation CJK Font Technology Using the Metafont",* LetterSeed 15, pp.87-101, Korea Society of Typography, 2017. 6.

MacTEX-2019, notification, and hardened runtimes

Richard Koch

Abstract

MacTEX installs everything needed to run TEX on a Macintosh, including TEX Live, Ghostscript, and four GUI applications: TeXShop, TEX Live Utility, LATEXiT, and BibDesk. In macOS 10.15, Catalina, Apple requires that install packages be notarized, and all command line and GUI applications in such a package must be signed and adopt a hardened runtime. I'll explain what this means and how it was accomplished.

MacTEX 2019

1 Recent changes

For many years, MacTEX supported macOS 10.5 (Leopard) and higher, on both PowerPC and Intel processors. Starting in 2017, we decided to limit support to those systems for which Apple still provides security updates. Consequently, we support the three latest systems; in 2019 we support Sierra, High Sierra, and Mojave (that is, 10.12 and higher). Each fall, Apple introduces a new system and we also support that. Thus MacTEX-2019 will support Catalina when that is released this fall.

Mojca Miklavec compiles Mac binaries for older systems; in 2019 she supports Snow Leopard (10.6) and higher. TEX Live contains both our binaries and Miklavec's binaries. Our web pages (`tug.org/mactex`) explain how to install TEX Live using either the MacTEX installer or the standard Unix install script (`install-tl`), so users with older systems can update using the Unix install script. Both methods produce exactly the same TEX Live in the end.

2 Security

I retired from the University of Oregon in 2002. In that year, freshmen arriving at the University discovered a CD and instruction sheet taped over the ethernet jacks in their dorm rooms. The sheet said

```
Warning: You must install the virus checker on
this CD before connecting your computer to the
ethernet. If you fail to follow this instruction,
you will lose ethernet privileges in this room.
```

The note ended with one more sentence:

```
Macintosh users can ignore this message.
```

But that was 2002. This April, I got the following:

```
From: koch@math.uoregon.edu
Date: April 4, 2019
To: koch@math.uoregon.edu

Hey! I compromised your account and gained full
access to it. I just sent this email from your
account. You visited an adult website and got
infected. This gave me access to all of your
contacts, browsing history, your passwords,
your webcam, and even your microphone.

I noticed you were trying to please yourself by
watching one of those nasty videos, well my son,
I recorded your actions ... (thanks to your
webcam) and even recorded your screen (the video
you were watching). Now, if you do nothing, then
I will send this video to all of your email,
social media and messenger contacts. You have
the option to prevent me from doing all of this.
All you need to do is to make the transfer of
\$958 to my bitcoin address ...
```

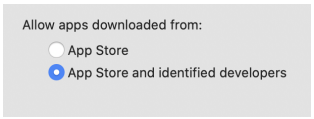
3 Lessons

- The Macintosh is built on top of Unix. Unix has strong protection against *other irresponsible* users. Like most companies, Apple has security engineers patching kernel and system bugs as they are found.
- But Macs are generally used by one person, and the remaining problem is to protect that person against himself or herself. If my Mac is attacked, I'm not worried that the criminal will become root. I'm worried that he will activate my camera, read my mail, find my contact list, or turn on my microphone.
- For several years, Apple has provided a (mandatory) solution for applications in the App Store. It is known as *sandboxing*. A sandboxed application cannot interact with other programs; it runs in its own sandbox.
- In Catalina (and also to some extent in Mojave) Apple provides a different kind of security protection for other programs. Unlike sandboxing, the new security is carefully tuned to allow any program to run as usual. Here's how it works.

4 Signing

This step was introduced in 2012. Apple Developers can *sign* their applications and their install packages. When software is downloaded from the Internet, the system checks that the software has not been modified since it was signed, and that the signature is from a known developer. It refuses to run software that doesn't pass. Otherwise it sets a Finder bit to disable future checks and runs the software. A

control panel in Apple's System Preferences controls this behavior:



Signing requires developer status from Apple, which costs $100 a year. TeXShop and MacTEX have always been signed.

Apple issues *two* developer signing certificates, one for applications and one for install packages. Signing applications is done in XCode as part of the build process. A command line binary signs install packages.

Tricks explained on the Internet allow users to disable the signing requirement and install any program. At this year's WWDC, Apple said that such tricks would *always* be available.

5 Notarization

This spring, Apple added notarization. This works like signing; both applications and install packages can be notarized. Once software is signed and just before release, it is sent to Apple. There it is checked for viruses (no human hands touch the software). Checking takes around 15 minutes. If the software passes the test, a "certificate" is mailed back and "stapled" to the software. In Catalina, software downloaded from the Internet must be both signed and notarized before it can run.

Previously, software was only tested once to make sure it was not modified. Now these tests will be rerun periodically. The details are somewhat vague (to me), so don't ask.

6 Hardened runtimes

Signing and notarization are small potatoes. The big security step in Catalina is the requirement that all applications and command line programs in a notarized install package must be signed and timestamped, and must adopt a Hardened Runtime. All of this is new. The MacTEX install package has been signed since 2012, but the individual TEX binaries are not signed. And while TeXShop is signed, the remaining applications TEX Live Utility, LATEXiT, and BibDesk are not signed. The kicker, however, is that these applications *and all command line apps* must adopt a hardened runtime. What is that?

Apple has a list of 13 dangerous operations a program might try to perform. I'll give the full list later, but among the items are these: accessing the camera, accessing the microphone, accessing location information, accessing the address book, accessing the user's calendars, accessing photos, sending Apple events to other applications, executing JIT-compiled code, loading third party libraries not by Apple. If an application adopts a hardened runtime, it is not allowed to perform any of these operations.

However, for each of the 13 dangerous operations, a developer can claim an *entitlement*. I have always dreamed of a TEX editor attached to a camera; to make a commutative diagram, draw it and take a picture and the editor converts the drawing into TEX. The author of such an editor would file an entitlement for the camera operation.

Nobody at Apple checks the entitlement list; there is no "approval process". A developer can claim all 13 entitlements and then the hardened runtime has no effect.

So calm down that case of paranoia. Apple isn't restricting developers. It is providing a tool to help open source developers improve security.

6.1 Dealing with command line programs

Command line programs can adopt a hardened runtime without recompiling. The command below does this for the `xz` binary used by `tlmgr`. The `--force` option says to replace any previous signing by the new one, and `--options=runtime` says to adopt a hardened runtime with no exceptions.

```
codesign \
 -s "Developer ID Application: Richard Koch" \
  --force --timestamp  --options=runtime xz
```

To claim exceptions for a command line program, add a flag `--entitlements=TUG.entitlement` to the previous call, where `TUG.entitlement` can be any name and is a short XML file. The example `TUG.entitlement` here allows linking with third party libraries. (One long line has been broken for *TUGboat* with a \; it should not be broken in a real file.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
 "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.security.cs.\
disable-library-validation</key>
  <true/>
</dict>
</plist>
```

By embedding the `codesign` call in a shell script, it is easy to construct scripts which sign, timestamp, and adopt hardened runtimes for all command line binaries in an install package.

6.2 Case 1: BasicTEX

In addition to the full MacTEX, we provide a smaller install package called BasicTeX, which installs the distribution obtained by using `install-tl` with the "small" scheme. To test the above ideas, I submitted this package unmodified to Apple for notarization. Apple refused to notarize it, but they sent back a detailed and easy-to-read error sheet. The bin directory of BasicTEX has 88 items. Apple ignored symbolic links, scripts, and other files, but had problems with 30 commands. These were exactly the commands which the Unix command `file` listed as "Mach-O 64-bit executable x86_64".

In addition, Apple found three other such binaries in `tlpkg/installer`: `lz4`, `wget`, `xz`.

I used the codesign script on these 33 binaries and submitted BasicTEX again to Apple for notarization. Approved!

6.3 Case 2: Ghostscript

Ghostscript only has two binaries, `gs-X11` with X11 support and `gs-noX11` without X. We install a symbolic link named `gs` to the appropriate binary.

I ran `codesign` on `gs-X11` and `gs-noX11` and submitted to Apple. Apple notarized the install package. But when the package was used to install Ghostscript, `gs` refused to run. Why?

Originally, Apple supplied an optional install package for X11. But their package was often out of date, so a mutual decision was made for a third party to supply X11 for the Macintosh as open source. Consequently, `gs-X11` links in a third party library, which is not allowed for hardened runtimes. Resigning `gs-X11` and claiming an entitlement for such linking solved the problem.

6.4 Case 3: `biber`

The `biber` binary is so complicated that TEX Live builders do not compile it. Instead the author submits binaries. The `codesign` script didn't work with this binary. I contacted the author, Philip Kime. A month later he sent a binary which worked. I suspect Kime knows a lot more about notarization than I do now.

6.5 Case 4: The big enchilada

Finally it was time to notarize the full TEX Live. I hardened `xz`, `wget`, `lz4`, and all the binaries in `bin/x86_64-darwin` which were not links and reported to be "Mach-0 64-bit executables' by `file`. Tests revealed that two of these binaries needed an exception for X11: `mf` and `xdvi-xaw`. I submitted the package to Apple. It was rejected.

A big difference between BasicTEX and the full TEX Live is that the second package has documentation provided by package makers. This documentation comes in a wide variety of formats: source files for illustrations, zip files, and so forth. When Apple tests an install package for viruses, does it unzip files and look inside? Yes, it does. Does it examine illustration source files? Yes, it does that too. So lots of things could go wrong.

Luckily, Apple provided clear explanations for rejection, and it turned out that MacTEX had only three problems:

- In `texmf-dist/doc/support/ctan-o-mat`, one file is given an extension `.pkg`. Apple believes that a file with extension `.pkg` is an install package, and this package was not signed. It turned out to be an ordinary text file.
- In `texmf-dist/doc/latex/codepage`, Apple could not unzip the file `demo.zip`.
- In `texmf-dist/source/latex/stellenbosch`, there is a zip file named `USlogos-4.0-src.zip` containing two CorelDraw source files for illustrations. Apple did not recognize these source files and flagged them.

The three problems were easy to work around. Bug reports were also sent to Apple so they can improve the notarization machinery.

7 Status of notarization for MacTEX-2019

Fully notarized install packages for MacTEX-2019, BasicTEX-2019, and Ghostscript-9.27 are available on the web for testing. Indeed, the Ghostscript-9.27 package on CTAN is already notarized. The MacTEX-2019 and BasicTEX-2019 packages will be moved to CTAN, replacing the original packages, in late summer just before Catalina is released.

TEX Live Utility, LATEXiT, and BibDesk are not in the notarized MacTEX-2019 because they are applications rather than command line programs, *so their authors must sign and notarize them.* This has not yet happened. If these authors used the XCode which comes with Mojave, these steps would be trivial, but they use an older XCode. We are working with the authors but have nothing to report.

8 Technical details

I end with some technical details for others who may need to deal with these issues on the Macintosh. I'll explain how to sign install packages and how to notarize such packages. Then I'll list the six runtime entitlements and seven resource access entitlements from an official Apple document.

8.1 Signing an install package

Signing requires developer status from Apple, which costs $100 a year. Certificate information and security codes are kept on Apple's KeyChain, and automatically retrieved by the signing software when needed. If you buy a new machine or install a new system, you must transfer this information to the new system. XCode makes this easy *if* you know what mysterious icon to click.

Signing applications happens automatically in XCode as part of the build process. Signing install packages is done on the command line. The command here signs `Temp.pkg` and writes the signed package `Basic.pkg`.

```
productsign  \
 --sign "Developer ID Installer: Richard Koch" \
 Temp.pkg Basic.pkg
```

8.2 Notarizing an install package

Notarization of install packages is done on the command line, and is somewhat trickier. Below are the crucial commands. The first command sends an install package to Apple to be notarized. If uploading succeeds, this command returns an identifier which I symbolize with `YYYY`; it is actually much longer.

```
xcrun altool --notarize-app \
  --primary-bundle-id \
       "org.tug.mactex.basictex" \
 --username "koch@uoregon.edu" \
 --password "XXXX" \
 --file BasicTeX.pkg
```

When Apple is finished, it sends a brief email stating whether notarization was successful. If there were errors, this second command asks for a detailed list of errors. The command returns a url, and the error list will then appear in a browser pointed to this url.

```
xcrun altool --notarization-info YYYY \
  --username  "koch@uoregon.edu" \
  --password "XXXX"
```

If notarization was successful, this third command staples the certificate to the install package, producing a notarized package:

```
xcrun stapler staple "BasicTeX.pkg"
```

In these commands, `altool` is a command line tool which communicates with Apple. This communication is normally protected using two-factor authentication, but that is not convenient for command line work. So before using `altool`, Apple asks developers to log into their account and give `altool` a temporary password. The symbol `XXXX` in the first and second commands represents this password.

The value `org.tug.mactex.basictex` in the first command identifies the install package for the notification process, but need not correspond to any similar string in the package. So the identifier can be selected randomly.

8.3 Runtime entitlements

All entitlements are boolean values; all keys start with `com.apple.security`, not shown here for brevity.

Allow Execution of JIT-compiled Code: whether the app may create writable and executable memory using the `MAP_JIT` flag. Key: `.cs.allow-jit`

Allow Unsigned Executable Memory: whether the app may create writable and executable memory without using the `MAP_JIT` flag. Key: `.cs.allow-unsigned-executable-memory`

Allow DYLD Environment Variables: whether the app may be impacted by DYLD environment variables, which can be used to inject code into the process. Key: `.cs.allow-dyld-environment-variables`

Disable Library Validation: whether the app may load plug-ins or frameworks signed by other developers. Key: `.cs.disable-library-validation`

Disable Executable Memory Protection: whether to disable code signing protections while launching the app. Key: `.cs.disable-executable-page-protection`

Debugging Tool: whether the app is a debugger and may attach to other processes or get task ports. Key: `.cs.debugger`

8.4 Resource access entitlements

Audio Input: whether the app may record audio using the built-in microphone and access audio input using Core Audio. Key: `.device.audio-input`

Camera: whether the app may capture movies and still images using the built-in camera. Key: `.device.camera`

Location: whether the app may access location information from Location Services. Key: `.personal-information.location`

Address Book: whether the app may have read-write access to contacts in the user's address book. Key: `.personal-information.addressbook`

Calendars: whether the app may have read-write access to the user's calendar. Key: `.personal-information.calendars`

Photos Library: whether the app may have read-write access to the user's Photos library. Key: `.personal-information.photos-library`

Apple Events: whether the app may send Apple Events to other apps. Key: `.automation.apple-events`

◇ Richard Koch
koch (at) math dot uoregon dot edu
http://math.uoregon.edu/koch/

Parsing complex data formats in LuaTeX with LPEG

Henri Menke

Abstract

Even though it is possible to read external files in TEX, extracing information from them is rather difficult. Ad-hoc solutions tend to use nested if statements or regular expressions provided by several macro packages. However, these quick hacks don't scale well and quickly become unmaintainable.
LuaTEX comes to the rescue with its embedded LPEG library for Lua. LPEG provides a Domain Specific Embedded Language (DSEL) that allows to write grammars in a natural way. In this article I will give a quick introducing to Parsing Expression Grammars (PEG) and then show how to write simple parsers in Lua with LPEG. Finally we will build a JSON parser to demonstrate how easy it is to even parse complex data formats.

1 Quick introduction to LPEG and Lua

The LPEG library [1] is an implementation of Parsing Expression Grammars for the Lua language. It provides a Domain Specific Embedded Language for this task. Its domain is obviously parsing. It is embedded in Lua using overloading of arithmetic operators to give it a natural syntax. The language it implements is PEG. The LPEG library has been included in LuaTEX since the beginning [2]. The examples in this article are based on the talk "Using Spirit X3 to Write Parsers" which was given by Michael Caisse at CppCon 2015 [3], where the speaker introduces the Spirit X3 library for C++ to write parsers using PEG. The Spirit library is not too dissimilar from LPEG and if you are looking for a parser generator for C++, I recommend it.

To make sure that we are all on the same page and the reader can easily understand the syntactic constructions used throughout this manuscript, we review some aspects of the Lua language. First of all, it is to note that all variables are global by default, whereas local variables have to be preceded by the `local` keyword.

```
local x = 1
```

Most of the time we want definitions to be scoped so this pattern will show up very often. Another important thing to note about the Lua language is that in contrast to many other programming languages, functions are first class variables. That means that when we declare a function, what we actually do is assign a value of type `function` to a variable. That is to say, that these two statements are equivalent.

```
function f(...) end    f = function(...) end
```

Lua implements only a single complex datastructure, the table. Tables in Lua act as arrays and key-value storage at the same time, in fact it is possible to mix both forms of access within a single instance as in the following example.

```
local t = { 11, 22, 33, foo = "bar" }
print(t[2], t["foo"], t.foo) -- 22 bar bar
```

Note that array indexing in Lua starts at `1`. For tables and strings Lua offers a useful shortcut. When calling a function with a single literal string or table, parentheses can be omitted. In the following snippet the statements on the left are equivalent to the ones on the right.

```
f("foo")            f"foo"
f({ 11, 22, 33 })   f{ 11, 22, 33 }
```

Especially when programming with LPEG this shortcut can save a lot of typing and, when used to it, makes the code a lot more readable. I will make extensive use of this technique.

2 Why use PEG?

Before we delve into the inner workings of LPEG, let me first give some motivation as to why we would like to build parsers using PEG. Imagine trying to verify that input has a certain format, e.g. a date in the form day-month-year: `09-08-2019`. One approach might be to split the input at the hyphens and verify that each field only contains numbers, which is simple enough to implement using TEX macro code. However, the task quickly becomes more complicated when further requirements come into play. Only because something is made up of three groups of numbers doesn't make it a valid date. In situations like these, regular expressions (regex) sound like a good solution and in fact, the regex to parse a "valid" date looks faily innocent.

```
[0-3][0-9]-[0-1][0-9]-[0-9]{4}
```

I put "valid" in quotation marks, because obviously this regex misses several cases, such as different number of days in different months or leap years. I encourage the reader to look up a regular expression which covers these special cases, to get an impression as to how quickly regex gets out of hand. To top it off, neither a pure TEX solution nor regex implemen-

tations in TEX are fully expandable which is often desirable. Maybe they can be made fully expandable but not without tremendous effort.

3 What is PEG?

The question remains, how does PEG help us here? Let's first look at a more or less formal definition of PEG, adapted from Wikipedia [4]. A parsing expression grammar consists of:

- A finite set N of non-terminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of parsing rules.
- An expression e_S termed the starting expression.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a parsing expression.

To illustrate this, we have a look at the following imaginary PEG for an email address.

$$\langle \text{name} \rangle \leftarrow [\texttt{a}-\texttt{z}]+ \ (\texttt{"."}\ [\texttt{a}-\texttt{z}]+)^*$$
$$\langle \text{host} \rangle \leftarrow [\texttt{a}-\texttt{z}]+ \ \texttt{"."}\ (\texttt{"com"}/\texttt{"org"}/\texttt{"net"})$$
$$\langle \text{email} \rangle \leftarrow \langle \text{name} \rangle \ \texttt{"@"}\ \langle \text{host} \rangle$$

The symbols in angle brackets are the non-terminal symbols. The quoted strings and expressions in square brackets are terminal symbols. The entry point e_S is the rule named email (although the entry point is not specially marked). The present grammar translates into natural language rather nicely. We start at the entry point, the email rule. The email rule tells us that an email is a name, followed by a literal @, followed by a host. The symbols name and host are non-terminal, so they can't be parsed without furhter information so we have to resolve them. A name is specified as one or more characters in the range a to z, followed by zero or more groups of a literal dot, followed by one or more characters a to z. A host is one or more characters a to z, followed by a literal dot, followed by one of the literals com, org, or net. Here the range of characters and the string literals are terminal symbols, because they can be parsed from the input without further information.

As a little teaser, we will have a look how the above grammar translated into LPEG.

```
local name  = R"az"^1 * (P"." * R"az"^1)^0
local host  = R"az"^1 * P"."
            * (P"com" + P"org" + P"net")
local email = name * P"@" * host
```

We can already see that there is sort of a mapping to translate PEG into LPEG, but at first sight it seems like this translation is almost 1:1. We will learn what the symbols mean in the next section.

4 Basic parsers

LPEG provides some basic parsers to make our life a little easier. These map the terminal symbols in the grammer. Here they are with examples:

- `lpeg.P(string)` Matches the provided `string` exactly. This matches "hello" but not "world":

  ```
  lpeg.P("hello")
  ```

- `lpeg.P(n)` Matches exactly `n` characters. To match any single character we could use

  ```
  lpeg.P(1)
  ```

  There is a special character which is not mapped by any encoding which is the end of input. In LPEG there is a special rule for it:

  ```
  lpeg.P(-1)
  ```

- `lpeg.S(string)` Matches any character in `string` (Set). To match any whitespace we use:

  ```
  lpeg.S(" \t\r\n")
  ```

- `lpeg.R("xy")` Matches any character between `x` and `y` (Range). Matching any digit is done using

  ```
  lpeg.R("09")
  ```

  To match any character in the ASCII range we can combine lowercase and uppercase letters:

  ```
  lpeg.R("az", "AZ")
  ```

It is tedious to constantly type the `lpeg.` prefix which is why we omit it from now on. This can be achieved by assigning the members of the `lpeg` table to the corresponding variables.

```
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R -- etc.
```

5 Parsing expressions

By themselves these basic parsers are rather useless. The real power of LPEG comes from the ability to arbitrarily combine parsers. This is achieved by means of parsing expressions. The available parsing expressions are listed in table 1. Below I show some examples where the quoted strings in the comments

represent input that is parsed successfully by the associated parser unless stated otherwise.

Description	PEG	LPEG
Sequence	e_1e_2	`patt1 * patt2`
Ordered choice	$e_1 \vert e_2$	`patt1 + patt2`
Zero or more	e^*	`patt^0`
One or more	$e+$	`patt^1`
Optional	$e?$	`patt^-1`
And predicate	$\&e$	`#patt`
Not predicate	$!e$	`-patt`
Difference		`patt1 - patt2`

Table 1 Available parsing expressions in LPEG with their name and corresponding symbol in PEG. Note that the difference operation is an extension by LPEG and not available in PEG.

- Sequence: This implements the "followed by" operation, i.e. the parser matches only if the first pattern is followed directly by the second pattern.

```
P"pizza" * R"09" -- "pizza4"
P(1) * P":" * R"09" -- "a:9"
```

- Ordered choice: The ordered choice parses the first operand first and only if it fails continues to the next operand. So the ordering is indeed important.

```
R"az" + R"09" + R".,;:?!"
-- "a", "9", ";"
-- "+" fails to parse
```

- Zero or more, one or more, and optional: These are all captured by the same contruct in LPEG, the exponentiation operator. A positive exponent n parses at least n occurences of the pattern, a negative exponent $-n$ parses at most n occurences of the pattern.

```
R"az"^0 + R"09"^1
-- "z86", "abcde99", "99"
R"az"^1 + R"09"^1
-- "z86", "abcde99"
-- "99" fails to parse
R"az"^-1 + R"09"^1
-- "z86", "99"
-- "abcde99" fails to parse
```

- And predicate and not predicate: These two expressions are special in that they don't consume any input. For the not predicate this is obvious because it only matches if the parser it negates does not match.

```
R"09"^1 * #P";"
-- "86;"
-- "99" fails to parse
P"for" * -(R"az"^1)
-- "for()"
-- "forty" fails to parse
```

- Difference: The difference expression will match the first operand only if the second operand does not match. This can be useful to match C style comments which collect everything between the first `/*` and the first `*/`. However, care must be taken that the second operand cannot successfully parse parts of the first operand. If that is the case, the resulting rule will never match.

```
P"/*" * (1 - P"*/")^0 * P"*/"
-- "/* comment */"
P"helloworld" - P"hell"
-- will never match!
```

6 Simple examples

Let us study a simple example which parses two words separated by a space. The LPEG grammar is stored in the variable `rule`. The rest of the example shows the boilerplate that is necessary.

```
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R

local input = "cosmic pizza"

local rule = R"az"^1 * P" " * R"az"^1
print(rule:match(input) .. " of " .. #input)
```

This will print on the terminal "13 of 12" because all the input has been consumed and the parser stopped at the end of input which is the 13th "character" in this string. As we can see the function `rule:match` parses a given input string using a given parser and returns the number of characters parsed. Another way to invoke a parse is using `lpeg.match(rule, input)`, which is equivalent to `rule:match(input)`.

The next example will be slightly more complicated. We will parse a comma-separated list of colon-separated key-value pairs.

```
local input = [[foo : bar ,
gorp : smart ,
falcou : "crazy frenchman" ,
name : sam]]
```

The double square brackets denote one of Lua's long

strings, which can have embedded newlines. The colons and commas that separate keys and values, and entries, respectively, are surrounded by whitespace. To match all possible optional whitespace we use the set parser and the optional expression.

```
local ws = S" \t\r\n"^0
```

With this the specification for the key field is simply one or more letters or digits surrounded by optional whitespace.

```
local name = ws * R("az", "AZ", "09")^1 * ws
```

The value field on the other hand can have either the same specification as the key field, which does not allow embedded whitespaces, or it can be a quoted string, which allows anything between the quotes. To this end we specify the grammar for a quoted string, which is simply the double quotes character, followed by anything that is not double quotes, followed by double quotes. The whole thing may be surrounded by optional whitespace.

```
local quote =
  ws * P'"' *  (1 - P'"')^0 * P'"' * ws
```

Therefore an entry in the key-value list is a `name`, followed by a colon, followed by either a `quote` or a `name`, followed by at most one comma. The whole key-value list is of course just any number of entries, so we apply the zero or more expression to the aforementioned rule.

```
local keyval =
  (name * P":" * (quote + name) * P",")^-1)^0
```

Matching the rule against the input in the same way as the previous example gives "67 of 66".

7 Grammars

The literal parser `P` has a second function. If its argument is a table, the table is processed as a *grammar*. The table has the following layout:

```
P{"<entry point>",
    <non-terminal> = <parsing expression>
    ...
}
```

The string "entry point" is the name of the rule to be processed first. Afterwards the rules are listed in the same manner as they were assigned to variables in the previous example. To refer to non-terminal symbols from within the grammar, the `lpeg.V` funtion is used. Collecting the aforementioned rules into a grammar could look like this:

```
local rule = P{"keyval",
  keyval =
    (V"name" * P":" * (V"quote" + V"name")
      * P",")^-1)^0,
  name =
    V"ws" * R("az", "AZ", "09")^1 * V"ws",
  quote =
    V"ws" * P'"' *  (1 - P'"')^0 * P'"'
      * V"ws",
  ws = S" \t\r\n"^0,
}
```

It becomes a little more verbose because names of non-terminal symbols have to be wrapped in `V"..."`. That is why I personally do not normally include general-purpose rules like the `ws` rule in the example into the grammar, because chances are high I want to use it elsewhere again. The level of verbosity might seem like a disadvantage but the encapsulation is much better that way. It also makes it much easier to define recursive rules, as we will see later.

8 Attributes

In the previous section we have parsed some inputs and confirmed their vailidity by a successful parse and we received the length of the parsed input. An important question remains, how do we extract information from the input? When a parse is successful, the basic parsers synthesize the value they encountered which I am going to call their *attribute*. These attributes can be extracted using LPEG's capture operations.

The simplest capture operation is `lpeg.C(patt)` which simply returns the match of `patt`. Here we parse a strip of only lowercase letters and print the result.

```
local rule = C(R"az"^1)
print(rule:match"pizza") -- pizza
```

Another, very powerful capture is the table capture `lpeg.Ct(patt)` which returns a table with all captures from `patt`. This allows us to write a very simple parser for comma separated values (CSV) in only three lines.

```
local cell = C((1 - P"," - P"\n")^0)
local row = Ct(cell * (P"," * cell)^0)
local csv = Ct(row * (P"\n" * row)^0)

local t = csv:match[[a,b,c
d,e,f
g,,h]]
```

The variable `t` now holds the table representing

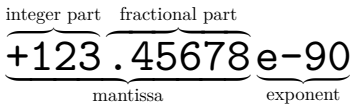
the CSV file and we can access the elements by `t[<row>][<column>]`, e.g. to access the "e" in the middle of the table we can use `t[2][2]`.

There are two more captures which I think are worth mentioning, the grouping capture and the folding capture. The grouping capture `lpeg.Cg(patt [, name])` groups the values produced by `patt`, optionally tagged with `name`. The grouping capture is mostly used in conjunction with the folding capture `lpeg.Cf(patt, func)` which folds the captures from `patt` with the functions `func`. The most common application is parsing of key-value lists. The key and the value are captured independently at first but are then grouped together. Finally they are folded together with an empty table capture.

```
local key = C(R"az"^1)
local val = C(R"09"^1)

local kv = Cg(key * P":" * val) * P","^-1
local kvlist = Cf(Ct"" * kv^0, rawset)

kvlist:match"foo:1,bar:2"
```

9 Actually useful parsers

Now that we know how to parse input and extract data, we can go ahead and start constructing parsers that are acutally useful. We will now construct a parser for floating point numbers. The parser presented here has some limitations. It doesn't handle an integer part that only contains a sign, i.e. `-.1` will not parse. It also doesn't handle hexadecimal, octal, or binary literals. This is left as an exercise to the reader. To construct a possible grammar for floating point numbers, let's take a look at what they look like.



With that we formulate the first rule in our grammar, namely

```
number = (V"int" * V"frac"^-1 * V"exp"^-1)
         / tonumber,
```

i.e. a number has an integer part, followed by an optional fractional part, followed by an optional exponent. The division by `number` that we see here is called a *semantic action*. A semantic action is applied to the result of the parser *ad-hoc*. In general it is a bad idea to use semantic actions, because they don't fit into the concept of recursive parsing and introduce additional state to keep track of. Nevertheless there are some cases when semantic actions are useful, like in this case, where we know that what we just parsed is a number and we merely convert the resulting string into Lua's number type.

Now let's parse the integer part. Here I show all the rules that go into it at once.

```
int = V"sign"^-1 * (R"19" * V"digits"
                                + V"digit"),
sign = S"+-",
digit = R"09",
digits = V"digit" * V"digits" + V"digit",
```

So the integer part is an optional sign, followed by a number between 1 and 9, followed by more digits or just a single digit. A sign is of course just the character `+` or `-`. A single digit is just a number between 0 and 9. The `digits` rule is recursive, because many digits are either a single digit followed by more digits, or just that single digit.

Next is the fractional part, which is very easy. It is just a period followed by digits.

```
frac = P"." * V"digits",
```

Last the exponential part, which is also simple. It is either a lower- or uppercase E, followed by an optional sign, followed by digits.

```
exp = S"eE" * V"sign"^-1 * V"digits",
```

Now let's check this parser with some test input. We expect the result to be the same number that we input and we expect it to be of Lua type `number`.

```
local x = number:match("+123.45678e-90")
print(x .. " " .. type(x))
```

Output: 1.2345678e-88 number

The full code of the number parser is given as part of the JSON parser in the Appendix in lines 5–14.

10 Complex Data Formats: JSON

JSON is short for JavaScript Object Notation and is a lightweight data format that is easy to read and write for both humans and machines. JSON knows six different data types of which two are collections. These are `null`, `bool`, `string`, `number`, `array`, and `object`. This maps nicely to Lua where `null` maps to `nil`, `bool` maps to `boolean`, `string` and `number` map to their eponymous counterparts, and `array` and `object` both map to Lua's `table` type.

On the top level there is always an object, i.e. a JSON file looks roughly like this [5]

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New",
       "onclick": "CreateNewDoc()"},
      {"value": "Open",
       "onclick": "OpenDoc()"},
      {"value": "Close",
       "onclick": "CloseDoc()"}
    ]
  }
}}
```

Before we begin writing a parser for this, we introduce a few general purpose parsers first, which are also not part of the grammar.

```
local ws = S" \t\n\r"^0
```

This rule matches zero or more whitespace characters, where whitespace characters are space, tab, newline and carrige return.

```
local lit = function(str)
    return ws * P(str) * ws
end
```

This function returns a rule that matches a literal string surrounded by optional whitespace. This is useful to match keywords.

```
local attr = function(str,attr)
    return ws * P(str) / function()
            return attr
        end * ws
end
```

This function returns an extension of the previous rule, in that it matches a literal string and if it matched returns an attribute using a semantic action. This is very useful for parsing a string but returning something unrelated, e.g. the `null` value of JSON will be represented by Lua's `nil`.

As mentioned before, at the top level a JSON file expects an object, so this will be the entry point.

```
local json = P{"object",
```

As discussed before JSON supports different kinds of values, so we want to map these in our parsing grammar.

```
value =
    V"null_value" +
    V"bool_value" +
    V"string_value" +
    V"number_value" +
    V"array" +
    V"object",
```

So a `value` is any of the value types defined by the JSON format. That was easy, but now we have to define what these values are and how to parse them. We begin with the easiest ones, the `null` and `bool` values:

```
null_value = attr("null", nil),
bool_value = attr("true", true)
           + attr("false", false),
```

These two types are defined entirely by keyword matching. We use the `attr` function to return a suitable Lua value. Next we define how to parse strings:

```
string_value = ws * P'"'
             * C((P'\\"' + 1 - P'"')^0)
             * P'"' * ws,
```

A string may be surrounded by whitespace and is enclosed in double quotes. Inside the double quotes we can use any character that is not the double quote, unless we escape it `\"`. The value of the string without surrounding quotes is captured. To parse number values, we will reuse the number parser defined in the previous section

```
number_value = ws * number * ws,
```

This concludes the parsing of all the simple datatypes and we move on to the aggregate types, starting with the array.

```
array = lit"["
      * Ct((V"value" * lit",")^-1)^0)
      * lit"]",
```

An array is simply a comma-separated list of values that is enclosed in square brackets. The list is captured as a Lua table. The final and most complicated type to parse is the object.

```
member_pair = Cg(V"string_value" * lit":"
                 * V"value") * lit",")^-1,
object = lit"{"
       * Cf(Ct"" * V"member_pair"^0, rawset)
       * lit"}"
```

An object is a comma-separated list of key-value pairs enclosed in curly braces, where a key-value pair is a string, followed by a colon, followed by a value. To pack this into a Lua table, we use the grouping and folding captures that we discussed before. This concludes the JSON grammar.

```
}
```

The full code of the parser is given in the Appendix with a little nicer formatting. Now we can go ahead an parse JSON files.

```
local result = json:match(input)
```

The variable `result` will hold a Lua table which can be indexed in a natural way. For example, if we had parsed the JSON example given in the beginning of this section, we could use

```
print(result.menu.popup.menuitem[2].onclick)
-- OpenDoc()
```

This way we could write configuration files for our document, parse them on-the-fly when firing up LuaTEX, and configure the style and content according to the specifications.

11 Summary and Outlook

Parsing even complex data formats like JSON is relatively easy using LPEG. A possible next step would be to parsing the LuaTEX input file in the `process_input_buffer` callback and replace templates in the file with values from JSON.

References

[1] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars, *Software: Practice and Experience* **39**(3), 221–258 (2009).

[2] T. Hoekwater, LuaTEX, *TUGboat* **28**(3), 312–313 (2007).

[3] M. Caisse, *Using Spirit X3 to Write Parsers*, https://www.youtube.com/watch?v=xSB-WklPLRvw (2015). (CppCon)

[4] Wikipedia, *Parsing expression grammar*, https://en.wikipedia.org/wiki/Parsing_expression_grammar (online). (Accessed on July 15, 2019)

[5] D. Crockford, *JSON Example*, https://json.org/example.html (online). (Accessed on July 15, 2019)

Henri Menke
9016 Dunedin
New Zealand
henrimenke@gmail.com

12 Appendix: Full code listing of the JSON parser

```
local lpeg = require"lpeg"
local C, Cf, Cg, Ct, P, R, S, V =
    lpeg.C, lpeg.Cf, lpeg.Cg, lpeg.Ct, lpeg.P, lpeg.R, lpeg.S, lpeg.V

-- number parsing
local number = P{"number",
    number = (V"int" * V"frac"^-1 * V"exp"^-1) / tonumber,
    int = V"sign"^-1 * (R"19" * V"digits" + V"digit"),
    sign = S"+-",
    digit = R"09",
    digits = V"digit" * V"digits" + V"digit",
    frac = P"." * V"digits",
    exp = S"eE" * V"sign"^-1 * V"digits",
}

-- optional whitespace
local ws = S" \t\n\r"^0

-- match a literal string surrounded by whitespace
local lit = function(str)
    return ws * P(str) * ws
end

-- match a literal string and synthesize an attribute
local attr = function(str,attr)
    return ws * P(str) / function() return attr end * ws
end

-- JSON grammar
local json = P{
    "object",

    value =
        V"null_value" +
        V"bool_value" +
        V"string_value" +
        V"number_value" +
        V"array" +
        V"object",

    null_value =
        attr("null", nil),

    bool_value =
        attr("true", true) + attr("false", false),

    string_value =
        ws * P'"' * C((P'\\"' + 1 - P'"')^0) * P'"' * ws,

    number_value =
        ws * number * ws,

    array =
```

```
54          lit"[" * Ct((V"value" * lit","^-1)^0) * lit"]",
55
56      member_pair =
57          Cg(V"string_value" * lit":" * V"value") * lit","^-1,
58
59      object =
60          lit"{" * Cf(Ct"" * V"member_pair"^0, rawset) * lit"}"
61  }
```

Evolutionary Changes in Persian and Arabic Scripts to Accommodate the Printing Press, Typewriting, and Computerized Word Processing

Behrooz Parhami
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
parhami@ece.ucsb.edu

1. Introduction

I have been involved in Iran's computing scene for five decades, first as an engineering student and instructor for five years, then as a faculty member at Tehran's Sharif (formerly Arya-Mehr) University of Technology for 14 years (1974-1988), and finally, as an interested observer and occasional consultant since joining University of California, Santa Barbara, in 1988. Recently, I put together a personal history of efforts to adapt computer technology to the demands and peculiarities of the Persian language, in English [1] and Persian [2], in an effort to update my earlier surveys and histories [3-6] for posterity, archiving, and educational purposes.

In this paper, I focus on a subset of topics from the just-cited publications, that is, the three key transition periods in the interaction of Persian script with new technology. The three transitions pertain to the arrivals in Iran of printing presses, typewriters, and computer-based word processors. Specifically, I will discuss how the Persian script was adapted to, and in turn shaped, the three technologies. In each adaptation stage, changes were made to the script to make its production feasible within technological limitations. Each adaptation inherited features from the previous stage(s); for example, computer fonts evolved from typewriter fonts.

2. The Persian Script

Throughout this paper, my use of the term "Persian script" is a shorthand for scripts of a variety of Persian forms (Farsi/Parsi, Dari, Pashto, Urdu), as well of Arabic, which shares much of its alphabet with Persian. Work on adapting the Arabic script to modern technology has progressed in parallel with the work on Persian script, with little interaction between the two R&D communities, until fairly recently, thanks to the Internet.

The Persian language has a 2600-year history, but the current Persian script was adapted from Arabic some 1200 years ago [7]. For much of this period, texts were handwritten and books were copied manually, or reproduced via primitive printing techniques involving etching of the text on stone or wood, covering it with a layer of ink, and pressing paper or parchment against it.

Given the importance attached by Persians to aesthetics in writing, decorative scripts were developed by artists adorning monuments and other public spaces with scripts formed by painting or tilework (Fig. 1). Unlike in printing, typewriting, and computer-based word processing, decorative writing is primarily focused on the proportions and interactions of textual elements and the color scheme, with script legibility being a secondary concern

Fig. 1. Calligraphic writing as art (left; credit: Farrokh Mahjoubi) and tile-based writing at Isfahan's Jāmeh Mosque, which is very similar to modern dot-matrix printing (uncredited photo).

Prior to the arrival of modern technology, Persian was commonly written in two primary scripts: Nastaliq and Naskh. Rules for the scripts were passed on by word of mouth from masters to students. Thus, there were many styles of writing, whose popularity rested on the reputation of the practicing master. Among the rules were proper ways of generating combinations of letter (much like the "fi" & "ffi" combinations in English calligraphy). Because the Naskh script is more readily adaptable to modern technology, including to computer printers and displays, it has become more popular and has pronged into many varieties in recent decades.

Nevertheless, Nastaliq holds a special place in the hearts and minds of Persian-speaking communities. The fanciest books of poetry are still produced in Nastaliq, and some printed flyers use Nastaliq for main headings to embellish and attract attention. Some progress has been made in producing the Nastaliq script automatically, and the results are encouraging. The Web site NastaliqOnline.ir allows its users to produce Nastaliq and a variety of other decorative scripts by entering their desired text within an input box. An image of the generated text can then be copy-pasted into other documents.

One final point about the Persian script, before entering the discussion of the three transition periods: On and off, over the past several centuries, reformation of the Persian script, to "fix" its perceived shortcomings in connection with modernity, has been the subject of heated debates. My personal view is that technology must be adapted to cultural, environmental, and linguistic needs, and not the other way around. Fortunately, success in producing high-quality print and display output has quelled sporadic attempts at reforming the Persian script or changing the alphabet [8], in a manner similar to what was done in Turkey, to save the society from "backwardness."

3. The Transition to Printing Press

The printing press arrived in Iran some 400 years ago (see the timeline in Fig. 2). Shah Abbas I was introduced to Persian and Arabic fonts and decided that he wanted them for his country [9]. A printing press and associated fonts were sent to Isfahan in 1629, but there is no evidence that they were ever put to use. Over the following decades, printing was limited mostly to a few religious tomes.

Broader use of printing technology dates back to 300 years ago. The invention of Stanhope hand-press in 1800 revolutionized the printing industry, because it was relatively small and easy to use. This device was brought to Tabriz, by those who traveled to Europe and Russia, around 1816 [10] and to Isfahan and Tehran a few years later, leading to a flurry of activities in publishing a large variety of books.

A key challenge in Persian printing was the making of the blocks that held the letters and other symbols (Fig. 3). English, with its comparably sized letters and the space between them, was much easier for printing than Persian, which features letters of widely different widths/heights, connectivity of adjacent letters, minor variations in letter shapes involving small dots (imagine having the letter "i," with 1, 2, or 3 dots), and more curvy letters.

Year	Events Affecting the Development of Persian Script
1600	
-	Printing press arrives in Iran; little/no use early on
-	Armenian press established in Jolfa, Isfahan
-	
-	
1700	
-	|
-	| Limited print runs; mostly on poetry and religion
-	|
-	Persian books published in Calcutta
1800	
-	First Stanhope hand-press arrives; printing spreads
-	Presses open in multiple cities; use of lithography
-	Technical books appear; newspapers flourish
-	
1900	First typewriter arrives in Iran
-	
-	Typewriters begin to be used widely
-	Electric typewriters, Linotype, and computers arrive
-	Standards for information code and keyboard layout
2000	Use of personal computers broadens
-	Computer-software and mobile-app industries thrive

Fig. 2. Rough timeline of key events and transitions in the history of adapting the Persian script to modern technology [9].

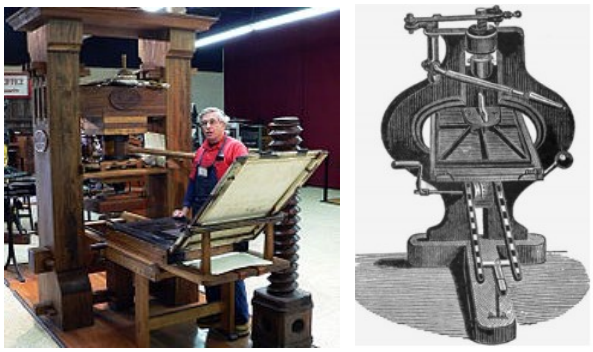


Fig. 3. Re-creation of Gutenberg's press at the International Printing Museum in Carson, California, USA (image: Wikipedia) and the Stanhope hand-press, introduced in 1800 [10].

The first order of business was to make the Persian script horizontally partitionable into letters that could then be juxtaposed to form the desired text. Pre-printing-press Persian script was not horizontally decomposable, as letters tended to mount each other vertically and overlap horizontally (bottom of Fig. 4). The modified form required some compromises in aesthetics, according to the prevailing tastes at the time (top-right of Fig. 4), which proved rather insignificant in retrospect.

Once conceptual changes were made, typographers got busy producing letters, letter combinations, and symbols for Persian printing (Fig. 5). We are now so used to the print-friendly Persian script that the pre-printing-press variants may look quaint to us!

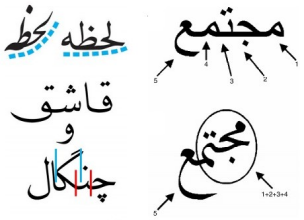


Fig. 4. For printing with movable type, the Persian script had to be made horizontally decomposable (uncredited Web images).

Fig. 5. Early Persian or Arabic metal fonts in the compartments of a typesetter's tray (uncredited Web image)

Fig. 6. Features of Persian script that make its printing difficult also create challenges in automatic text recognition [11].

The variable sizes and spacings of Persian letters also created manufacturing headaches for the font and difficulties for typesetters, who needed to handle blocks of widely different sizes. Interestingly, the features that make typesetting of Persian texts difficult are the same ones that make their automatic recognition challenging (Fig. 6). These include connectivity (a), error-causing minor differences (b), significant width variations (c), horizontal overlaps (d), and vertical overlaps (e).

Eventually, font designers succeeded in rendering the Persian alphabet with four shapes for each letter, in lieu of the nearly unlimited variations in calligraphic writing, where letters morph in shape, depending on the preceding and following letters (and sometimes, according to an even broader context). Still, with 4 variations for each letter, the number of different blocks needed was more than twice that of Latin-based scripts, the latter requiring a total of only 52 lowercase/uppercase letters. This made the utilization of typeface variations (boldface, italics, and the like) a lot more challenging.

Linotype, a hot-metal typesetting system invented by Ottmar Mergenthaler for casting an entire line of text via keyboard data entry, arrived in Iran in the 1950s, transforming and somewhat easing the typesetting problem for daily newspapers [12]. Contemporary Persian print output is now vastly improved (Fig. 7).

Fig. 7. Contemporary Persian newspaper print scripts. (Credit: *The Atlantic* Web site; Atta Kenare / Getty Images).

4. The Transition to Typewriting

Typewriters arrived in Iran around 120 years ago (Fig. 8), but much like the printing press, their use did not catch on right away. By the 1950s, many Western office-machine companies had entered Iran's market. Again, peculiarities of the Persian script created adaptation challenges.

Direct adoption of print fonts was impossible, given that with 32 letters, each of which having four variants, too many keys would be required. For most Persian letters, however, the initial and middle forms, and the solo and end forms, are sufficiently similar to allow combining, with no great harm to the resulting script's readability and aesthetic quality. Of course, early typewriters all using fixed-width symbols, were ill-suited to the Persian script, with its highly-variable letter widths. It would be many years before variable-width symbols improved the Persian typewritten script quality substantially.

For example, the letters "meem" (م) and "beh" (ب) aren't too damaged by having two forms in lieu of four (Fig. 9). The same holds for "heh" (ه), at the left edge of Fig. 9, with slightly more distortion. The letters "ein" (ع) and "ghein" (غ) are the only exceptions needing all four variations (see the top-left of Fig. 9).

One of the highest-quality fonts for typewriters was offered by IBM in its Selectric line, which used a golf-ball print mechanism (right panels of Figs. 8 and 9). The golf-ball was easily removable for replacement with another golf-ball bearing a different font or alphabet (italic, symbol, etc.), making is easy to compose technical manuscripts involving multiple typefaces and equations. Even multiple languages could be easily incorporated in the same document. I used such a typewriter to produce my first textbook, *Computer Appreciation* [13], sample pages of which appear in Fig. 10.

Fig. 8. Mozaffar al-Din Shah's custom-made typewriter, ca. 1900 (Golestan Palace Museum, Tehran) and a later-model IBM Selectric with golf-ball printing mechanism, ca. 1975 (IBM).

Fig. 9. The four shapes of Persian letters and their reduction to two shapes in most cases (left; uncredited Web image) and IBM's Persian golf-ball print mechanism (personal photo).

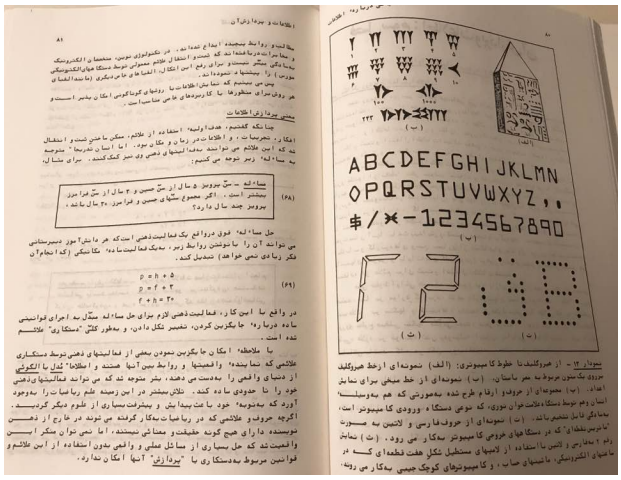


Fig. 10. Pages of the author's book *Computer Appreciation* [13] which he personally created on an IBM Selectric (Fig. 8, right) with a Persian golf-ball print mechanism (Fig. 9, right).

A common approach to building a Persian keyboard was to take an existing Arabic keyboard and add to it the four Persian-specific letters at arbitrary spots, giving rise to a multiplicity of layouts and making it difficult for typists to move between different typewriters. A standard Persian typewriter keyboard layout was thus devised [14]. Years later, standardization was taken up in connection with computer keyboards, creating the "Zood-Gozar" (زود گذر) layout [15], so named because of the sequence of letters at the very bottom row of Fig. 11, similar to the naming of the QWERTY keyboard. However, neither the keyboard layout nor the accompanying data interchange code [16] was adopted, given the pre-/post-revolutionary chaos.

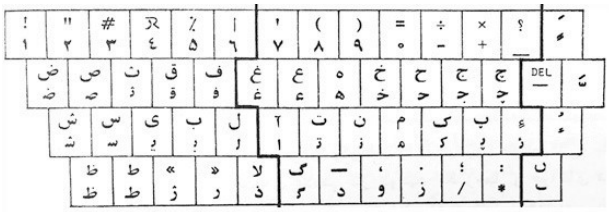


Fig. 11. Unified Persian keyboard layout, a proposed standard for computers, typewriters, and other data-entry systems [15].

Intelligent typewriters soon arrived on the scene. First came word-processors that could store a line of text, thus allowing back-spacing to correct errors by striking the printing hammer on a white ribbon that would overwrite what was previously printed in a given position. This easy erasure mechanism is what allowed a non-professional typist like me to consider self-producing an entire book; cut-and-paste was, of course, still necessary for making larger corrections or moving paragraphs around.

The ultimate in intelligent typewriters, dubbed "word processors," allowed the use of a single key for each letter, with a built-in algorithm deciding which variant of the letter to print. This required a one-symbol delay in printing, as the shape of each letter could depend on the letter that followed it. As an example, to print the word "kamtar" (کمتر), first the letter "kāf" (ك) would be entered. That letter would then be transformed from the solo/end variant to initial-middle form (ـكـ), once the connectable letter "meem" (م) follows. This process continues, until a space or line-break is encountered.

Interestingly, I cannot enter on my Microsoft Word program the initial/middle variant of "kāf" in isolation, as it is automatically converted to the solo/end variant. Thus, in the preceding paragraph, I was forced to connect something to "kāf" and then change the color of that letter to white, in order to make it disappear!

5. The Transition to Computer Printing

True word-processing and desktop publishing arrived in Iran in the 1980s [17], a few years after the worldwide personal-computer revolution. Prior to that, we produced Persian-script output on bulky line-printers and other kinds of printer devices connected to giant mainframes running in air-conditioned rooms of our computer centers, and, in later years, to mini- and micro-computers in our departmental and personal research labs.

One of the earliest computer printer technologies was the drum printer (Fig. 12, left). The rotating drum had one band of letters and symbols for each of the (typically 132) print positions. With the drum rotating at high speed, every letter/symbol would eventually be aligned with the print position, at which time, a hammer would strike on the paper and print ribbon, causing an impression of the raised symbol to be formed on the paper. A complete line was printed after one full revolution of the drum.

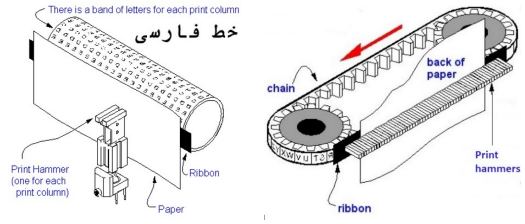


Fig. 12. Print mechanisms in early drum and chain printers (credit: *PC Magazine* Encyclopedia).

Drum printers were bulky and noisy, but, more importantly, were ill-suited to the production of legible Persian script. The separation of the bands of symbols on the drum and the spacing between adjacent hammers led to the appearance of white space between supposedly connected letters (Fig. 12, top-left). This space, combined with up- and down-shifting of symbols due to imprecision in the timing of hammer strikes, led to additional quality problems. The Latin script remains legible if adjacent letters are slightly up- or down-shifted, but the Persian script is much more sensitive to mis-alignment.

The problem with the bulk of drum printers was mitigated with chain (Fig. 12, right) and daisy-wheel printers, but print quality did not improve much, if at all. All three mechanisms suffered from smudging due to high-speed hammer strikes. Thus, letters appeared to be fuzzy, which, ironically, helped with filling the undesirable inter-symbol gaps, but it created additional legibility problems for similar-looking Persian letters.

Several other printing technologies came and went, until improvements in dot-matrix printing made all other methods obsolete. Early dot-matrix printers had a column of 7 pins that made contact with a ribbon to form small black dots on paper (Fig. 13, left). Then, either the needles moved to the next print column or the paper moved in the reverse direction, thereby forming symbols via printing 5 or more columns and continuing on until a complete line of text was formed.

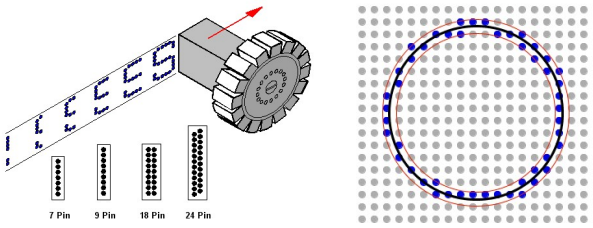


Fig. 13. Early dot-matrix print mechanism with a column of pins (left; credit: *PC Magazine* Encyclopedia) and the versatility of dot-matrix printing for producing images, in addition to text.

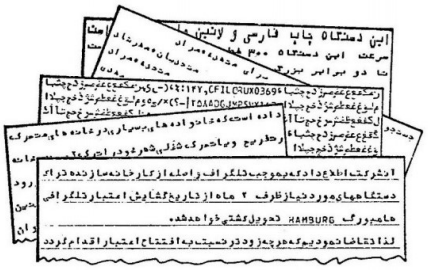


Fig. 14. Examples of Persian scripts produced by line printers and very early dot-matrix printers in the 1970s. [13]

Early dot-matrix printers, though convenient and economical, did not improve the quality of computer-generated Persian scripts, due to the matrix used being too small. In fact, there was a noticeable drop in print quality at first (Fig. 14). As matrix sizes grew and the dots were placed closer and closer to each other, the quality grew accordingly. We faced two categories of R&D problems in those days. First, given a dot-matrix size, how should the Persian letters and digits be formed for an optimal combination of legibility and aesthetic quality? Second, for a desirable level of legibility and aesthetics, what is the minimum required dot-matrix size?

To answer the first question, we would fill out matrices with letter designs and assemble them into lines (at first manually and later using a computer program) to check the script quality (Fig. 15, left). We then repeated the process with different matrix sizes to see the trade-offs. From these studies, we drew two key conclusions in connection with the second question.

First, for low-cost applications in which we cannot afford to use large dot-matrices, a lower bound of 9-by-9/2 dot-matrix size was established, below which legibility and quality become unacceptable. The simulation results for fonts in 7-by-5, 7-by-9/2, and 9-by-9/2 are depicted in Fig. 15, right. A matrix dimension $m/2$ implies the presence of m rows/columns of dots in skewed format, so that the physical dimension of the matrix is roughly $m/2$, despite the fact that there are m elements. This kind of skewed arrangement helps with generating fonts of higher quality, when the letters have curved or slanted strokes.

Second, we used the results from a Persian printed-text automatic recognition study to conclude that a "pen-width" of 4 is adequate for a legible and aesthetically pleasing script output (Fig. 16, left), although, of course, greater resolution can only help (Fig. 16, right).

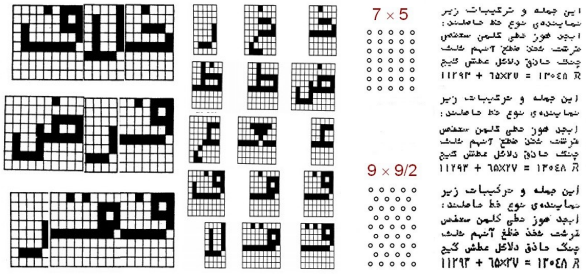


Fig. 15. Illustrating the design of dot-matrix fonts and juxtaposition of letters to check on the quality of the resulting script (left) and results of a study to establish a lower bound on the size of dot-matrix for producing Persian script [18].

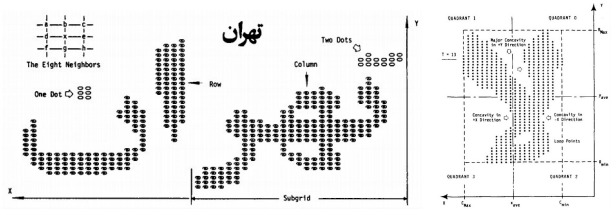


Fig. 16. Decomposition of connected Persian text into letters and recognizing the letters or composite forms [11].

In modern computer applications, a variety of Persian fonts are available to us. Legibility has improved significantly, but the aesthetic quality is still lacking in some cases. In order to make small point sizes feasible, certain features of Persian letters must be exaggerated, so that details are not lost when font sizes are adjusted downward or when images are resized (as in fitting a map on the small screen of a mobile device). Some examples based on the Arial font appear in Fig. 17.

For actual modern computer-generated Persian scripts, I have chosen samples from Microsoft Word (Fig. 18). The samples show both high legibility/quality and problem areas (such as inordinately small dots for Tahoma).

Fig. 17. Illustrating the quality of Persian script using the Arial font of different sizes (top) and the effects of font-size adjustment and image resizing on readability of the resulting text.

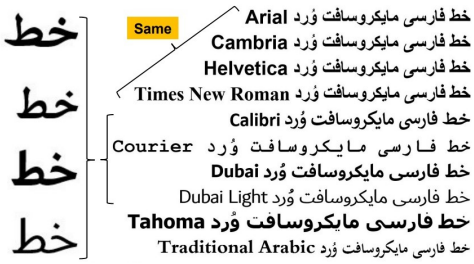


Fig. 18. Examples of modern Persian text output produced by Microsoft Word and the resulting script quality [1-2].

It appears that Calibri and Dubai fonts provide the best combination of legibility and aesthetic quality. The fixed-width Courier sample near the middle of Fig. 18 highlights the fact that fixed-width fonts produce even poorer-quality Persian text than is the case for Latin.

6. Digital Display Technologies

Displays used the dot-matrix approach much earlier than printers. CRT displays, in which an electron beam scans various "rows" on the screen, turning the beam on and off to produce a light or dark point on the screen's coating, constitute a form of dot-matrix scheme. Before modern LCD or LED displays made the use of dot-matrix method for display universal, stadium scoreboards and airport announcement boards used a primitive form of dot-matrix display formed by an array of light bulbs.

For completeness of this historical perspective, I present a brief account of efforts to build Persian line-segment displays for calculators and other low-cost devices. The designs and simulated outputs are depicted in Fig. 19. Peculiarities of the Persian script made the designs of such displays a major challenge. We established that 7 segments would be barely enough for displaying Persian digits and that a minimum of 18 segments would be required for a Persian script that is readable (with some effort). Such displays became obsolete before the project moved to the production stage.

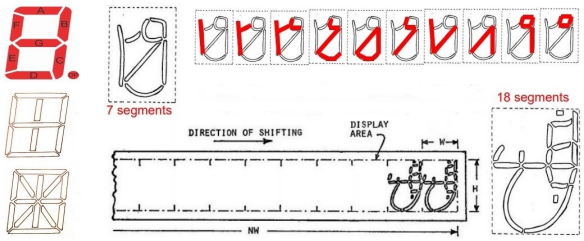


Fig. 19. Line-segment displays for Latin-based alphabets (left) and corresponding designs for Persian digits (top) and letters [1].

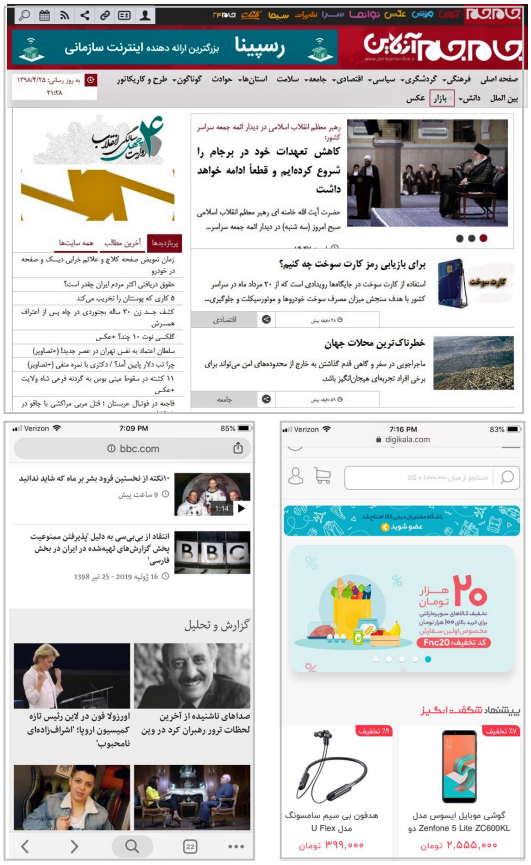


Fig. 20. Persian text displayed on Jam-e Jam news site of the government-run Islamic Republic of Iran Broadcasting system (top; laptop screen capture on July 16, 2019, 10:30 AM PDT) along with the BBC Persian news site and Digikala e-commerce site on a smartphone (bottom; captured the same afternoon).

Dot-matrix display methods are now producing Persian scripts that are comparable in quality to those of our best printers. The transition from CRTs to LCD, LED, and other modern display technologies has removed the flicker problem, the effect of low refresh rate which is particularly significant on CRT displays. Even though modern screens have a much larger number of dots, increases in processing rate and clock speed has made it less likely to have an inadequate refresh rate.

Examples of Persian scripts on modern displays, both spacious desktop/laptop screens and smaller screens found on personal electronic devices appear in Fig. 20. Web sites generally format their contents differently, depending on whether they are viewed on a big screen or a small screen, so that legibility does not become an issue even on the smallest device screens. It is however true that when such screens are viewed in bright environments, such as well-lit offices or outdoors, legibility may suffer.

7. Conclusion and Future Work

Today, technological tools for producing legible and aesthetically pleasing Persian script are widely available. So, whatever problems still remain are algorithmic and software-based in nature. Put another way, whereas until a couple of decades ago, computer typefaces had to be designed with an eye toward capabilities and limitations of printing and display devices, we can now return to typeface design by artists, with only aesthetics and readability in mind. Any typeface can now be mapped to suitably large dot-matrices to produce high-quality and easily-readable Persian script.

We now have reasonably good tools for generating and editing Persian texts. Among them are TeX systems for Arabic [19] and Persian [20], as well as many other text-processing systems based on Unicode [21]. Some popular programming languages also have built-in support for Persian text processing and I/O [22].

What remains to be done are systematic studies of trade-offs between Persian script legibility [23] and aesthetic quality and devising methods for taking care of formatting issues, particularly when bilingual text is involved. Use of crowdsourcing may help with solving the first problem. The second problem has persisted through many attempted solutions over several decades. It is still the case that when, for example, a Persian word is entered within an English text, or vice versa, the text may be garbled depending on the location of the alien word in the formatted line (close to a line break, e.g.). An integrated, easy-to-use bilingual keyboard and improved optical character recognition would be important first steps in solving the remaining text-input problem.

References

[1] B. Parhami, "Computers and the Challenges of Writing in Persian: A Personal History Spanning Five Decades," being prepared for publication. (English version of [2])

[2] B. Parhami, "Computers and Challenges of Writing in Persian" (in Persian), *Iran Namag*, Vol. 4, No. 2, Summer 2019, to appear. (Persian version of [1])

[3] B. Parhami and F. Mavaddat, "Computers and the Farsi Language: A Survey of Problem Areas," *Information Processing 77* (Proc. IFIP World Congress), North Holland, 1977, pp. 673-676.

[4] B. Parhami, "On the Use of Farsi and Arabic Languages in Computer-Based Information Systems," *Proc. Symp. Linguistic Implications of Computer-Based Information Systems*, New Delhi, India, November 1978, pp. 1-15.

[5] B. Parhami, "Impact of Farsi Language on Computing in Iran," *Mideast Computer*, Vol. 1, No. 1, pp. 6-7, September 1978.

[6] B. Parhami, "Language-Dependent Considerations for Computer Applications in Farsi and Arabic Speaking Countries," *System Approach for Development* (Proc. IFAC Conf.), North-Holland, 1981, pp. 507-513.

[7] G. Lazard, "The Rise of the New Persian Language," *The Cambridge History of Iran*, Vol. 4 (Period from the Arab Invasion to the Saljuqs), 2008, pp. 566-594.

[8] M. Borjian and H. Borjian, "Plights of Persian in the Modernization Era," *Handbook of Language and Ethnic Identity: The Success-Failure Continuum in Language and Ethnic Identity Efforts*, Vol. 2, pp. 254-267, 2011.

[9] W.M.Floor, "Čāp," *Encyclopedia Iranica*, I/7, pp. 760-764.

[10] N. Green, "Persian Print and the Stanhope: Industrialization, Evangelicalism, and the Birth of Printing in Early Qajar Iran," *Comparative Studies of South Asia, Africa, and the Middle East*, Vol. 30, No. 3, 2010, pp. 473-490.

[11] B. Parhami and M. Taraghi, "Automatic Recognition of Printed Farsi Texts," *Pattern Recognition*, Vol. 14, Nos. 1-6, pp. 395-403, 1981.

[12] T. Nemeth, *Arabic Type-Making in the Machine Age: The Influence of Technology on the Form of Arabic Type, 1908-1993*, Brill, Leiden, 2017, p. 288.

[13] B. Parhami, *Computer Appreciation* (in Persian), Tehran, Tolou'e Azadi, 1984.

[14] Institute of Standards and Industrial Research of Iran, *Character Arrangement on Keyboards of Persian Typewriters* (in Persian), ISIRI 820, 1976.

[15] B. Parhami, "Standard Farsi Information Interchange Code and Keyboard Layout: A Unified Proposal," *J. Institution of Electrical and Telecommunications Engineers*, Vol. 30, No. 6, pp. 179-183, 1984.

[16] Iran Plan and Budget Organization, *Final Proposal for the Iranian National Standard Information Code* (INSIC), Persian and English versions, 1980.

[17] M. Sanati, "My Recollections of Desktop Publishing" (in Persian), *Computer Report*, Vol. 40, No. 239, pp. 53-60, Fall 2018.

[18] B. Parhami, "On Lower Bounds for the Dimensions of Dot-Matrix Characters to Represent Farsi and Arabic Scripts," *Proc. 1st Annual CSI Computer Conf.*, Tehran, Iran, December 1995, pp. 125-130.

[19] K. Lagally, "ArabTEX, a System for Typesetting Arabic," *Proc. 3rd Int'l Conf. Multi-lingual Computing: Arabic and Roman Script*, Vol. 9, No. 1, 1992.

[20] B. Esfahbod and R. Pournader, "FarsiTeX and the Iranian TeX Community," *TUGboat*, Vol. 23, pp. 41–45, 2002.

[21] Unicode.org, "About the Unicode Standard," on-line resource page with pertinent links, accessed on July 16, 2019: https://unicode.org/standard/standard.html

[22] Python.org, "Links to Python Information in Persian/Iranian/Farsi," On-line resource page, accessed on July 16, 2019: https://wiki.python.org/moin/PersianLanguage

[23] N. Chahine, "Reading Arabic: Legibility Studies for the Arabic Script," Doctoral Thesis, Leiden University, 2012.

Type 3 Fonts and PDF Search

Tomas Rokicki

Abstract

PDF files generated from the output of `dvips` using bitmapped fonts are not properly searchable, indexable, or accessible. While a full solution is challenging, only minimal `dvips` changes are required to support English language text, changes that are at least two decades overdue. I will describe these changes and discuss their limitations.

1 Introduction

The Type 3 fonts generated by `dvips` for bitmapped fonts lack a reasonable encoding vector, and this prevents PDF viewers from interpreting those glyphs as text. This in turn prevents text search, copy and paste, screen readers, and search engine indexing from working correctly. Fixing this is easy, at least for English text, and comes with no significant cost.

This small change is not nearly a full solution to create accessible PDF multilingual documents. Modern support for eight-bit input encodings [2], explicit font encodings [3], and direct generation of PDF can yield better results. But if you want to use METAFONT fonts as-generated and `dvips`, this is an important change.

I describe how I generated reasonable encoding vectors for common METAFONT fonts, how `dvips` finds these encoding vectors and embeds them in the PostScript file, and how the current implementation allows for future experimentation and enhancement.

2 A Little History

When `dvips` was originally written in 1986, the lone PostScript interpreter on hand was an Apple Laserwriter with 170K available memory. I treated PostScript as just a form of compression for the page bitmap, doing the bare minimum to satisfy the requirements for Level 1 Type 3 fonts. One of those requirements was to supply an `/Encoding` vector, despite the fact that at the time, the vector was completely unnecessary in rendering the glyphs. Not considering that people might someday use that encoding vector for glyph identification, on that fateful day in 1986 I generated a semantically nonsensical but syntactically acceptable vector (`/A0`–`/H3` in base 36) for all bitmapped fonts, and this vector remains to this day, subverting any attempt to search copy, or use screen readers.

Replacing this encoding vector with something more reasonable allows PDF viewers to properly understand what characters are being rendered, at least for English-language text.

3 A Sample

The following TEX file, cribbed from `testfont.tex` but using only a single font, will be used for illustration.

```
\hsize=3in \noindent
On November 14, 1885, Senator \& Mrs.~Leland
Stanford called together at their San
Francisco mansion the 24~prominent men who
had been chosen as the first trustees of The
Leland Stanford Junior University.
?‘But aren’t Kafka’s Schlo{\ss} and {\AE}sop’s
{\OE}uvres often na{\"\i}ve  vis-\‘a-vis the
d{\ae}monic ph{\oe}nix’s official r\^ole
in fluffy souffl\’es?
\bye
```

When you run this through TEX and `dvips` (giving the `-V1` option to enforce bitmapped and not Type 1 fonts), and then `ps2pdf`, the resulting PDF does not support text search in most PDF viewers. In Acrobat with copy and paste it almost works; the c's are dropped throughout (San Francisco becomes San Fran is o). The c's are dropped because the original `dvips` encoding uses `/CR` as the name for this character, and it is apparently interpreted as a non-marking carriage return. Ligatures also don't work. In OSX Preview (the default PDF viewer for the Mac), selecting text appears to fail (it actually works, but the selection boxes are too small to see that anything has actually been selected) and no characters are recognized as alphabetic. In Chrome PDF preview, selecting text gives a random note appearance with each word separately selected by its bounding box and no alphabetic characters recognized.

Conversely, when you process the file with Type 1 fonts, all text functions perform normally, except that accented characters are detected as two separate characters (the accent and the base character). The critical difference is not Type 3 (bitmaps) versus Type 1 (outline fonts), but rather the lack of a sensible encoding vector in the Type 3 font.

4 First Attempts and Failure

If I manually copy the `Encoding` vector from the output of dvips using Type 1 fonts and put that in the font definition for the Type 3 fonts, the situation improves; now Adobe Acrobat properly supports text functions (including ligatures but not accented characters). The other PDF viewers now recognize alphabetic characters, but they still have a number of problems.

With OSX Preview, if you use command-A (to select all the text) and then command-C (to copy it),

and you copy the result into a text editor (or a word processing program "without formatting"), you get the following mismash of text:

> On Novemb er 14, 1885, Senator & Mrs. Leland Stanford called mansion the 24 together at their San Francisco prominent men who had b een cho- Stanford sen as the first trustees of The Leland Junior Æsop's University. ¿But aren't Kafka's Schloß and Œuvres often na"ıve vis-'a-vis the dæmonic phœnix's official rˆole in fluffy souffl'es?

In addition to the broken words and split accented characters, if you look carefully you will notice some surprising and substantial word reordering! What could be going on?

5 Refinements and Success

All PDF viewers use some heuristics to turn a group of rendered glyphs into a text stream. The heuristics differ significantly from viewer to viewer. The most important heuristic appears to be interpreting horizontal escapement into one of three categories: kerns, word breaks, and column gutters. OSX Preview was failing so badly because it was recognizing rivers in the paragraph as separating columns of text. To satisfy the PDF viewers I had access to, I made two additional modifications to each bitmapped font.

First, I adjusted the font coordinate system. The default Adobe font coordinate system has 1000 units to the em, while the original `dvips` uses a coordinate system with one unit to the pixel both for the page and for the font, and doesn't use the PostScript `scalefont` primitive. But not using `scalefont` apparently makes some viewers think all the fonts are just one point high, and they use spacing heuristics appropriate for such a font. By providing a font matrix more in line with conventional fonts, and using `scalefont`, PDF viewers make better guesses about the appropriate font metrics for their heuristics.

Second, I provide a real font bounding box. The original `dvips` code gives all zeros for the font bounding box, which is specifically allowed by PostScript, but this confuses some PDF viewers. So I wrote code to calculate the actual bounding box for the font from the glyph definitions.

With these adjustments, using `dvips` with bitmapped fonts and `ps2pdf` generates PDF files that can be properly searched with most PDF viewers—at least, for English language text.

6 Other Languages: No Success

I would have liked things to work with other languages as well, but was not able to get it to work. Clearly the PDF viewers are recognizing characters by the glyph names, but this appears to work only with a small set of glyph names. I hoped that those listed in the official Adobe Glyph List [1] would work, but in my experiments they (for the most part) did not. I also tried Unicode code point glyph names such as `/uni1234` and `/u1234` but neither of these formats worked in the PDF viewers I tried. I also experimented with adding a `cmap` to the font, with no success, and even tried some lightly documented GhostView hacks, but was only able to achieve distressingly partial success for most non-Roman characters.

Even if the individual glyphs are recognized, problems remain with accents, and more generally, virtual fonts. With a standard seven-bit encoding, accents are generally rendered as two separate characters, where the PDF viewer expects to see only a single composite character. Further, the entire virtual font layer would need to be mapped in some fashion, as the PDF contains the physical glyphs that are often combined in some way to provide the semantic characters. Supporting this would have required significantly more effort and heuristics, and there are already efforts in this direction from people much more knowledgeable and capable than I am. The most logical general solution is to use properly coded input, such as UTF-8, and where transformation to multiple glyphs is necessary, embed the appropriate mapping information directly in the PDF file.

The lack of success for other languages diminishes these proposed changes, but the changes are still important as they do provide reasonable support for English-language documents. Since PDF viewers are a moving target, as are the PostScript to PDF converters, the implementation provides for some future experimentation and extension.

7 Finding Font Encodings

In order to provide more than a proof of concept, I had to locate appropriate glyph names for the fonts provided with TEXLive, as well as provide a mechanism for end users to add their own glyph names for their own personal fonts.

Over the years others have translated nearly all (if not all) of the METAFONT fonts provided with TEXLive, and as part of that process, reasonable encoding vectors have been created for the glyphs. I decided to leverage this work, so I wrote a script that located all the METAFONT sources in the TEXLive distribution, all the corresponding Type 1 fonts, and any encoding files used in the relevant `psfonts.map` file. A big Perl script chewed on all of this, extracting encoding vectors and creating appropriate files for `dvips`. Some of the encoding vectors use glyph names

that are not particularly useful, and some use glyph names based on Unicode code points that are not currently recognized by the PDF viewers I tried. I did not want to edit the names in any way; I aimed for functional equivalence to using the Type 1 fonts. If improvements are made to the Type 1 font glyph names, or to the PDF viewers, I wanted to be able to pick up those improvements.

I considered having `dvips` read the encoding vectors directly from the Type 1 fonts, rather than extracting them and storing them elsewhere, but decided against this; I wanted `dvips` to use appropriate glyph names even if the Type 1 fonts didn't exist at all. This does introduce redundancy which can potentially lead to an inconsistency in the glyph names, but the fonts are currently mostly stable, and the glyph name extraction process can be repeated as needed if meaningful changes are made.

8 Storing and Distributing Encodings

After scanning all of the relevant METAFONT files and corresponding Type 1 files, I found there were 2885 fonts; storing the encodings separately one per font would require an additional 2,885 files in TEXLive, occupying about 5 megabytes. I felt this was excessive for the functionality added.

Karl Berry suggested combining all the encodings into a single file, along with a list of fonts using any particular encoding. Since there were only 138 distinct encodings, this gave tremendous compression, letting me store all of the encodings for all of the fonts in a single file of size 183K. This also enabled me to distribute a simple test Perl script that mocked the changes so people could try them out without updating their TEX installation.

This combined file, called `dvips-all.enc`, provides the default encoding used by the 2885 distributed TEXLive METAFONT fonts. In every case that `dvips` looks for an encoding, e.g., for `cmr10`, it first searches for `dvips-cmr10.enc` and only falls back to the information in the combined file if the font-specific file is not found. This permits users to override the provided encodings, as well as define their own encoding for local METAFONT fonts.

The format of the encoding file is slightly different from that of other encoding files in the TEXLive distribution. The encoding file should be a PostScript fragment that pushes a single object on the operand stack. That object should either be a legitimate encoding vector consisting of an array of 256 PostScript names, or it should be a procedure that pushes such an encoding vector. It should not attempt to define the `/Encoding` name in the current dictionary, as some other encoding file formats do. A sample file, one that can be used for `cmr10` (and many other Computer Modern fonts) is:

```
[/Gamma/Delta/Theta/Lambda/Xi/Pi/Sigma/Upsilon
/Phi/Psi/Omega/ff/fi/fl/ffi/ffl/dotlessi
/dotlessj/grave/acute/caron/breve/macron/ring
/cedilla/germandbls/ae/oe/oslash/AE/OE/Oslash
/suppress/exclam/quotedblright/numbersign
/dollar/percent/ampersand/quoteright/parenleft
/parenright/asterisk/plus/comma/hyphen/period
/slash/zero/one/two/three/four/five/six/seven
/eight/nine/colon/semicolon/exclamdown/equal
/questiondown/question/at/A/B/C/D/E/F/G/H/I/J
/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z/bracketleft
/quotedblleft/bracketright/circumflex
/dotaccent/quoteleft/a/b/c/d/e/f/g/h/i/j/k/l
/m/n/o/p/q/r/s/t/u/v/w/x/y/z/endash/emdash
/hungarumlaut/tilde/dieresis
128{/.notdef}repeat]
```

9 Deduplicating Encodings

The encodings inserted in the fonts do use a certain amount of PostScript memory, and this memory usage is not presently accounted for in the memory usage calculation of `dvips`. The memory usage is small and modern PostScript interpreters have significant memory. Further, I doubt anyone actually sets the `dvips` memory parameters anymore anyway. So this is unlikely to be an issue. But to minimize the effect, and also to minimize the impact on file size, encodings that are used more than once are combined into a single instance and reused for subsequent fonts.

10 The `dvips` Changes

Almost all changes to `dvips` are located in the single new file `bitmapenc.c`, although a tiny bit of code was added to `download.c` to calculate an aggregate font bounding box, and the font description structure extended to store this information. I also added code to parse command line options and configuration file options to disable or change the behavior of the new bitmap encoding feature.

By default this feature is turned on in the new version of `dvips`. If no encoding for a bitmapped font is found, no change is made to the generated output for that font.

11 Testing the Changes Without Updating

You can test my proposed changes to the `dvips` output files without updating your distribution or building a new version of `dvips`. The Perl script `addencodings.pl` [4] reads a PostScript file generated by `dvips` on standard input and writes the PostScript file that would be generated by a modified `dvips` on standard output. No additional files

are required for this testing; the default encodings for the standard TEX Live fonts are built in to the Perl script.

12 How to Use a Modified `dvips`

In general, `dvips` usage is unchanged. Warnings in the functionality of the bitmap encoding are disabled by default, so as to not disturb existing workflows; this may change in the future.

We add a single command line and configuration option, using the previously unused option character `J`. The option `-J0` disables the new bitmap encoding functionality. The option `-J` or `-J1` enables it but without warnings, and is the default. The option `-J2` enables it with warnings for missing encoding files.

13 Extension Support

Remember that the encoding file is an arbitrary PostScript fragment that pushes a single object on the operand stack, and that object can be a procedure. I permit it to be a procedure to support experimenting with other changes to the font dictionary to improve text support in PDF viewers. For instance, if a technique for introducing Unicode code points for glyphs into a PostScript font dictionary is standardized and supported by various PostScript to PDF convertors, such a procedure can introduce the requisite structures. The procedure will not be executed until the font dictionary for the Type 3 font is created and open.

To test this functionality, I created a `rot13.enc` file that defines a procedure that modifies the Encoding vector to swap single alphabetic characters much like the `rot13` obfuscation common during the Usenet days. With this modification, copying text from a PDF copies (mostly) content that has been obfuscated (except for ligatures). This brings us full circle to the current unreadable text copied from the original `dvips`.

References

[1] Adobe. Adobe glyph list specification. `https://github.com/adobe-type-tools/agl-specification`, August 2018.

[2] A. Jeffrey and F. Mittelbach. inputenc.sty. `https://ctan.org/pkg/inputenc`, 2018.

[3] R. Moore. Include cmap resources in pdf files from pdfTEX. `https://ctan.org/pkg/mmap`, 2008.

[4] T. G. Rokicki. Type 3 search code. `https://github.com/rokicki/type3search`, July 2019.

◇ Tomas Rokicki
Palo Alto, California
United States
`rokicki@gmail.com`

The Design of the HINT File Format

Martin Ruckert

Abstract

The HINT file format is intended as a replacement of the DVI or PDF file format for on-screen reading of TEX output. Its design should therefore meet the following requirements: reflow of text to fill a window of variable size, convenient navigating of text with links in addition to paging forward and backward, efficient rendering on mobile devices, simple generation from existing TEX input files, and an exact match of traditional TEX output if the window size matches TeX's paper size.

This paper describes the key elements of the design and motivates the design decisions.

Why do we need a new file format?

The first true output file format for TEX was the DVI format[2]. When PostScript became available, it was soon supplemented by dvips[7], and now, most people I know use pdftex to produce TEX output in PDF format. There are two good reasons for that: To begin with, the PDF format is a perfect match[4] for the demands of the TEX typesetting engine, but first and foremost, the PDF format is in wide spread use. It enables us to send documents produced with TEX to practically anybody around the globe and be sure that the receiver will be able to open the document and that it will print exactly as intended by its author (unless a font is neither embedded in the file nor available on the target device) .

But the main limitation of the PDF format is its inherent inability to adapt to the given window size. For reading documents on mobile devices, the HTML format is a much more convenient format. Part of the concept of HTML is a separation of content and presentation: the author prepares the content, the browser decides on the presentation—at least in principle. It turns out that designers of web pages spare no effort to control the presentation, but often the results are poor. Different browsers have different ideas about presentation, users' preferences and operating systems interfere with font selection, and all that might conflict with the presentation the author had in mind. When is comes to eBooks, the popular epub format[3] is derived from HTML and inherits its advantages as well as its shortcomings. As a consequence, eBooks when compared with printed books are often of inferior quality.

What is needed, is a document format, that meets the demands of the TEX typesetting engine and that gives the author as much control over the presentation as possible but still can adapt to a given paper format—be it real or electronic paper. These two design objectives guided the development of the HINT file format.

While the TEX typesetting engine, its internal representation of data, its algorithms, and its debugging output, was the driving force of the development of the HINT file format, giving the whole project its name (the recursive acronym for "HINT Is Not TEX"), the result is not limited to the TEX universe. In the contrary, it makes the best parts of TEX available to all systems that use the HINT file format.

Faithful Recording of TEX output

At the beginning of the design, the primary necessity was the ability to faithfully capture the output of the TEX typesetting engine.

To build pages, TEX adds nodes to the so called "contribution list". The content of a HINT file is basically a list of all these nodes from which a viewer can reconstruct the contributions and build pages using TEX's original algorithms. So with few exceptions, TEX nodes are matched one-to-one by HINT nodes.

Of course, we need characters, ligatures, kerns, rules, hlists and vlists; and as in TEX, dimensions are expressed as scaled points. But even a simple and common construction like `\hbox to \hsize {...}` requires new types of nodes: a horizontal list that may contain glue nodes and has a width that depends on `\hsize` which is not known when the HINT file is generated. To express dimensions that depend on `\hsize` and `\vsize`, HINT uses linear functions $w + h \cdot$ `\hsize` $+ v \cdot$ `\vsize`, called *extended dimensions*. Linear functions are a good compromise between expressiveness and simplicity. The computations that most TEX programs perform with `\hsize` and `\vsize` are linear and in the viewer, where `\hsize` and `\vsize` are finally known, extended dimensions are easily converted to ordinary dimensions. Necessarily, HINT adopts TEX's concepts of stretchability, shrinkability, glue, and leaders.

One of the highlights of TEX is its line breaking algorithm. And because line breaking depends on `\hsize`, it must be performed in the viewer. But wait, an expensive part of line breaking is hyphenation and this can be done without knowledge of `\hsize`. So HINT defines a paragraph node, its width

preliminary draft, 13 Jul 2019 16:59 *preliminary draft, 13 Jul 2019 16:59*

is an extended dimension, and all the words in it contain all possible hyphenation points in the form of TEX's discretionary hyphens. To maintain complete compatibility between TEX and HINT, two types of hyphenation points had to be introduced: explicit and automatic. TEX uses a three pass approach for breaking lines: In the first pass, TEX will not attempt automatic hyphenation and uses only discretionary hyphens that are already provided by the author. Likewise HINT will use in its first pass only the explicit hyphenation points. Given the same value of `\hsize`, TEX and HINT will produce exactly the same line breaks. In a paragraph node, HINT also allows vadjust nodes and a new node type for displayed formulas to make sure that the positioning of displayed equations and their equation numbers is exactly as in TEX.

The present HINT format has also an experimental image node that can stretch and shrink like a glue node. Therefore, images stretch or shrink together with the surrounding glue to fill the enclosing box. The insertion of images in TEX-documents is common practice. But TEX treats images as "extensions" that are not standardized. In a final version of HINT, I expect to have a more general media node. I think it is better to have a clearly defined, limited set of media types that is supported in all implementations than a wide variation of types with only partial support.

One node type of TEX that is not present in HINT is the mark node. TEX's mark nodes contain token lists, the "machine code" for the TEX interpreter, and for reasons explained next, HINT does not implement token lists.

Efficient and Reliable Rendering

On mobile devices, rendering must be efficient and files must be self-contained. To meet these goals, the proper foundation is laid in the design of the file format.

The most important decision was to ban the TEX interpreter from the rendering application. A HINT file is pure data. As a consequence, TEX's output routines (and with them mark nodes) were replaced by a template mechanism. Templates, while not as powerful as programs, will always terminate and can be processed efficiently. Whether they offer sufficient flexibility has to be seen. It is a fact, however, that only very few users of TEX or LATEX write their own output routines. So it can be expected that a collection of good templates will serve most authors well.

The current template mechanism of HINT is still experimental. It is sufficient to replace the output routines of plain TEX and LATEX.

HINT files contain all necessary resources, notably fonts and images, making them completely self-contained. Embedding the fonts will make HINT files larger—the effect is more pronounced for short texts and less significant for large books—and it makes HINT files independent of local resources and of local character encodings. Indeed, a HINT file does not encode characters, it encodes glyphs. While HINT files use the UTF8 encoding scheme, it is possible to assign arbitrary numbers to the glyphs as long as the assignment in the font matches the assignment in the text. The only reason not to depart from the standard UTF8 encoding is the ability to search for user-entered strings.

Zoom and Size Changes

On mobile devices it is quite common to switch within one application between landscape or portrait mode to use the screen space as efficient as possible. Further, users usually can adjust the size of displayed content by zooming in or out.

For rendering a HINT file, these operations simply translate into a change of `hsize` and `vsize`, with consequences for line and page breaking. While changing line breaks affects only individual paragraphs, changing a page break has global implications which makes precomputing page breaks impractical. Consequently, the HINT file format must support rendering either the next page or the previous page based alone on the top or bottom position of the current page and this implies that it must be possible to parse the content of a HINT file in forward as well as in backward direction.

A HINT file encodes TEX's contribution list in its content section. To support bidirectional parsing, each encoding of a node starts with a tag byte and it ends with the very same tag byte. From the tag byte, the layout of the encoding can be derived. So decoding in backward direction is as simple as decoding in forward direction. Changes in the parameters of TEX, for example paragraph indentation or baseline distance, pose another problem for bidirectional parsing. HINT solves this problem by using a stateless encoding of content. All parameters are assigned a permanent default value. To specify these defaults, HINT files have a definition section. Any content node that needs a deviation from the default values must specify the new values locally. To make local changes efficient, nodes in the content section can reference suitable predefined lists

preliminary draft, 13 Jul 2019 16:59 *preliminary draft, 13 Jul 2019 16:59*

of parameter values specified again in the definition section.

Simple and Compact Representation

On the top level, a HINT file is a sequence of sections. To locate each section in the file, the first section of a HINT file is the directory section; it's a sequence of entries that specify location and size of each section. The first entry in the directory section, the root entry, describes the directory section itself. The HINT file format supports compressed sections according to the zlib specification[1]. Using the directory, access to any section is possible without reading the entire file.

The directory section is preceded by a banner line: It starts with the four byte word `hint` and the version number; it ends with a line-feed character. The directory section is followed by two mandatory sections: the definition section and the content section. All further sections, containing fonts, images, or any other data, are optional. The size of a section must be less or equal to 2^{32} byte. This restriction is strictly necessary only for the content section. It sets a limit of about 500 000 pages and ensures that positions inside the content section can be expressed as 32 bit numbers.

For debugging, the specification of a HINT file also describes a "long" file format. This long file format is a pure ASCII format designed to be as readable as possible. Two programs, `stretch` and `shrink`, convert the short format to the long format and back, and constitute—as literate programs[5]—the format specification[8].

Since large parts of a typical content section contain mostly character sequences, there is a special node type, called a text node, optimized for the representation of plain text. It breaks with two conventions that otherwise are true for any other node: The content of a text node can not be parsed in backward direction, and it depends on a state variable, the current font. To mitigate the restriction to forward parsing, the size of a text node is stored right before the final tag byte. This enables a parser to move from the final tag byte directly to the beginning of the text. Since text nodes can not span multiple paragraphs, they are usually short.

Inside a text, all UTF8 codes in the range $2^5 + 1$ to 2^{20} encode a character in the current font; codes from `0x00` to `0x20` and `0xF8` to `0xFF` are used as control codes. Some of them are reserved as shorthand notation for frequent nodes—for example the space character `0x20` encodes the inter-word-glue—others introduce font changes or mark the start of a node given in its regular encoding.

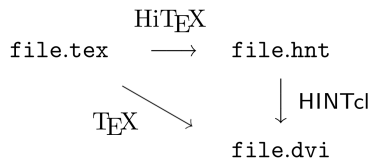
The two forms of content encoding, as regular nodes or inside a text node, introduce a new requirement: when decoding starts at a given position, it must be possible to decide whether to decode a regular node, an UTF8 code, or a control code. Control codes have only a limited range and the values of tag bytes can be chosen to avoid that range. Conflicts between UTF8 codes and tag bytes can not be avoided. Hence positions inside text nodes are restricted to control codes. A position of an arbitrary character inside a text node can still be encoded because there is a control code to encode characters (with a small overhead).

Clear Syntax and Semantics

Today, there are many good formal methods to specify a file format, and the time when file formats where implicit in the programs that would read or write these files seems like ancient history. The specification of the HINT file format, however, is given as two literate programs: `stretch` and `shrink`. The first reads a HINT file and translates it to the "long" format and the second goes the opposite direction and writes a HINT file.

Of course, these programs use modern means like regular expressions and grammar rules to describe input and output and are, to a large extend, generated from the formal description using `lex` and `yacc`. For this purpose, the `cweb` system[6] for literate programming had to be extended to generate and typeset `lex` and `yacc` files. I consider this representation an experiment. I tried to combine the advantages of a formal syntax specification with the less formal exposition of programs that illustrate the reading and writing process and can serve as reference implementations. The programs `stretch` and `shrink` can also be used to verify that HINT files conform to the format specification.

Specifying semantics is a difficult task and a formal specification is entirely impossible if the correctness depends partly on personal taste. Fortunately the new file format is just an "intermediate" format as part of the TEX universe. So the following commutative diagram is an approximation to a formal specification:



Currently the programs HiTEX and HINTcl mentioned in the diagram are still under development. HiTEX is a modified version of TEX that produces HINT files as output; HINTcl is a command line program, that reproduces TEX's page descriptions as if `\tracingoutput` where enabled. While it does not actually produce a DVI file, its output can be compared to the page descriptions in TEX's `.log` file to make sure the diagram above would indeed be commutative. The prototypes available so far do not yet support all the features of TEX or HINT.

Conclusion

The experimental HINT file format proves that file formats supporting efficient, high quality rendering of TEX output on electronic paper of variable size are possible. The upcoming prototypes for a TEX version (HiTEX) that produces such files and viewer programs on Windows and Android will provide a test environment to investigate and improve concepts and performance in practice.

In the long run, I hope that a new standard for electronic documents will emerge that enjoys wide spread use, provides the output quality of real books, is easy to use and powerful enough to encode TEX output, offers the author maximum control over the presentation of her or his work, and can cope with the variations in screen size and screen resolution of modern mobile devices.

References

[1] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC Editor, 1996.

[2] David Fuchs. The format of TEX's DVI files. *TUGboat*, 3(2):14–19, October 1982.

[3] EPUB 3 Community Group. epub 3. `http://www.w3.org/publishing/groups/epub3-cg`.

[4] Hans Hagen. Beyond the bounds of paper and within the bounds of screens; the perfect match of TEX and Acrobat. In *Proceedings of the Ninth European TEX Conference*, volume 15a of *MAPS*, pages 181–196. Elsevier Science, September 1995.

[5] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[6] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison Wesley, 1994. `https://ctan.org/pkg/cweb`.

[7] Tom Rokicki. *Dvips: A DVI-to-PostScript translator*.

[8] Martin Ruckert. *HINT: The File Format*. August 2019. ISBN 978-1079481594.

◇ Martin Ruckert
Hochschule München
Lothstrasse 64
80336 München
Germany
`ruckert (at) cs dot hm dot edu`

The Unreasonable Effectiveness of Pattern Generation

Petr Sojka and Ondřej Sojka

Abstract

Languages are constantly evolving organisms, and so are the hyphenation rules and needs. The effectiveness and utility of TEX's hyphenation have been proven by its usage in almost all typesetting systems in use today. The current Czech hyphenation patterns were generated in 1995 and no hyphenated word database is freely available.

We have developed a new Czech word database and have used the `patgen` program to efficiently generate new effective Czech hyphenation patterns and evaluated their generalization qualities. We have achieved almost full coverage on the training dataset of 3,000,000 words and validated the patterns on the testing database of 105,000 words approved by the Czech Academy of Science linguists.

Our pattern generation case study exemplifies an effective solution of widespread dictionary problem. The study has proved the versatility, effectiveness and extensibility of Liang's approach to hyphenation developed for TEX. The unreasonable effectiveness of pattern technology has lead to applications that are and will be used even more widely even 40 years after its inception.

> ... the best approach appears to be to embrace the complexity of the domain and address it by harnessing the power of data: if other humans engage in the tasks and generate large amounts of unlabeled, noisy data, new algorithms can be used to build high-quality models from the data. (Peter Norwig, [7])

1 Introduction

In their famous essays, Wigner [18], Hamming [1] and Norwig [7] consider mathematics and data-driven approaches miraculously, unreasonably effective. One of the very first mathematically founded approaches that harnessed the power of data was Franklin Liang's language-independent solution for TEX's hyphenation algorithm [6] and his program `patgen` for generation of hyphenation pattern from a word list.

Dictionary problem The task at hand was the *dictionary problem*. A dictionary can be seen as a database of records; in each record, we distinguish the key part (the word) and the data part (its division). Given the already hyphenated word list of a language, a set of *patterns* is magically generated. Language hyphenated patterns are much smaller than original word list and typically encode almost all hyphenation points in input list without mistakes. Liang's pattern approach thus could be viewed as an efficient lossy or lossless *compression* of hyphenated dictionary with several orders of magnitude compression ratio.

It has been proved [14, chapter 2] that optimization problem of exact lossless pattern minimization is non-polynomial by reduction to the minimum set cover problem.

Generated patterns have minimal length, e.g., shortest context possible, which results in their *generalization* properties. Patterns could hyphenate words not seen in the learning phase by analogy: yet another miracle of the generated patterns.

Pattern preparation During 36 years of `patgen`, there were hundreds of hyphenation patterns created, either by hand or *generated* by program `patgen`, or by the combination of both methods [8]. The advantage of *pattern generation* is that one can fine-tune pattern qualities for specific usage. Having an open-source and maintained word list adds another layer of flexibility and usability to the deployment of patterns. This approach is already set up for German variants and spellings [5], and was an inspiration for doing the same for the Czech language.

In this paper, we report on the development of the new Czech word list with a free license and complementary sets of hyphenation patterns. We describe the iterative process of initial word list preparation, word form collection, estimation of pattern generation parameters, and novel applications of the technology.

> Hyphenation is neither anarchy nor the sole province of pedants and pedagogues. Used in moderation, it can make a printed page more visually pleasing. If used indiscriminately, it can have the opposite effect, either putting the reader off or causing unnecessary distraction. (Major Keary)

2 Initial word list preparation

As a rule of thumb, the development of new big hyphenated word list starts on small data set first. The experience and outputs from this initial phase, e.g., hyphenation patterns, are applied to the bigger and bigger ones.

Bootstrapping idea As word lists of a well-established language are sizeable, and manual creation of huge hyphenated word list is tedious work, we used the bootstrapping technique. We illustrate the process of initial word list preparation by the diagram on Figure 1 on the following page. We have obtained a hyphenated word list with 105,244 words from the Czech Academy of Sciences, Institute of

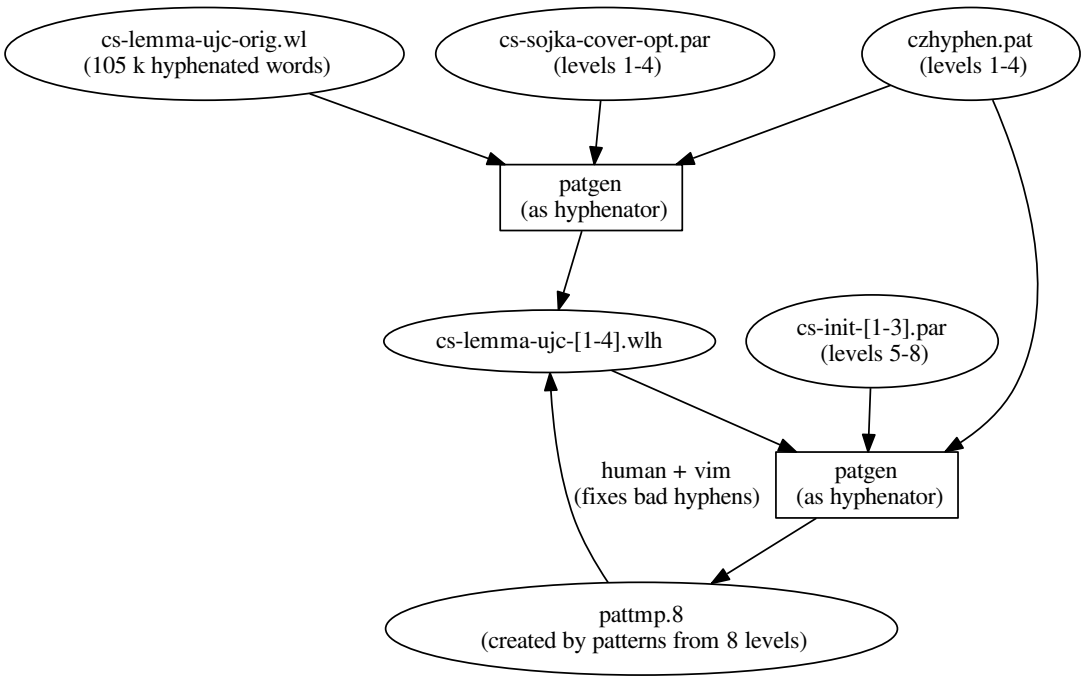


Figure 1: Life cycle of initial word list preparation, illustrated on the development of 105 k Czech consistently hyphenated words. `czhyphen.pat` represents the original Czech hyphenation patterns from [15] and `cs-sojka-cover-opt.par` are correct optimized `patgen` parameters from the same paper. `cs-init-[1-3].par` are custom parameters that trade off bad hyphens (which have to be manually checked) for missed hyphens. Information on which hyphenations `patgen` missed and where it wrongly put a hyphen is sourced from `pattmp`.

the Czech Language (ÚJČ). After a closer inspection, we have discovered many problems with the data, probably stemming from the fact that it has been crafted by multiple linguists and over the years. The few hyphenation rules [2] that are in the Czech language are not getting applied consistently there. The borderline cases were typically between syllabic (ro-zum) and etymological variants (roz-um) of hyphenation, or the way how to handle words borrowed from German or English into Czech.

It is impractical to try and manually find inconsistencies and systemic errors, even in a relatively short word list like this. We slightly modified and extended the process suggested in [13, page 242]: We used `patgen` and the current Czech patterns to hyphenate the word list and manually checked only the 25,813 words where the proposed hyphenation points differed from the official (were bad or missed), creating new word list `cs-lemma-ujc-1.wlh` [11] in the process.

But we are erroneous humans making mistakes. To find these, we have used `patgen` to generate the four additional levels of hyphenation patterns on top of the current patterns from the checked word list. We have also adjusted the parameters, see `cs-init-[1-3].par` [11] used for generation of the four additional levels to trade off bad hyphens (which have to be manually checked) for missed ones. We have then used these patterns, with eight levels in total, to hyphenate the checked word list and manually rechecked the wrongly hyphenated points (dots in `patgen` output), with missed hyphenation point (implicitly marked as the hyphen sign in hyphenated word list). We have repeated this process three times, iterating on `cs-lemma-ujc-[2-4].wlh`. The word list number four, referred to as the 'small one', is used for generation of bootstrapping patterns and final pattern validation.

3 Word list preparation and design

Any live language constantly changes, and Czech is no exception. Many new Czech words now come from

other languages, mostly from English. This presents a challenge for the patterns; they must not only correctly hyphenate Czech words according to Czech syllabic boundaries, but foreign words must be hyphenated correctly too, according to their new Czech syllabic pronunciation. [12] To have the patterns keep up with language evolution, we must maintain not only the patterns, but also a hyphenation word list. In this section, we will detail how we have built such a word list.

csTenTen corpus We have first obtained a word list with frequencies, generated from the Czech Web Corpus of TenTen family (csTenTen) [3]. We then filtered this word list to include only words that are present more than ten times in two crawls [17] made in years 2012 and 2017. We ended up with word list containing 922,216 words, non-negligible part of which are misspellings and jargon.

Word list cleanup We have then cleaned this word list by using the Czech morphological analyzer `majka` [10] to remove all words not known to it. We removed 370,291 typos, misspellings, and similar atypical lexemes and kept only 551,925 frequently occurring valid words in the dataset.

Word list expansion The morphological analyzer `majka` [10] also allows us to expand words into all their flexive forms as shown on Figure 2. We chose not to use the expansion feature of `majka` because the word list would grow to 3,779,379 (almost a fourfold increase) and csTenTen already contains most of the commonly used forms. It would also distort which hyphenation `patgen` gives most weight to. We supply word frequencies from csTenTen to the word list, so one can give higher weight to patterns that cover the most common words, eventually.

We expanded the word list with `majka` by adding 54,569 base forms of words that were present in the word list, but not in their base form. This increased the size to 606,494 words.

abdominální: abdominální, abdominálních, abdominálního, abdominálním, abdominálníma, abdominálními, abdominálnímu, neabdominální, neabdominálních, neabdominálního, neabdominálním, neabdominálníma, neabdominálními, neabdominálnímu

Figure 2: One lemma, expanded into all its lexemes by `majka`.

Sustainability The German *wortliste* [5] project served as inspiration for our open word list format, detailed in the `README.md` [11].

> One must regard the hyphen as a blemish to be avoided wherever possible. (Winston Churchill)

4 Bootstrapping — iterative development of hyphens in the big word list

It would be very tedious to manually hyphenate such a big word list by hand, so we train patterns on the small one and apply them on the big word list, as illustrated in Figure 3 on the following page. Then, we train patterns on the (now hyphenated) big word list and have `patgen` show what it would have hyphenated differently. With this approach, we cherry picks inconsistencies in the word list.

Since the big word list contains not only lemmas (base forms) of words, but also common inflections, we use regular expressions to add hyphens around them and fix inconsistencies. We keep iterating on this, as shown in Figure 3 on the next page, until patterns, generated with `cs-init-[1-3].par` [11], achieve nearly perfect coverage.

The resulting patterns hyphenate according to the standard Czech hyphenation rule: hyphenation is allowed everywhere where it does not change the pronunciation of the word. Thanks to the effectiveness of pattern generation, this works not only in Czech words but also foreign (Latin, French, German, English) ones.

> Hyphens, like cats, are capable of arousing tenderness or shudders. (Pamela Frankau)

5 Pattern generation

The last Czech hyphenation patterns were generated in 1995 [15], and are in use not only in TEX but also in other widespread typesetting systems. For conservative users there is no strong incentive for change, because the error rate is relatively low (the first version of validation set had about 4% error rate), and coverage is relatively high (the first version of validation set had around 7% missed hyphenation points).

Pattern generation from 3,000,000 words doesn't take hours as it did two decades ago, but seconds, even on commodity hardware, which allows for rapid development of "home-made" patterns.

We have developed a Python wrapper for `patgen` that we use in Jupyter notebooks. It allows quick iteration and easy sharing of results — see Table 1 on page 905 or `demo.ipynb` [11].[1]

It has also become common to use a validation dataset to ensure generalization abilities. Our usage

[1] In this preprint version we do not report final Czech hyphenation patterns yet, because we are still iterating on the big wordlist. We will include final statistics in the journal version.

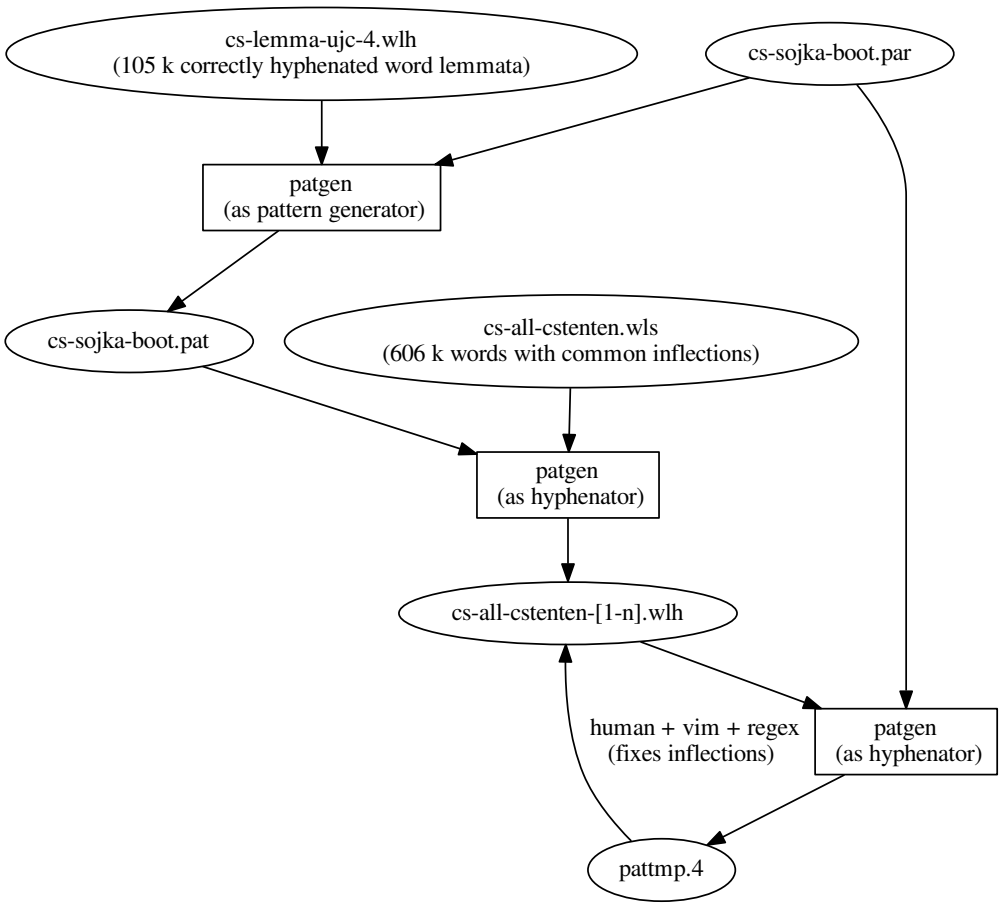


Figure 3: How we bootstrapped hyphenation of the big word list by training patterns (`cs-sojka-boot.pat`) on the small word list and applying them on the big one. `cs-sojka-boot.par` are `patgen` parameters that are designed to generate many patterns but still retain their generalization properties. `pattmp` highlights which hyphenation points in the source file the new pattern level missed, which were correctly covered and where they wrongly put a hyphen.

of a validation dataset has proved useful. Table 2 shows that if we were to use the *correct optimized* parameters from [16] that have been in use for Czech, we would overfit the training dataset and perform *worse* than their *size optimized* counterparts.

We believe that the patterns could be developed and serve as lossless compression of wordlist dataset, thus maximize the effectiveness of pattern technology.

> Life is the hyphen between matter and spirit.
> (Augustus William Hare)

6 The unreasonable effectiveness

We were able to solve the dictionary problem for Czech hyphenation effectively.

Space effectiveness From 3,000,000+ hyphenated words stored in approximately 30,000,000 bytes we have produced patterns of size 30,000 bytes, achieving roughly 1000× space *lossless* compression.

Time effectiveness Using the trie data structure for patterns makes the time complexity of accessing the record related to the word, e.g., hyphenation point, in very low *constant* time. The constant is adequate to the depth of the pattern trie data structure, e.g., 5 or 6 in the case of Czech. In the case, the whole pattern trie resides in RAM, the time for finding the patterns for a word is on the scale of tens, at most hundreds of single processor instructions. Word hyphenation throughput is then about 1,000,000 words per second on a modern CPU.

Table 1: Outputs from running `patgen` in our Jupyter notebook with two different parameter sets. The first parameter set is from the German Trennmuster project [5] and generates 7,291 patterns, 40 kB. The second one from [16] generates shorter and smaller patterns — 4,774 patterns, 25 kB.

Level	Patterns	Good	Bad	Missed	Lengths	Params
1	750	1,683,529	525,670	0	1 5	1 1 1
2	3,178	1,628,874	38	54,655	2 6	1 2 1
3	2,548	1,683,528	9,931	1	3 7	1 1 1
4	1,382	1,683,287	0	242	4 8	1 4 1
5	92	1,683,528	0	1	5 9	1 1 1
6	0	1,683,528	0	1	6 10	1 6 1
7	1	1,683,529	0	0	7 11	1 4 1

Level	Patterns	Good	Bad	Missed	Lengths	Params
1	1,608	1,655,968	131,481	27,561	1 3	1 5 1
2	1,562	1,651,840	2,533	31,689	1 3	1 5 1
3	2,102	1,683,528	2,584	1	2 5	1 3 1
4	166	1,683,135	6	394	2 5	1 3 1

Table 2: A comparison of validation scores of patterns trained on the big (606 k words) wordlist with different parameters.

Params	Good	Bad	Missed	Size	Patterns
correctopt [16]	99.41 %	3.47 %	0.59 %	25 kB	4,774
german [5]	99.40 %	3.33 %	0.60 %	40 kB	7,291

Optimality Even though finding exact space and time-optimal solutions is not feasible, finding an approximate solution close to optimum is possible. Heuristics and insight expressed above, together with Jupyter notebook interactive fine-tuning of `patgen` parameter options allows for rapid pattern development.

Automation A close-to-optimal solution to the dictionary problem could be useful not only for Czech hyphenation, but for all other languages [9, 8], and more generally, for other instances of the dictionary problem. Developing heuristics for thresholding of `patgen` pattern generation parameters, based on a statistical analysis of big input, data could allow the deployment of presented approaches on a much broader problem set and scale. We believe that parameters could be automatically approximated from the statistics of the input data.

Pattern generation — in Wigner terms — "has proved accurate beyond all reasonable expectations". Let us paraphrase another one of his quotes:

> The miracle of the appropriateness of the language of ~~mathematics~~ *patterns* for the formulation of the laws of ~~physics~~ *data* is a wonderful gift which we neither understand nor deserve. We should be grateful for it and hope that it will remain valid in future research and that it will extend, for better or for worse, to our pleasure, even though perhaps also to our bafflement, to wide branches of learning.

> "We should stop acting as if our goal is to author extremely elegant theories, and instead embrace complexity and make use of the best ally we have: the unreasonable effectiveness of data." (Peter Norvig, [7])

7 Conclusion

We have developed a flexible open language–independent system [11] for hyphenation pattern generation. We have demonstrated the effectiveness of this system by updating the old Czech hyphenation patterns [15] and achieving record accuracy. We have also applied recent data and computer science advancements, like the usage of interactive Jupyter notebooks and a validation dataset to prevent overfitting, to the more than three decades old problem of pattern generation.

Future work

Word lists for other languages Logical next steps will be applying developed techniques for different languages: for Slovak and virtually all that does not yet have word list based hyphenation pattern generation and word list either in Sketch Engine or elsewhere are available.

Stratification Pattern generation could be further speed up by several techniques like stratification of word list on the level of input or on the level of counting pros and cons examples to include a new pattern or not.

Pattern-encoded spellchecker We have a big dictionary of frequent spelling errors from csTenTen word list. Nothing prevents us from encoding them into specific patterns or pattern layers with extra levels and used that information during typesetting, e.g., to typeset those words with red underlining in LuaTEX. LuaTEX allows dynamic pattern loading and Lua programming that will enable the implementation of this feature, which people are used to using in editors.

Pattern-based learnable key memories Solutions to versions of dictionary problem are a hot topic of leading-edge research to design memory data architectures like those used in a machine learning of language [4]. Pattern-based memory network architectures could speed up language data access in big memory neural networks considerably.

Acknowledgements

We owe our gratitude to the whole bunch of people: to Vít Suchomel of Lexical Computing for word lists from Sketch Engine, to Pavel Šmerk for `majka` and paper proofreading, to Don Knuth and Frank Liang for TEX and `patgen`, and to Vít Novotný for paper proofreading.

References

[1] R. W. Hamming. The Unreasonable Effectiveness of Mathematics. *The American Mathematical Monthly* 87(2):81–90, 1980. `http://www.jstor.org/stable/2321982`

[2] Internetová jazyková příručka (Internet Language Reference Book). `http://prirucka.ujc.cas.cz/?id=135`

[3] M. Jakubíček, A. Kilgarriff, et al. The TenTen Corpus Family. In *Proc. of 7th International Corpus Linguistics Conference (CL)*, pp. 125–127, Lancaster, July 2013.

[4] G. Lample, A. Sablayrolles, et al. Large Memory Layers with Product Keys, 2019. `https://arxiv.org/pdf/1907.05242`

[5] W. Lemberg. A database of German words with hyphenation information. `https://repo.or.cz/wortliste.git`

[6] F. M. Liang. *Word Hy-phen-a-tion by Com-put-er*. PhD thesis, Department of Computer Science, Stanford University, Aug. 1983.

[7] F. Pereira, P. Norvig, and A. Halevy. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems* 24(02):8–12, Mar. 2009. `doi:10.1109/MIS.2009.36`

[8] A. Reutenauer and M. Miklavec. TEX hyphenation patterns. `https://www.tug.org/tex-hyphen/`

[9] K. P. Scannell. Hyphenation patterns for minority languages. *TUGboat* 24(2):236–239, 2003.

[10] P. Šmerk. Fast Morphological Analysis of Czech. In P. Sojka and A. Horák, eds., *Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2009*, pp. 13–16, Karlova Studánka, Czech Republic, Dec. 2009. Masaryk University. `http://nlp.fi.muni.cz/raslan/2009/`

[11] O. Sojka and P. Sojka. cshyphen repository. `https://github.com/tensojka/cshyphen`

[12] P. Sojka. Notes on Compound Word Hyphenation in TEX. *TUGboat* 16(3):290–297, 1995.

[13] P. Sojka. Hyphenation on Demand. *TUGboat* 20(3):241–247, 1999. `tug.org/TUGboat/tb20-3/tb64sojka.pdf`.

[14] P. Sojka. *Competing Patterns in Language Engineering and Computer Typesetting*. PhD thesis, Masaryk University, Brno, Jan. 2005.

[15] P. Sojka and P. Ševeček. Hyphenation in TEX — Quo Vadis? *TUGboat* 16(3):280–289, 1995.

[16] P. Sojka and P. Ševeček. Hyphenation in TEX — Quo Vadis? In M. Goossens, ed., *Proceedings of the TEX Users Group 16th Annual Meeting, St. Petersburg, 1995*, pp. 280–289, Portland, Oregon, U.S.A., 1995. TEX Users Group.

[17] V. Suchomel and J. Pomikálek. Efficient web crawling for large text corpora. In A. Kilgarriff and S. Sharoff, eds., *Proc. of the seventh Web as Corpus Workshop (WAC)*, pp. 39–43, Lyon, 2012. `http://sigwac.org.uk/raw-attachment/wiki/WAC7/wac7-proc.pdf`

[18] E. P. Wigner. The Unreasonable Effectiveness of Mathematics in the Natural Sciences. Richard Courant Lecture in Mathematical Sciences delivered at New York University, May 11, 1959. *Communications on Pure and Applied Mathematics* 13(1):1–14, 1960. doi:10.1002/cpa.3160130102

◇ Petr Sojka
The Faculty of Informatics at Masaryk University
Brno, Czech Republic and $\mathcal{CS}$TUG
sojka (at) fi dot muni dot cz
https://www.fi.muni.cz/usr/sojka/

◇ Ondřej Sojka
$\mathcal{CS}$TUG
Brno, Czech Republic
ondrej.sojka (at) gmail dot com

Quickref: a Stress Test for Texinfo

Didier Verna

Abstract

Quickref is a global documentation project for the Common Lisp ecosystem. It creates reference manuals automatically by introspecting libraries and generating a corresponding documentation in Texinfo format. The Texinfo files may subsequently be converted into PDF or HTML. Quickref is non-intrusive: software developers do not have anything to do to get their libraries documented by the system.

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Common Lisp ecosystem at once. The result is a website containing almost two thousand reference manuals. Quickref provides a Docker image for an easy recreation of this website, but a public version is also available and actively maintained.

Quickref constitutes an enormous and successful stress test for Texinfo. In this paper, we give an overview of the design and architecture of the system, describe the challenges and difficulties in generating valid Texinfo code automatically, and put some emphasis on the currently remaining problems and deficiencies.

1 Introduction

Lisp is a high level, general purpose, multi-paradigm programming language created in 1958 by John McCarthy[2]. We owe to Lisp many of the programming concepts that are still considered as fundamental today (functional programming, garbage collection, interactive development *etc.*). Over the years, Lisp evolved as a family of dialects (including Scheme, Racket, and Clojure, to name a few) rather than as a single language. Another Lisp descendant of notable importance is Common Lisp, a language targeting the industry, which was standardized in 1994[5].

The Lisp family of languages is mostly known for two of its most prominent (and correlated) characteristics: a minimalist syntax and a very high level of expressiveness and extensibility. The root of the latter, right from the early days, is the fact that code and data are represented in the same way (a property known as *homoiconicity*[3, 1]). This makes meta-programming not only possible but also trivial. Being a Lisp, Common Lisp not only maintains this property, but also provides an unprecedented arsenal of paradigms making it much more expressive and extensible than its industrial competitors such as C++ or Java.

Interestingly enough, the technical strengths of the language come with serious drawbacks community-wise (a phenomenon also affecting other dialects, such as Scheme). These problems are known and have been discussed many times already[4, 7]. They may explain, at least partly, why in spite of its technical potential, the Lisp family of languages never really took over, and probably never will. The situation can be summarized as follows: Lisp usually makes it so easy to "hack" things away that every Lisper ends up developing his or her own solution, inevitably leading to a *paradox of choice*. The result systematically is a plethora of solutions for every single problem that every single programmer faces. Most of the time, these solutions work, but they are either half-baked or targeted to the author's specific needs and thus not general enough, it is difficult to assert their quality, and they are usually not (well) documented.

Being aware of the situation, the community has been attempting to "consolidate" itself in various ways. Several websites aggregate resources related to the language or its usage (books, tutorials, implementations, development environments, applications, *etc.*). The Common Lisp Foundation (`https://cl-foundation.org/`) provides technical (sometimes even financial) support and infrastructure for project authors. Once a year, the European Lisp Symposium (`https://www.european-lisp-symposium.org`) gathers the international community, equally opening to researchers and practitioners, newcomers and experts.

From a more technical standpoint, solving the paradox of choice, that is, deciding on official solutions for doing this or that is much more problematic, mainly because there is no such thing as an official authority in the community. On the other hand, some libraries do impose themselves as *de facto* standards. Two of them are worth mentioning here. Most non-trivial Common Lisp packages today use ASDF for structuring themselves. ASDF allows you to define your package architecture in terms of source files and directories, dependencies and other metadata. It automates the process of compiling and loading (dependencies included). The second one is Quicklisp (`https://www.quicklisp.org`). Quicklisp is both a central repository for Common Lisp libraries (not unlike CTAN) and a programmatic interface for it. With Quicklisp, downloading, installing, compiling and loading a specific package on your machine (again, dependencies included) essentially becomes a one-liner.

One remaining problem is that of documentation. Of course, it is impossible to force a library

```
(asdf:defsystem :net.didierverna.declt
  :long-name "Documentation Extractor from Common Lisp to Texinfo"
  :description "A reference manual generator for Common Lisp libraries"
  :author "Didier Verna"
  :mailto "didier@didierverna.net"
  :homepage "http://www.lrde.epita.fr/~didier/software/lisp/"
  :source-control "https://github.com/didierverna/declt"
  :license "BSD"
  ...)
```

Figure 1: ASDF system definition excerpt

author to properly document his or her work. One could consider writing the manuals they miss for the third-party libraries they use, but this never happens in practice. There is still something that we can do to mitigate the issue, however. Because Common Lisp is highly reflexive, it is relatively straightforward to retrieve the information necessary to automatically create and typeset *reference* manuals (as opposed to *user* manuals). Several such projects exist already (remember the paradox of choice). In this paper we present our own, probably the most complete Common Lisp documentation generator to date.

Enter Quickref...

2 Overview

Quickref is a global documentation project for the Common Lisp ecosystem. It generates reference manuals for libraries available in Quicklisp automatically. Quickref is non-intrusive, in the sense that software developers do not have anything to do to get their libraries documented by the system: mere availability in Quicklisp is the only requirement. In this section, we provide a general overview of the system's features, design, and implementation.

2.1 Features

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Quicklisp world at once, which means that almost two thousand reference manuals are generated. Creating a local documentation website can be done in two different ways: either by using the provided Docker image (the most convenient solution for an exhaustive website), or directly via the programmatic interface, from within a running Lisp environment (when only the documentation for the local, partial, installation is required). If you don't want to run Quickref yourself, a public website is also provided and actively maintained at `quickref.common-lisp.net`. It always contains the result of a full run of the system on the latest Quicklisp distribution.

2.2 Documentation Items

Reference manuals generated by Quickref contain information collected from various sources. First of all, many libraries provide a README file of some sort, which can make for a nice introductory chapter. In addition to source files and dependencies, ASDF offers ways to specify project-related metadata in the so-called *system definition* form. Figure 1 illustrates this. Such information can be easily (programmatically) retrieved and used. Next, Lisp itself has some built-in support for documentation, in the form of so-called *docstrings*. As their name suggests, docstrings are (optional) documentation strings that may be attached to various language constructs such as functions, variables, methods and so on. Figure 3 provides an example. When available, docstrings greatly contribute to the completeness of reference manuals, and again, may be retrieved programmatically through a simple standard function call.

```
(defmacro @defconstant (name &body body)
  "Execute BODY within a @defvr {Constant}.
NAME is escaped for Texinfo prior to rendering.
BODY should render on *standard-output*."
  `(@defvr "Constant" ,name ,@body))
```

Figure 3: Common Lisp docstring example

As for the rest, the solution is less straightforward. We want our reference manuals to advertise as many software components as possible (functions, variables, classes, packages *etc.*). In general there are two main strategies for collecting that kind of information.

Code Walking The first one, known as *code walking*, consists in statically analyzing the source code. A code walker is usually at least as complicated as the syntax of the target language, because it requires a parser for it. Because of Lisp's minimalist syntax, using a code walker is a very tempting solution. On the other hand, Lisp is extremely dynamic in nature, meaning that many of the final program's

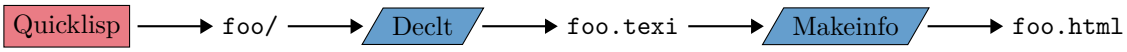


Figure 2: Quickref pipeline (■ Main thread, ▱ External Process)

components may not be directly visible in the source code. On top of that, programs making syntactic extensions to the language would not be directly parsable. In short, it is practically impossible to collect all the required information by code walking alone. Therefore, we do not use that approach.

Introspection Our preferred approach is by *introspection*. Here, the idea is to actually compile and load the libraries, and then collect the relevant information by inspecting the memory. As mentioned before, the high level of reflexivity of Lisp makes introspection rather straightforward. This approach is not without its own drawbacks however. First, actually compiling and loading the libraries requires that all the necessary (possibly foreign) components and dependencies are available. This can turn out to be quite heavy, especially when the two thousand or so Quicklisp libraries are involved. Secondly, some libraries have platform, system, compiler, or configuration-specific components that may or may not be compiled and loaded, depending on the exact conditions. If such a component is skipped by our system, we won't see it and hence we won't document it. We think that the simplicity of the approach by introspection greatly compensates for the risk of missing a software component here and there. That is why introspection is our preferred approach.

2.3 Toolchain

Figure 2 depicts the typical reference manual production pipeline used by Quickref , for a library named `foo`.

1. Quicklisp is first used to make sure the library is installed upfront, which results in the presence of a local directory for that library.
2. Declt (`https://www.lrde.epita.fr/~didier/software/lisp/misc.php#declt`) is then run on that library to generate the documentation. Declt is another library of ours, written 5 years before Quickref, but with that kind of application in mind right from the start. In particular, it is for that reason that the documentation generated by Declt is in Texinfo intermediate format.
3. The Texinfo file is finally processed into HTML. Texinfo (`https://www.gnu.org/software/texinfo/`) is the GNU official documentation format. There are three main reasons why this format was chosen when Declt was originally written. First, it is particularly well suited to technical documentation. More importantly, it is designed as an abstract, intermediate format from which human-readable documentation can in turn be generated in many different forms (PDF and HTML notably). Finally, it includes very convenient built-in anchoring, cross-referencing, and indexing capabilities.

Quickref essentially runs this pipeline on the required libraries. Some important remarks need to be made about this process.

1. Because Declt works by introspection, it would be unreasonable to load almost two thousand libraries in a single Lisp image. For that reason, Quickref doesn't actually run Declt directly, but instead forks it as an external process.
2. Similarly, `makeinfo` (`texi2any` in fact), the program used to convert the Texinfo files to HTML, is an external program written in Perl (with some parts in C), not a Lisp library. Thus, here again, we fork a `makeinfo` process out of the Quickref Lisp instance in order to run it.

2.4 Performance

Experimental studies have been conducted on the performance of the system. There are different scenarios in which Quickref may run, depending on the exact number of libraries involved, their current state, and the level of required "isolation" between them. All the details are provided in [6], but in short, there is a compromise to be made between the execution time and the reliability of the result. We found that for a complete sequential run of the system on the totality of Quicklisp, the most frequent scenario takes around 2 hours on our test machine, whereas the safest one requires around seven hours.

In order to improve the situation, we recently added support for parallelism to the system. The upgraded architecture is depicted in Figure 4. In this new processing scheme, an adjustable number of threads is devoted to generating the Texinfo files in parallel. In a second stage, an also adjustable number of threads is in charge of picking the Texinfo files as they come, and creating the corresponding HTML versions. A specific scheduling algorithm (not unlike that of the `make` program) delivers libraries in an order, and at a time suitable to parallel processing by the Declt threads, avoiding any concurrency problems. With this new architecture in place, we were

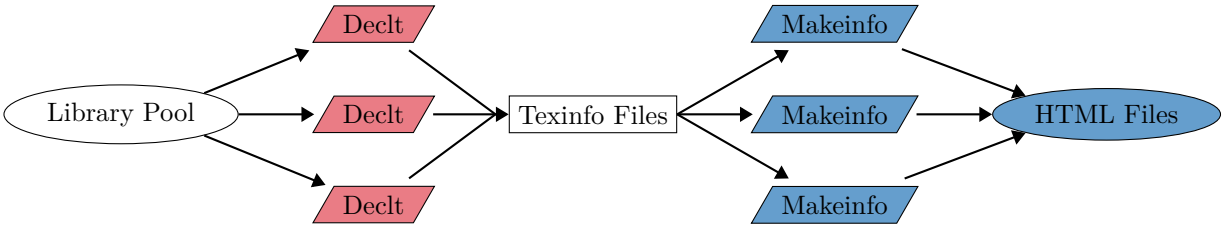


Figure 4: Quickref parallel processing (Declt thread, Makeinfo thread)

able to cut the processing time by a factor of four, reducing the worst case scenario to 1h45 and the most frequent one to half an hour. These numbers make it reasonable to run Quickref on one's local machine again.

3 Challenges

Quickref is a challenging project in many regards. Two thousand libraries is *a lot* to process. Setting up the environment necessary to properly compile and run those libraries is not trivial, especially because many of them have platform or system-specific code and require foreign dependencies. Finally, Quickref constitutes a considerable (and successful) stress test for Texinfo. The Texinfo file sizes range from 7Ko to 15Mo (make it double for the generated HTML ones). The number of lines of Texinfo code in those files extends from 364 to 285 020, the indexes may contain between 14 and 44 500 entries, and the processing times vary from 0.3s to 1m 38s per file.

Challenges related to the project scalability and performance have been described previously[6]. This section focuses on more general or typesetting / Texinfo-oriented ones.

3.1 Metadata Format Underspecification

One difficulty in collecting metadata is that their format is often underspecified, or not specified at all, as is the case with ASDF system ones. To give just a single example, Figure 5 lists several of the possible values we found for the author metadata. As you can see, most programmers use strings, but the actual contents vary greatly (single or multiple names, email addresses, middle letter, nicknames, *etc.*), and so does the formatting. For the anecdote, we found one attempt at pretty printing the contents of the string with a history of authors, and one developer even went as far as concealing his email address by inserting Lisp code into the string itself…

It would be unreasonable to even just try to understand all these formats (what others will we discover in the future?), so we remain somewhat strict in what we recognize — in that particular case, strings of the form `"author[ <email>]"`, or a list of such. The Declt user manual has a Guidelines section with some advice for library authors that would accept to be friendlier with our tool. We cannot force anyone to honor our guidelines however.

```
"Didier Verna"
"Didier Verna <didier@lrde.epita.fr>"
"Didier Verna didier@lrde.epita.fr"
"didier@lrde.epita.fr"
"<didier@lrde.epita.fr>"
"Didier Verna and Antoine Martin"
"Didier Verna, Antoine Martin"
"Didier Verna Antoine Martin"
"D. Verna Antoine E Martin"
"D. Verna Antoine \"Joe Cool\" Martin"
("Didier Verna" "Antoine Martin")
"
     Original Authors:
       Salvi Péter,
       Naganuma Shigeta,
       Tada Masashi,
       Abe Yusuke,
       Jianshi Huang,
       Fujii Ryo,
       Abe Seika,
       Kuroda Hisao
     Author Post MSI CLML Contribution:
       Mike Maul  <maul.mike@gmail.com>"
"(let ((n \"Christoph-Simon Senjak\")) ~
          (format nil \"~A <~C~C~C~C~A>\" ~
            n (elt n 0) (elt n 10) (elt n 16) ~
            #\\@ \"uxul.de\"))"
```

Figure 5: ASDF author metadata

On the other hand, Quickref has an interesting social effect that we particularly noticed the first time the public website was released. In general, people don't like *our* documentation for *their* work to look bad, especially when it is publicly available. In the first few days following the initial release and announcement of Quickref, we literally got dozens of reports related to typesetting glitches. Programmers *rushed* to the website in order to see what *their* library looked like. In the cases were the bugs weren't on our part, many of the concerned authors were hence willing to slightly bend their own coding style, in order for *our* documentation to look better. We still count on that social effect.

3.2 Definitions Grouping

Rather than just providing a somewhat boring list of functions, variables, and other definitions, as reference manuals do, Declt attempts to improve the presentation in different ways. In particular, it makes sense to group related definitions together when possible.

A typical example of this is when we need to document accessors (readers and writers to the same information). It makes sense to group these definitions together, provided that their respective docstrings are either nonexistent, or exactly the same (this is one of the incentives given to library authors in the Declt's user manual Guidelines section). This is exemplified in Figure 6. Another typical example consists in listing methods (in the Object-Oriented sense) within the corresponding generic function's entry.

`context-hyperlinksp` *CONTEXT* [Function]
`(setf context-hyperlinksp)` *BOOL CONTEXT* [Function]
Access *CONTEXT*'s hyperlinksp flag.

Package [`net.didierverna.declt`], page 29,
Source [`doc.lisp`], page 24, (file)

Figure 6: Accessors definitions grouping

Texinfo provides convenient macros for defining usual programming language constructs (`@defun`, `@defvar`, *etc.*), and "extended" versions for adding sub-definitions (`@defunx`, `@defvarx`, *etc.*). Unfortunately, definitions grouping prevents us from using them, for several reasons.

1. Nesting `@def...` calls would lead to undesirable indentation.
2. Heterogeneous nesting is prohibited. For example, it is not possible use `@defvarx` within a call to `@defun` (as surprising as it may sound, such kind of heterogeneous grouping makes sense in Lisp).

On the other hand, that kind of thing is possible with the lower-level (more generic) macros, as heterogeneous *categories* become simple macro arguments. One can, for example use the following (which we frequently do):

```
@deffn {Function} ...
@deffnx {Compiler Macro} ...
...
@end deffn
```

This is why we stick to those lower-level macros, at the expense of re-inventing some of the higher-level built-in functionality.

Even with this workaround, some remaining limitations still get in our way.

1. There are only nine canonical categories and it is not possible to add new ones (at least not without hacking Texinfo's internals).
2. Although we understand the technical reasons for it (parsing problems, probably), some of the canonical categories are arguable. For example, the distinction between typed and untyped functions makes little sense in Common Lisp which has optional static typing. We would prefer to have a single function definition entry point handling optional types.
3. Heterogeneous mixing of the lower-level macros is still prohibited. For example, it remains impossible to write the following (still making sense in Lisp):

```
@deffn {Function} ...
@defvrx {Symbol Macro} ...
...
@end deffn
```

3.3 Pretty Printing

Pretty printing is probably the biggest challenge in typesetting Lisp code, because of the language's flexibility. In particular, it is very difficult to find the right balance between readability and precision.

Identifiers In Lisp, identifiers can be basically *anything*. When identifiers contain characters that are normally not usable (*e.g.* blanks or parenthesis), the identifier must be escaped with pipes. In order to improve the display of such identifiers, we use several heuristics.

- A symbol containing blank characters is normally escaped like this: `|my identifier|`. Because the escaping syntax doesn't look very nice in documentation, we replace blank characters with more explicit Unicode ones, for instance `my␣identifier`. We call this technique "revealing". Of course, if one identifier happens to contain one of our revealing characters already, the typesetting will be ambiguous. The case should be extremely rare though.
- In some situations on the other hand, it is actually better to *not* reveal the blank characters. The so-called *setf* (setter / writer) functions are such an example. Here, the identifier is in fact composed of several symbols, such as in `(setf this)`. Revealing the whitespace character would only clutter the output, so we leave it alone.
- Finally, some unusual identifiers that are normally escaped in Lisp, such as `|argument(s)|`,

do not pose any readability problems in documentation, so we just typeset them without the escaping syntax.

Qualification Another issue is that of symbol *qualification*. With one exception, symbols in Lisp belong to a *package* (more or less the equivalent of a namespace). Many Lispers use Java-style package names, which can be quite long. Typesetting a fully qualified symbol would give something like that: `my.long.package.name:symbol`. Lisp libraries usually come with their own very few packages, so typesetting a reference manual with thousands of symbols fully qualified with the same package name would look pretty bad. Because of that, we avoid typesetting the package names in general. Unfortunately, if different packages contain eponymous symbols, this will lead to a confusing output. Currently, we don't have a satisfactory answer to this problem.

Docstrings The question of how to typeset docstrings is also not trivial. People tend to use varying degrees of plain-text formatting in them, with all kinds of line lengths, *etc.* Currently, we use only a very basic heuristic to determine whether an end of line in a docstring is really wanted here, or just a consequence of reaching the "right margin". We are also considering providing an option to simply display the docstrings verbatim, and on the long term, we plan to support markup languages such as Markdown.

References A Texinfo-related problem we have is that links are displayed differently, depending on the output format, and with some rather undesirable DWIM behavior. Table 1 shows the output of a call to `@ref{anchor, , label}` in various formats (`anchor` is the link's internal name, `label` is the desired output).

HTML	label
PDF	[label], page 12,
Info	*note label: anchor.
Emacs Info mode	See label.

Table 1: Texinfo links formatting in various output formats

In PDF, the presence of the trailing comma is context-dependent. In Info, both the label and the actual anchor name are typeset, which is very problematic for us (see Section 3.4). In Emacs Info mode, the casing of "See" seems to vary. In general, we would prefer to have more consistent output across the different formats, or at least, more control over it.

3.4 Anchoring

The final Texinfo challenge we want to address here is that of anchoring. In Texinfo, anchor names have severe limitations: dots, commas, colons, and parenthesis are explicitly forbidden (due to the final display syntax in Info). This is very unfortunate because those characters are extremely common in Lisp (parenthesis of course, but also dots and colons in the package qualification syntax).

Our original (and still current) solution is to replace those characters by a sequence such as `<dot>`. Of course, this makes anchor names particularly ugly, but we didn't think that was a problem because we have nicer *labels* to point to them in the output (in fact, labels have a less limited syntax, although this is not well documented). However, we later realized that anchor names still appear in the HTML output and also in pure Info. Consequently, we are now considering changing our escaping policy, perhaps by using Unicode characters as replacements, just like we already do on identifiers (see Section 3.3).

The second anchoring problem we have is that of Texinfo nodes, the fundamental document structuring construct. Nodes have two very strong limitations: their names must be unique and there is no control over the way they are displayed in the output. This is a serious problem for us because Lisp has a lot of different namespaces. A symbol may refer to a variable, a function, a class, and many other things at the same time. Consequently, if we were to use one node for each definition, we would need to mangle the node name in a way that would make it barely human-readable. Consequently, we originally decided not to do this, and avoid nodes as much as possible (somewhat of a paradox, given the importance of nodes in Texinfo). For example, our generated reference manuals have just one node entitled "Exported Functions", and the library's API (perhaps constituted of hundreds of functions) listed in there, with manual anchors instead of nodes for every public function.

While this works well for PDFoutput, we later realized that this makes indexing practically useless in other formats. In PDF, the indexes point to pages, which is as specific as it can get on (even virtual) paper. In formats which don't have page numbers however, such as Info and HTML, the indexes point to the menu entry containing the first node referenced. Basically, this means for instance that all indexes for the hundreds of public functions in a library point to the same location in the HTML reference manual, which is way too coarse to be usable.

Because of that, we are now considering changing our original policy with respect to nodes, and get back to associating every definition with one node, at the expense of having very long and ugly node names. It is our hope that one day, the node names uniqueness constraint in Texinfo be relaxed, perhaps disambiguating by using their hierarchical organization.

4 Conclusion and Perspectives

Although a relatively young project, Quickref is already quite successful. It is able to document almost two thousand Common Lisp libraries without any showstopper. Less than 2% of the Quicklisp libraries still pose problems and some of the related bugs have already been identified. The Common Lisp community seems generally grateful for this project.

Quickref also constitutes an enormous, and successful, stress test for Texinfo. Given the figures involved, it was not obvious how `makeinfo` would handle the workload, but it turned out to be very reliable and scalable. Although the design of Texinfo sometimes gets in our way, we still consider it a good choice for this project, in particular given the diversity of its output formats and its built-in indexing capabilities.

In addition to solving the problems described in this paper, the project also has much room for improvement left. In particular, the following are at the top level of our TODO list.

1. The casing problem needs to be addressed. Traditional Lisp is case-insensitive but internally upcases every symbol name (except for escaped ones). Several modern Lisps offer alternative policies with respect to casing. Quickref doesn't currently address casing problems at all (not even that of escaped symbols).
2. Our indexing policy could be improved. Currently, we only use the built-in Texinfo indexes (Functions, Variables, *etc.*) but we also provide one level of sub-indexing. For instance, macros appear in the function index, but they are listed twice: once as top level entries, and once under a Macro sub-category. The question of which amount of sub-indexing we want, and whether to create and use new kinds of indexes is under consideration.
3. Although our reference manuals are already stuffed with cross-references, we plan to add more. Because Declt was originally designed to generate one reference manual at a time, only internal cross-references are available. The existence of Quickref now raises the need for external cross-references (that is, between different manuals).
4. Many aspects of the pretty printing could still be improved, notably that of so-called "unreadable" objects and lambda lists.
5. In addition to HTML, we plan to provide PDF as well as Info files on the website, since they are readily available.
6. We intend to integrate Quickref with Emacs and Slime (a *de facto* standard Emacs-based development environment for Common Lisp). In particular, we want to give Emacs the ability to browse the Info reference manuals online or locally if possible, and provide Slime with commands for opening the Quickref documentation directly from Lisp source code displayed in Emacs buffers.
7. Finally, we are working on providing new index pages for the website. Currently, we have a library index and an author index. We are working on providing keyword and category indexes as well.

References

[1] A. C. Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.

[2] J. MacCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM* 3(4):184–195, Apr. 1960. doi:10.1145/367177.367199

[3] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM* 3:214–220, Apr. 1960. doi:10.1145/367177.367223

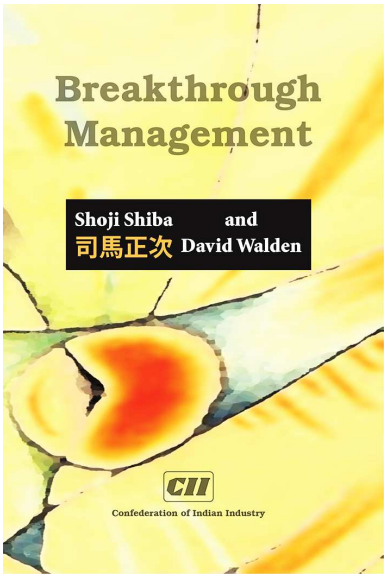
[4] M. Tarver. `http://www.marktarver.com/bipolar.html`, 2007.

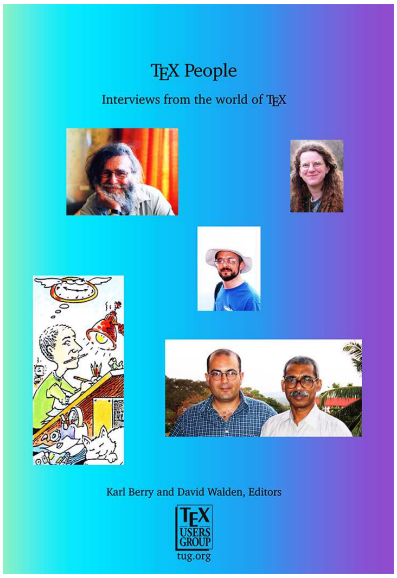
[5] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

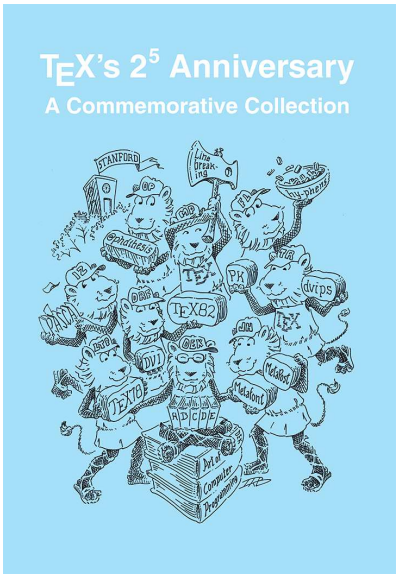
[6] D. Verna. Parallelizing quickref. In *12th European Lisp Symposium*, pp. 89–96, Genova, Italy, Apr. 2019. doi:10.5281/zenodo.2632534

[7] R. Winestock. The Lisp curse. `http://www.winestockwebdesign.com/Essays/Lisp_Curse.html`, Apr. 2011.

◇ Didier Verna
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre
France
didier (at) lrde.epita.fr
http://www.didierverna.info







Congratulations to the TeX Users Group
on the occasion of its 40th annual conference

Thirteenth ConTEXt user meeting

Bassenge, Belgium

September 16–21, 2019

`meeting.contextgarden.net`

GuIT 2019

Turin, Italy

October 26, 2019

`guitex.org/home/en/meeting`
