CSTUG

ZPRAVODAJ

Československého sdružení uživatelů TEXu

1–4

2018

ISSN 1211-6661 Tištěná verze

ISSN 1213-8185 Online verze

Ročník 28

OBSAH

Zpravodaj Československého sdružení uživatelů TeXu je vydáván v tištěné podobě a distribuován zdarma členům sdružení. Vydaná čísla Zpravodaje v elektronické podobě (PDF) jsou bezodkladně veřejně vystavena na webové adrese `https://www.cstug.cz/`.

Své příspěvky do Zpravodaje můžete zasílat v elektronické podobě, nejlépe jako jeden archivní soubor `(.zip, .arj, .tar.gz)`, na e-mailovou adresu `bulletin@cstug.cz`. Nezapomeňte přiložit všechny soubory, které dokument načítá (s výjimkou standardních součástí TeX Live), zejména v případě, kdy Vás nelze kontaktovat e-mailem.

ISSN 1211-6661 (tištěná verze)
ISSN 1213-8185 (online verze)

Příprava Zpravodaje CSTUG

Vít Novotný

Článek popisuje strukturu, sazbu a předtiskovou přípravu Zpravodaje CSTUG. Rozepsány jsou veškeré kroky na cestě rukopisu článku do schránek čtenářů. Autor je od roku 2016 technickým redaktorem Zpravodaje.

Klíčová slova: LaTeX, ConTeXt, barvy, Ghostscript, BibLaTeX, `biber`, XML

1. Úvod

Zpravodaj CSTUG je recenzovaný vědecký a odborný časopis s původními pracemi, který vychází od roku 1991. S přípravou časopisu souvisí množství kroků, počínaje úpravou recenzovaných článků a konče sesazením a rozesláním hotového čísla čtenářům. V tomto článku se budu věnovat jednotlivým krokům v takové míře detailu, aby je čtenář mohl využít při přípravě vlastních dokumentů.

Článek je strukturován následovně: V sekci 2 popíšu strukturu zpravodajového článku a dostupné šablony pro formáty LaTeX a ConTeXt. Tato sekce je tedy bezprostředně užitečná autorům, kteří se chystají svůj rukopis publikovat ve Zpravodaji. V sekci 3 popíšu strukturu čísla Zpravodaje, technické řešení jeho sazby a předtiskovou kontrolu. Nakonec nastíním problematiku převodu citací do strojově čitelné podoby pro zaslání registrační agentuře DOI.

2. Zpravodajové články

Ačkoliv mohou autoři zasílat články k recenznímu řízení jako neformátovaný rukopis, i ten by měl zachycovat požadovanou strukturu článku. Větší kontrolu nad výsledným formátováním pak autorům dávají poskytované TeXové šablony.

2.1. Struktura článku

Zpravodajový článek je psaný v českém, slovenském nebo anglickém jazyce. Sestává z názvu, abstraktu, klíčových slov, textu článku členěného do podkapitol a seznamu literatury. Název, abstrakt a klíčová slova se uvádí česky nebo slovensky a zároveň anglicky. Autor uvádí jméno, příjmení a e-mailovou adresu.

2.2. Šablona pro LaTeX

LaTeXová dokumentová třída `csbulletin` od Karla Horáka a Zdeňka Wagnera slouží pro sazbu zpravodajových článků a s výjimkou redakčních úprav dává

DOI: 10.5300/2018-1-4/1

autorovi věrný náhled výsledné podoby článku. Základní struktura českého nebo slovenského článku je následující:

```
1  \documentclass{csbulletin}[2019/01/12]
2  \selectlanguage{czech}
3  \usepackage[utf8]{inputenc}
4  \begin{document}
5
6  \title{Název článku}
7  \EnglishTitle{Anglický název článku}
8  \author{Jméno autora}
9  \podpis{Jméno autora, e-mailová adresa autora}
10 \maketitle
11
12 \begin{abstract}
13 Abstrakt článku.
14 \end{abstract}
15 \klicovaslova: první klíčové slovo, druhé klíčové slovo, ...
16
17 \section{Název podkapitoly}
18 Text článku.
19
20 \begin{summary}
21 Anglický abstrakt článku.
22 \keywords: první anglické klíčové slovo, druhé anglické ...
23 \end{summary}
24 \end{document}
```

V angličtině nahradíme řádek 2 za `\selectlanguage{english}`, řádky 12–15 za:

```
25 \begin{abstract}
26 Anglický abstrakt článku.
27 \keywords: první anglické klíčové slovo, druhé anglické ...
28 \end{abstract}
```

a řádky 20–23 za:

```
29 \section*{Český název článku}
30 \begin{otherlanguage}{czech}
31 Český abstrakt článku.
32 \end{otherlanguage}
33 \klicovaslova: první české klíčové slovo, druhé české ...
```

Ve slovenštině nahradíme příkaz `\klicovaslova` za příkaz `\klucoveslova`. Článek vysázíme příkazem `pdflatex`. Šablona je dostupná v archivu CTAN, v TEXových distribucích a na službě Overleaf (vizte `tiny.cc/overleaf-csbulletin`).

2.3. Šablona pro ConTEXt

Ačkoliv je Lamportův LaTeX stále dominantní TeXový formát, komunita kolem TeXového stroje LuaTeX a formátu ConTEXt Mark IV nabírá v posledních letech na síle a soustavně roste množství článků, které jsou připravené ve formátu ConTEXt. Přepis článků z ConTEXtu do LaTeXu představuje pro redakci časovou zátěž a vznikl proto experimentální modul pro ConTEXt,[1] který umožňuje vysadit tělo článku zapsané ve formátu ConTEXt při dodržení formátu Zpravodaje. Základní struktura českého nebo slovenského článku vypadá následovně:

```
34 \enableregime[utf]
35 \usemodule[t][csbulletin]
36 \language[czech]
37 \starttext
38 \startsection[title=Název podkapitoly]
39 Text článku.
40 \stopsection
41 \stoptext
```

V angličtině nahradíme řádek 36 za `\language[english]`. Pro vysázení článku stáhneme do adresáře se článkem soubor `t-csbulletin.tex`[1] a soubor článku `clanek-context.tex` zpracujeme příkazem `context`.

Nadpis a abstrakty se v současné době sází LaTeXovou dokumentovou třídou a výsledný článek vznikne překrytím výstupů LaTeXu a ConTEXtu. V LaTeXovém článku na straně 2 přidáme za řádek 36 `\usepackage{pdfpages}` a nahradíme řádky 17–23 za:

```
42 \section{Název podkapitoly}
43 Text první strany článku.
44
45 \makeatletter
46 \long\def\overlay{%
47   \thispagestyle{empty}
48   % Uprav odsazení tak, aby anglický abstrakt nepřekrýval text.
49   \vspace*{12.345cm}
50   \begin{summary}
51   Anglický abstrakt článku.
52   \keywords: první anglické klíčové slovo, druhé anglické ...
53   \end{summary}
54   \csbul@podepis
55 }%
56 \makeatother
57
58 {\shorthandoff{-}%
```

[1] Vizte `https://github.com/witiko/csbulletin-context`.

```
\pdfximage{clanek-context.pdf}%
\edef\last{\the\pdflastximagepages}%
\edef\penultimate{\number\numexpr\last-1}%
\includepdf[pages=1-\penultimate]{clanek-context}
\includepdf[pages=\last,pagecommand=\overlay]{clanek-context}}
```

Článek vysadíme příkazem `pdflatex`. Příkladem užití ConTEXtového modulu jsou články Hanse Hagena v tomto čísle.

3. Zpravodajová čísla

Při přípravě čísla Zpravodaje se články spojí do jednoho samostatného dokumentu. Je vygenerován obsah, strojově čitelná metadata pro registrační agenturu DOI a proběhne předtisková kontrola.

3.1. Struktura čísla

Číslo Zpravodaje sestává z obálky, obsahu, textu jednotlivých článků, tiráže a anglického obsahu. Na obálce se nachází grafika, která souvisí s tématem čísla. Obsah je generován automaticky ze značkování článků.

3.2. Sazba

Pro sazbu čísla se používá stejná LaTeXová dokumentová třída jako pro přípravu článků. Toto autorovi dává příležitost překvapit čtenáře nečekanou obálkou. ☺

Obrázek 1: Obálka Zpravodaje č. 2017/1–2 při zadání volby `color` (vlevo), `twocolor` (uprostřed) a žádné (vpravo).

Základní struktura čísla, zde ilustrovaná na čísle 2017/3–4, je následující:

```
64 \documentclass[color]{csbulletin}[2019/01/12]
65 \usepackage[utf8]{inputenc}
66 \usepackage{url}
67
68 \renewcommand\cslogodir{.}
69
70 \rok{2017}
71 \cislo{3--4}
72 \doisufix{2017-3-4/\thepage}
73 \uzaverka 9.12.2017
74 \naklad{310}
75 \CSBULilustrace{Petra Rychlá}
76
77 \begin{document}
78 \Obalka[%
79   \includegraphics[clip, scale=.3]%
80     {Sojka-TeXInSchools/texicons-p1-obalka}%
81 ]
82
83 \tableofcontents\vfill
84 \NovyTextPodObsah
85
86 \StartPage[117]
87
88 \PDFclanek[Uvodnik]{sojka-uvodnik}
89 \PDFclanek[Sojka-TeXInSchools]{main-cstug}
90 \PDFclanek[Hagen-LuaMetapost]{about-metafun}
91 \PDFclanek[Wilson-Glisterings-Odstavce]{7}
92
93 \tiraz
94
95 \EnglishTOC
96 \end{document}
```

Volba `color` na řádku 64 zadává, že se číslo bude tiskout barevně. Při nahrazení volbou `twocolor` se na obálce použijí pouze dvě barvy. Při odstranění volby dostáváme černobílou variantu. Jednotlivé varianty jsou k nahlédnutí v obrázku 1. Autor může použít příkaz `\ifcsbul@color` pro zjištění, jestli se číslo sází barevně, nebo černobíle. Pro přechod z tiskařského barevného modelu CMYK do barevného modelu RGB slouží volba `web`, na kterou se může autor analogicky dotazovat příkazem `\ifcsbul@web`.

Příkaz `\PDFclanek[`⟨*adresář*⟩`][`⟨*soubor*⟩`]` na řádcích 88–91 slouží k vložení článku ⟨*adresář*⟩`/`⟨*soubor*⟩`.tex` do čísla. Příkaz nejprve zapíše do pomocného souboru čísla informaci o použití příkazu. Dále příkaz zapíše do souboru ⟨*adresář*⟩`/`⟨*soubor*⟩`.info` informaci o číslu počáteční stránky v článku, identifikátor DOI článku na základě čísla počáteční stránky (vizte řádek 72), a nastavení příkazů `\ifcsbul@color` a `\ifcsbul@web` podle voleb zadaných v čísle. LaTeXová i ConTeXtová šablona pro články soubor ⟨*adresář*⟩`/`⟨*soubor*⟩`.info` načítá, čímž je zajištěn tok informací od celého čísla k jednotlivým článkům. Nakonec příkaz `\PDFclanek` vloží obsah souboru ⟨*adresář*⟩`/`⟨*soubor*⟩`.toc` do obsahu čísla, čímž je zajištěn tok informací od jednotlivých článků k celému číslu, a, pokud se jedná o druhý překlad, vloží do čísla PDF dokument ⟨*adresář*⟩`/`⟨*soubor*⟩`.pdf`.

Postup při sazbě čísla sestává z následujících kroků:

1. Poprvé přeložíme číslo, čímž získáme obálku a tiráž, ale obsah čísla je zatím prázdný a články nejsou vysázené. V souborech ⟨*adresář*⟩`/`⟨*soubor*⟩`.info` ale nyní máme úplnou informaci o barevnosti.
2. Poprvé přeložíme všechny články. K tomuto slouží program, který z pomocného souboru čísla načte seznam užití příkazu `\PDFclanek[`⟨*adresář*⟩`][`⟨*soubor*⟩`]` a následně v adresářích s články spustí příkaz `make -B`, pokud existuje soubor `Makefile`, nebo `latexmk -pdf` ⟨*soubor*⟩, pokud neexistuje.
3. Podruhé přeložíme číslo, které už kromě obálky a tiráže obsahuje i obsah a články, ale zatím s nesprávnými identifikátory DOI a čísly stránek.
4. Zopakováním kroků 2 a 3 získáme výsledný PDF dokument čísla.

3.3. Předtisková kontrola

Pro ušetření nákladů za tisk je vhodné zajistit, že ve článcích bude použitý barevný inkoust pouze pro ilustrace, u kterých je barva významná, a nikoliv k namíchání černé, nebo pro bezúčelné efekty. Za tímto účelem lze použít virtuální výstupní zařízení `inkcov`[2] programu Ghostscript, které pro každou stránku dokumentu vypíše procento stránky pokryté azurovým, purpurovým, žlutým a černým tiskařským inkoustem. Objekty v barevném modelu RGB jsou zařízením převedeny do tiskařského barevného modelu CMYK, přičemž odstíny šedé v RGB nevyužívají po převodu do CMYK jiný než černý inkoust. Následuje ukázkový výstup programu Ghostscript pro tři varianty obálky z obrázku 1:

```
$ gs -o - -sDEVICE=inkcov obalka-color.pdf
 0.00178  0.03065  0.02888  0.18653 CMYK OK
$ gs -o - -sDEVICE=inkcov obalka-twocolor.pdf
 0.00000  0.02888  0.02888  0.18831 CMYK OK
$ gs -o - -sDEVICE=inkcov obalka-gray.pdf
 0.00000  0.00000  0.00000  0.21718 CMYK OK
```

[2] Vizte `https://ghostscript.com/doc/Devices.htm#Ink_coverage_output`.

3.4. Export citací

Zpravodaj CSTUG je registrovaná organizace DOI s prefixem `10.5300`. To nás opravňuje přidělovat jednotlivým článkům identifikátory DOI, které jsou celosvětově unikátní, trvalé a usnadňují citování článků. Pro každé publikované číslo Zpravodaje vyžaduje systém DOI vytvoření strojově čitelných metadat, která popisují číslo, obsažené články a citované zdroje.

LaTeXový balíček BibLaTeX a program `biber` [1] jsou duchovní nástupci programu BibTeX [2], který rozšiřují mimo jiné o formálně zadefinovaný datový model bibliografických databází a o možnost převodu bibliografických databází ve formátu BIB do jazyka XML. Toho využíváme při generování strojově čitelných metadat pro registrační autoritu DOI Crossref, která přebírá metadata od registrovaných organizací DOI rovněž v jazyce XML.[3]

V prvním kroku upravíme články s citacemi tak, aby využívaly balíček BibLaTeX. Tato konverze slouží pouze pro generování strojově čitelných citací poté, co byl již vygenerován PDF dokument článku. Přesto však doporučujeme, aby autoři při přípravě využili právě balíček BibLaTeX. Významnou výhodou datového modelu bibliografických databází balíčku BibLaTeX je skutečnost, že je zpětně kompatibilní s de facto datovým modelem programu BibTeX. Pokud tedy máme následující soubor `clanek.tex`, který obsahuje článek s citacemi zpracovanými pomocí programu BibTeX:

```
\documentclass{csbulletin}[2019/01/12]
\bibliographystyle{unsrt}
\begin{document}
\bibliography{databaze-literatury}
\end{document}
```

potom stačí, když ho upravíme do následující podoby:

```
\documentclass{csbulletin}[2019/01/12]
\bibliographystyle{unsrt}
\usepackage{biblatex}
\addbibresource{databaze-literatury.bib}
\begin{document}
\printbibliography
\end{document}
```

U článků, které formátují seznam bibliografických citací příkazem `\bibitem` nebo příkazy balíčku `bib.sty` [3], je třeba nejprve ručně vytvořit bibliografickou databázi ve formátu BIB. Upravený článek přeložíme, díky čemuž se nám nyní v pomocném souboru článku nacházejí informace o odkazovaných citacích a v bibliografických databázích se nacházejí jednotlivé citace.

V druhém kroku vytvoříme soubor ⟨*prefix*⟩`_result.xml` se strojově čitelnými metadaty v jazyce XML z pomocného souboru článku a z bibliografických databází

[3] Vizte `https://support.crossref.org/hc/en-us/articles/214169586`.

tak, že v adresáři článku spustíme program `extract-citations`[4] příkazem `make -f extract-citations.mk`. Jednotlivé operace programu `extract-citations` ilustruje strom závislostí v obrázku 2, který si rozebereme od listů až po kořen představovaný souborem ⟨*prefix*⟩`_result.xml`.

Bibliografická databáze ⟨*prefix*⟩`_prefilter_bibertool.bltxml` v jazyce BibLaTeXML vznikne spojením bibliografických databází v adresáři článku do souboru ⟨*prefix*⟩`_prefilter.bib` a spuštěním příkazu `biber --input-format=bibtex --tool --output-format=biblatexml` ⟨*prefix*⟩`_prefilter.bib`. Seznam odkazů ⟨*prefix*⟩`_bibcites.xml` vznikne načtením příkazů `\bibcite` a `\abx@aux@cite` z pomocného souboru článku. Spojením bibliografické databáze ⟨*prefix*⟩`_prefilter_bibertool.bltxml` se seznamem odkazů ⟨*prefix*⟩`_bibcites.xml` vznikne soubor ⟨*prefix*⟩`_prefilter.xml`.

Bibliografická databáze ⟨*prefix*⟩`_bibertool.bltxml` v jazyce BibLaTeXML, která obsahuje pouze odkazované citace z bibliografické databáze ⟨*prefix*⟩`_prefilter_bibertool.bltxml`, vznikne XSL transformací ⟨*prefix*⟩`_prefilter.xsl` souboru ⟨*prefix*⟩`_prefilter.xml` na bibliografickou databázi ⟨*prefix*⟩`_prefilter_result.bltxml` a rozřešením křížových odkazů příkazem `biber --input-format=biblatexml --output-resolve --tool --isbn-normalise` ⟨*prefix*⟩`_prefilter_result.bltxml`.

Bibliografická databáze ⟨*prefix*⟩`_result.xml` v interním formátu CSTUGu, která obsahuje citace z bibliografické databáze ⟨*prefix*⟩`_bibertool.bltxml` seřazené podle pořadí odkazů ve článku, vznikne spojením bibliografické databáze ⟨*prefix*⟩`_bibertool.bltxml` se seznamem odkazů ⟨*prefix*⟩`_bibcites.xml` do souboru ⟨*prefix*⟩`.xml` a XSL transformací ⟨*prefix*⟩`.xsl` souboru ⟨*prefix*⟩`.xml`. Interní XSL transformací bibliografické databáze vzniknou metadata pro registrační autoritu DOI Crossref.

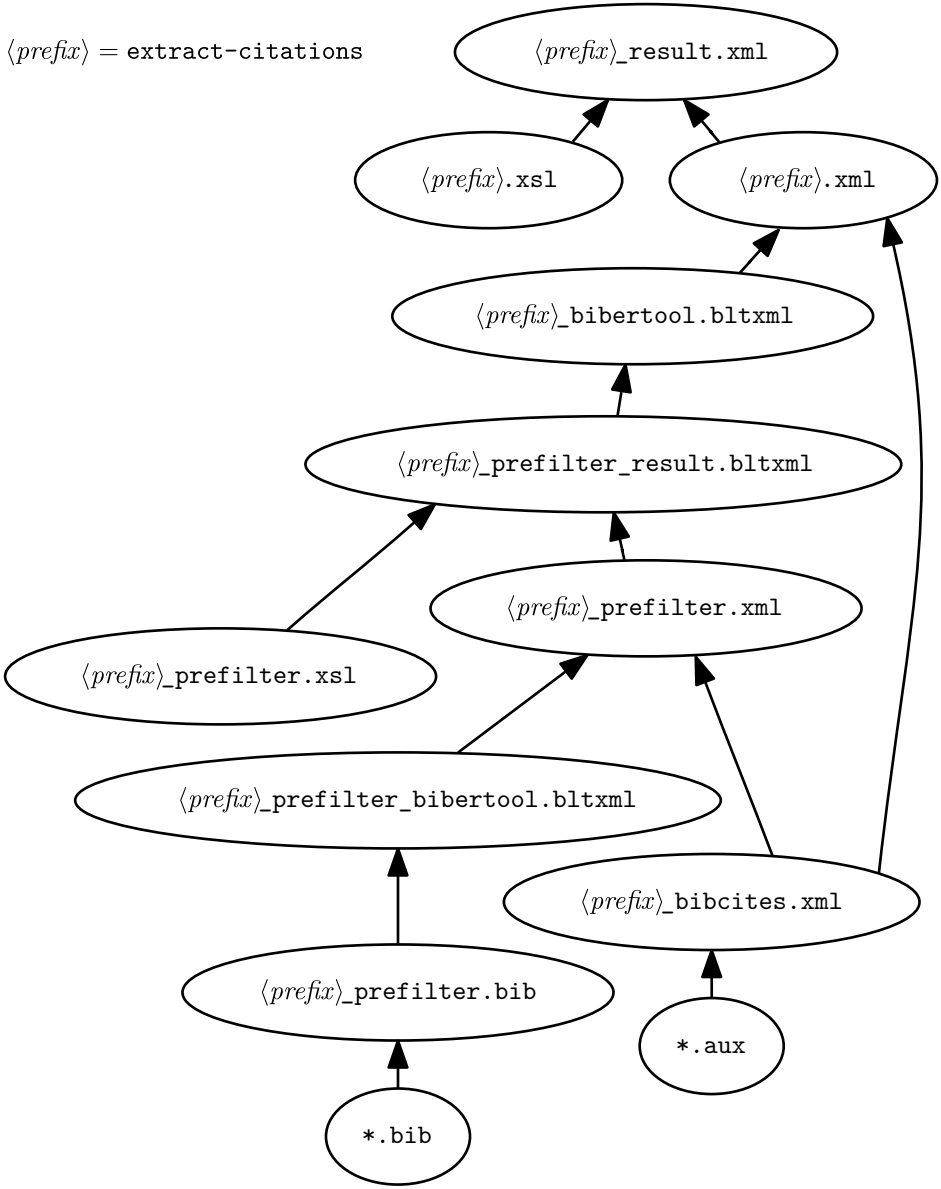
4. Závěr

Zpravodaj CSTUG je recenzovaný vědecký a odborný časopis s původními pracemi, který vychází od roku 1991. V článku jsem popsal strukturu a způsob přípravy zpravodajového článku a čísla, předtiskovou kontrolu a převod citací do strojově čitelné podoby vhodné pro zaslání registrační agentuře DOI.

Odkazy

[1] LEHMAN, Philipp; KIME, Philip; WEMHEUER, Moritz; BORUVKA, Audrey; WRIGHT, Joseph. *The biblatex Package* [online]. 2018 [cit. 2018-

[4] Vizte `https://github.com/witiko/extract-citations`.



Obrázek 2: Strom závislostí programu `extract-citations`.

11-30]. Dostupné z: http://mirrors.ctan.org/macros/latex/contrib/biblatex/doc/biblatex.pdf.
[2] PATASHNIK, Oren. *BibTEXing* [online]. 1988 [cit. 2018-11-30]. Dostupné z: http://mirrors.ctan.org/biblio/bibtex/base/btxdoc.pdf.
[3] HÁLA, Tomáš. Značkovací styl pro rychlou sazbu bibliografických citací. *Zpravodaj CSTUG*. 2008, roč. 18, č. 2008/3, s. 142–50. Dostupné z DOI: 10.5300/2017-3/142.

Summary: Preparing the CSTUG Bulletin

The article describes the structure, the typesetting and the preflight of the CSTUG Bulletin. We take a detailed look at the journey of a manuscript to the readers' mailboxes. The author has been the editor of the CSTUG Bulletin since 2016.

Keywords: LaTeX, ConTeXt, color spaces, Ghostscript, BibLaTeX, `biber`, XML

Vít Novotný, witiko@mail.muni.cz

Publikování z LaTeXu na web pomocí TeX4ht

Michal Hoftich

Publikování dokumentů vytvářených pomocí LaTeXu jako webových stránek je poměrně komplikovaný úkol. Představíme si konverzní systém *TeX4ht*, který tento proces umožňuje, a sestavovací program `make4ht`, který ho řídí.

Klíčová slova: TeX4ht, HTML

Úvod

Tento článek navazuje na předchozí příspěvek autora ve *Zpravodaji* [1], který byl zaměřený na problematiku tvorby elektronických knih, ovšem byly v něm také představeny postupy využívané konverzním systémem *TeX4ht* pro konverzi z TeXu do výstupních formátů a nastíněna problematika samotné konverze z TeXu do formátů založených na značkovacím jazyce XML. Nyní se zaměříme na některé nové vlastnosti sestavovacího programu pro *TeX4ht* jménem `make4ht` a pokročilejší možnosti konfigurace podoby výstupních souborů.

Připomeňme si však některé základní informace. *TeX4ht* je konverzní systém který je založený na vkládání instrukcí výstupního formátu přímo při překladu dokumentu TeXem. Výsledkem kompilace je DVI soubor, který je následně zpracován příkazem `tex4ht`. Ten vytvoří výsledné XML soubory a speciální soubory s příponami *.idv* a *.lg*. První z nich obsahuje DVI kód, ze kterého mají být vytvořeny obrázky (v základním nastavení například matematická prostředí). Druhý je řídící soubor pro poslední příkaz, `t4ht`. Ten je využíván pro konverzi obrázků z *.idv* souboru, volání externích příkazů, nebo vytvoření hlavního CSS souboru.

V předchozím odstavci jsme použili výraz *TeX4ht* ve dvou různých významech. Jednak jako název celého konverzního systému, jednak jako jednu z částí tohoto systému, která zpracovává DVI soubor. Existuje ještě další součást systému s tímto názvem, balíček TeXových maker `tex4ht.sty`, který zajišťuje vkládání XML značek do výstupního dokumentu.

Sestavovací program `make4ht`

Jak do tohoto schématu zapadá `make4ht`? Protože se celý konverzní proces skládá z několika na sebe navazujících kroků, využívají se pro kompilaci systémem *TeX4ht* skripty, které tento proces ulehčují. Existuje jich celá řada, liší se v podporovaném výstupním formátu a použitém TeXovém enginu a formátu. Nejznámější z nich

je *htlatex*, který využívá engine pdfTEX s formátem LATEX a produkuje výstup ve formátu HTML. Skripty zajišťují načtení balíčku s makry `tex4ht.sty` bez nutnosti jeho použití v konvertovaném dokumentu.

Tyto skripty jsou ovšem neflexibilní, každé jejich spuštění provede trojnásobnou kompilaci dokumentu TEXem. To zajistí správnou strukturu hypertextových odkazů a tabulek, ty totiž vyžadují několikanásobnou kompilaci pro svojí správnou funkci. V následujících krocích je dokument zpracován `tex4ht` a `t4ht`. Pokud dokument obsahuje například seznam literatury nebo zkratek, které jsou vytvářeny externími programy, je třeba zavolat nejdříve `htlatex`, poté požadovaný program a poté opět `htlatex`. V případě větších dokumentů může být čas potřebný pro kompilaci tímto způsobem poměrně dlouhý.

`make4ht` umožňuje tvorbu sestavovacích skriptů v jazyce *Lua*, ve kterých lze volat externí příkazy podle potřeby. S pomocí takzvaných *módů* lze ovlivnit pořadí kompilace pomocí přepínačů přímo z příkazové řádky. Například základní skript používaný `make4ht` podporuje *draft* mód, který spustí pouze jednu kompilaci dokumentu, místo obvyklých tří. Toho se dá využít pro zrychlení kompilace v případě, kdy se v dokumentu od poslední kompilace nezměnily křížové odkazy a pouze chceme získat rychlý náhled na jeho současnou podobu.

Původně byl `make4ht` součástí systému *TeX4ebook*, kde zajišťoval podporu pro sestavovací skripty, postupně se z něj však oddělil a v současné době plně nahradil staré skripty používané pro konverzi systémem *TeX4ht*. Kromě podpory pro sestavovací skripty také umožňuje snadný výběr TEXového engine použitého pro kompilaci, volbu výstupního formátu, podporu pro kódování Unicode ve výstupních souborech a další vlastnosti.

V současné době je podporován pouze LATEX, podpora pro PlainTEX je komplikovanější a ConTEXt podporován není. V následujícím textu se zaměříme pouze na LATEX.

Volby `make4ht` pro příkazovou řádku

`make4ht` podporuje řadu přepínačů a voleb, které ovlivňují průběh kompilace a zpracování výstupních souborů. `make4ht` může být spuštěn tímto způsobem:

```
make4ht [přepínače pro make4ht] soubor.tex "volby pro tex4ht.sty" \
"přepínače pro tex4ht" "přepínače pro t4ht" "přepínače pro TeX"
```

Tento komplikovaný způsob vychází z fungování `htlatex` a ostatních starých skriptů, které řešily nutnost předávání voleb pro všechny komponenty zapojené v kompilaci. Ve většině případů naštěstí není třeba této možnosti využívat, většinu vlastností, které poskytují `tex4ht` a `t4ht` lze vyžádat pomocí přepínačů pro `make4ht`.

Všechny přepínače, které `make4ht` podporuje, mají krátkou a dlouhou verzi. Krátké přepínače lze navíc spojovat dohromady. Následující dva příkazy jsou totožné:

```
make4ht --lua --utf8 --mode draft clanek.tex
make4ht -lum draft clanek.tex
```

Tento příkaz využije LuaLaTeX pro kompilaci, která proběhne pouze jednou díky využití *draft* módu, a výsledný dokument bude v textovém kódování UTF-8. Volby pro přepínače jsou od nich odděleny mezerou.

Kromě přepínačů `--lua`, `--utf8` a `--mode` existuje ještě řada dalších užitečných přepínačů:

`--config (-c)` konfigurační soubor pro *TeX4ht*. Umožňuje uživatelsky změnit značky vkládané do výstupních souborů.

`--build-file (-e)` sestavovací soubor.

`--output-dir (-d)` adresář, do kterého budou zkopírovány výstupní soubory.

`--shell-escape (-s)` použije volbu `--shell-escape` pro LaTeX, bude tedy možné spouštět z něj externí příkazy.

`--xetex (-x)` dokument bude kompilován pomocí XeLaTeXu.

`--format (-f)` volba výstupního formátu a rozšíření.

Existují ještě další přepínače, ale tyto jsou nejužitečnější pro běžné využití.

Výstupní formáty a rozšíření

TeX4ht podporuje širokou škálu formátů založených na XML, od XHTML přes ODT až po DocBook a TEI. Zájem uživatelů je však zaměřen prakticky pouze na varianty HTML a OpenDocument Format, který je podporován textovými procesory jako je LibreOffice nebo MS Word.

Přepínač `--format` tedy podporuje pouze následující formáty: html5, xhtml a odt (je třeba využít malá písmena). Přednastaveným formátem je HTML5. Ostatní formáty, jako je DocBook, lze vybrat pomocí volby pro `tex4ht.sty`:

```
make4ht filename.tex "docbook"
```

V přepínači `--format` lze také načíst rozšíření. Rozšíření umožňují ovlivnit kompilaci bez nutnosti použít sestavovací skript. Seznam využitých rozšíření lze zapsat za název formátu, zapínají se pomocí znaku plus, pomocí znaku mínus lze rozšíření zakázat[1]. Následující příkaz využije příkaz `HTML Tidy` pro opravení některých běžných chyb ve vygenerovaném HTML souboru:

```
make4ht -f html5+tidy simple-example.tex
```

Dostupná jsou následující rozšíření:

`latexmk_build` využije příkaz `latexmk` pro sestavení dokumentu. Ten se postará o volání externích příkazů, například pro tvorbu seznamu literatury.

`tidy` vyčistí HTML soubor pomocí příkazu `tidy`.

`dvisvgm_hashes` efektivní generování obrázků pomocí příkazu `dvisvgm`. Dokáže využít více procesorových jader a vytváří pouze obrázky, které byly změněny

[1] Rozšíření lze zapnout i ze sestavovacího skriptu, takové rozšíření je poté možné vypnout z příkazové řádky.

nebo vytvořeny od poslední kompilace. Tím dochází ke znatelnému zrychlení kompilace.

common_filters a **common_domfilters** vyčistění dokumentu pomocí filtrů. Filtrům se budeme věnovat dále v textu.

mathjaxnode konverze matematického kódu ve formátu MathML do speciálního HTML za využití projektu *MathJax Node Page*[2]. Díky tomu lze zobrazit kvalitně matematiku i ve webových prohlížečích bez podpory MathML, bez nutnosti využít JavaScript.

staticsite vytvoří dokument použitelný pro generátory statických stránek, jako je například *Jekyll*[3]. Ty jsou využitelné například pro tvorbu blogu.

Rozšíření je možné dále konfigurovat. Tím se dostáváme ke konfiguračnímu souboru a sestavovacím skriptům.

Konfigurační soubor pro make4ht

make4ht podporuje sestavovací skripty v jazyce Lua. Jejich pomocí je možné volat externí příkazy v průběhu kompilace, předávat jim parametry, používat filtry na výstupní soubory, ovlivňovat konverzi obrázků nebo konfigurovat rozšíření.

Konfigurační soubor .make4ht je speciální sestavovací skript, který je načítaný automaticky a měl by obsahovat pouze obecné konfigurace. Oproti tomu normální sestavovací soubory mohou obsahovat konfigurace platné pro daný dokument. Konfigurační soubor může být umístěný v adresáři s dokumentem, nebo v jeho nadřazených adresářích. To je užitečné například pokud bychom tvořili blog, kde každý dokument je umístěný ve vlastním adresáři. V nadřazeném adresáři může být umístěný konfigurační soubor, který zajistí správné zpracování. Zde je drobný příklad:

```
filter_settings "staticsite" {
  site_root = "output"
}

Make:enable_extension("common_domfilters")
if mode=="publish" then
  Make:enable_extension("staticsite")
  Make:htlatex {}
end
```

Tento konfigurační soubor nastavuje volbu site_root pro rozšíření *staticsite* pomocí příkazu filter_settings. Pomocí tohoto příkazu lze nastavit volby pro filtry, ale také rozšíření. Název filtru nebo rozšíření je oddělený od příkazu mezerou, poté následuje další mezerou oddělené pole, ve kterém lze nastavit volby

[2] https://github.com/pkra/mathjax-node-page/

[3] https://jekyllrb.com/

daného filtru. Oproti běžným konvencím příkazů v jazyce Lua zde nevyužíváme uvozovky jako u běžných funkcí!

Dalším použitým příkazem je `Make:enable_extension`, který povolí rozšíření. V tomto případě je rozšíření `common_domfilters` použito pro každou kompilaci, ovšem rozšíření `staticsite` pouze v módu *publish*, čehož je dosaženo podmínkou `mode=="publish"`.

Příkaz `Make:htlatex {}` vynutí jednu kompilaci LaTeXem.

Nyní lze spustit `make4ht` v módu *publish*:

```
make4ht -um publish simple-example.tex
```

Bude vytvořen adresář *publish*, pokud již neexistuje, do kterého budou zkopírovány HTML a CSS soubory ve formátu *datum publikace-originální název*. Statické generátory většinou očekávají názvy souborů v tomto formátu, čímž se zajistí chronologické řazení stránek.

Výsledný HTML soubor může mít následující podobu:

```
---
time: 1544811015
date: '2018-12-14 18:10:47'
title: 'sample'
styles:
- '2018-12-14-simple-example.css'
meta:
- charset: 'utf-8'
---
<p>Sample document</p>
```

Hlavička dokumentu uzavřená mezi dvojicí `---` obsahuje proměnné ve formátu YAML extrahované z HTML souboru, ze kterého zůstal pouze základní text. Statický generátor poté může vytvořit stránku na základě šablony a proměnných v hlavičce.

Toto byl pouze základní příklad, filtry a rozšíření mají mnohem širší možnosti nastavení, všechny volby jsou popsány v dokumentaci `make4ht` [2].

Sestavovací skripty

V sestavovacích skriptech můžeme využívat stejné postupy jako v konfiguračním souboru, ale více zaměřené na konkrétní kompilovaný dokument. Příklad sestavovacího skriptu byl představen v předešlém článku [1, s. 95], ukážeme si zde tedy pouze novou vlastnost, kterou jsou DOM filtry. Ty využívají možností knihovny LuaXML [3], která umožňuje zpracovávat XML soubory pomocí rozhraní *Document Object Model (*DOM*)*. Díky tomu lze snadno procházet elementy, upravovat je, vytvářet nebo odstraňovat.

Využití DOM filtrů si ukážeme na příkladu určeném pro LuaLaTeX:

```
\documentclass{article}
\begin{document}
Test {\itshape háčků}
\end{document}
```

Kvůli známé chybě při zpracování DVI souboru příkazem `tex4ht` bude vytvořen HTML soubor, kde každý znak s diakritikou bude umístěn v samostatném elementu:

```
<!--l. 4--><p class="noindent" >Test <span
class="rm-lmri-10">h</span><span
class="rm-lmri-10">á</span><span
class="rm-lmri-10">čk</span><span
class="rm-lmri-10">ů</span> </p>
```

Následující sestavovací soubor toto napraví pomocí vestavěného DOM filtru *joincharacters*. Navíc změníme atribut *class* všech elementů `<p>` na hodnotu *mypar*, abychom si ukázali práci s DOM rozhraním:

```
local domfilter = require("make4ht-domfilter")

local function domsample(dom)
  -- následující příkaz projde
  -- všechny elementy <p>
  for _, par in ipairs(dom:query_selector("p")) do
    -- nastavit atribut "class"
    par:set_attribute("class", "mypar")
  end
  return dom
end

local process = domfilter({"joincharacters", domsample})
Make:match("html$", process)
```

Skript využívá standardní funkci jazyka Lua `require` k načtení knihovny `make4ht-domfilter`. To vytvoří funkci `domfilter`, která má jako parametr seznam DOM filtrů, které se mají vykonat. Každé volání funkce `domfilter` vytvoří další funkci, která může být využita jako parametr pro funkci `Make:match`. Parametry v poli mohou být buď název existujícího DOM filtru, nebo funkce definovaná v sestavovacím souboru. Funkce `process` bude spuštěna na každém souboru jehož název končí na `html`.

Výsledný HTML soubor nyní neobsahuje nadbytečné elementy `<span>` a element `<p>` má hodnotu atribut *class* nastavenu na *mypar*:

```
<!-- l. 3 --><p class='mypar'>
Test <span class='rm-lmri-10'>háčků</span>
</p>
```

Konfigurace TeX4ht

Od `make4ht` nyní přejděme ke konfiguraci *TeX4ht* jako takového. Značky výstupního formátu vkládané do dokumentu jsou plně konfigurovatelné pomocí několika mechanismů. Nejjednodušší je využití voleb balíčku `tex4ht.sty`, větší možnosti nastavení nabízí konfigurační soubor a nejpokročilejší pak *4ht* soubory.

Při kompilaci TEXového souboru pomocí `make4ht` nebo jiného kompilačního skriptu se ještě před načtením samotného dokumentu načte balíček `tex4ht.sty`. Volby balíčku jsou získány z argumentů kompilačního skriptu. Díky tomu není třeba vkládat balíček `tex4ht.sty` v samotném dokumentu.

Mechanismus načítání souborů je upraven tak, aby se každý načítaný soubor registroval do *TeX4ht*. Pro některé balíčky *TeX4ht* také obsahuje kód, který zabraňuje jejich načítání, nebo okamžitě předefinovává některá makra. To je nezbytné pro balíčky, které jsou s *TeX4ht* nekompatibilní, jako je například *Fontspec*.

Po vykonání *preamble* dokumentu se načítají konfigurační soubory pro balíčky detekované při jejím zpracování. Tyto soubory mají název `jméno souboru bez přípony + .4ht`. Jejich hlavní funkcí je vkládat konfigurovatelná makra, takzvané háčky, do příkazů poskytovaných balíčkem. Obecně je lepší makra nepředefinovávat, ale pouze záplatovat pomocí příkazů, které pro tento účel *TeX4ht* poskytuje.

Po zpracování konfiguračních souborů pro balíčky se načítají konfigurační soubory výstupního formátu. Ty definují obsah konfiguračních maker, háčků. Především se do nich vkládají značky výstupního formátu, ale mohou obsahovat libovolné příkazy. Je možná také podmíněná konfigurace na základě voleb balíčku `tex4ht.sty`. Každý konfigurační soubor výstupního formátu může testovat libovolnou volbu, z čehož vyplývá, že neexistuje jejich konečný seznam a pro každý formát existují jiné volby.

Volby balíčku `tex4ht.sty`

Protože se nedoporučuje vkládat balíček `tex4ht.sty` přímo do dokumentu, je možné předat mu volby jiným způsobem. Nejjednodušším je pomocí argumentu pro kompilační skript. Vždy je to argument následující po názvu dokumentu:

```
make4ht filename.tex "mathml,mathjax"
```

Volby použité v tomto případě jsou *mathml* a *mathjax*. Další možností předání voleb je předání pomocí příkazu `\Preamble` v soukromém konfiguračním souboru, který si ukážeme v další sekci.

Jak již bylo řečeno, neexistuje konečný seznam voleb, každý výstupní formát může podporovat jejich libovolné množství.

Můžeme si ovšem ukázat některé volby, které se týkají matematických výstupů v HTML. Normální konfigurace pro matematická prostředí produkuje směs for-

mátovaného textu a obrázků pro složitější výstupy, které nejdou snadno vytvořit pomocí HTML elementů. Někdy tato vlastnost nevypadá dobře. Jako alternativu lze použít obrázky pro veškerý matematický obsah. Toho lze dosáhnout pomocí voleb *pic-m* pro inline matematiku a *pic-název prostředí* pro matematická prostředí, například *pic-align*.

Normálně jsou obrázky vytvářeny ve formátu PNG. Vyšší kvality lze dosáhnout s využitím vektorového formátu SVG. Ten může být vynucen pomocí volby *svg*.

Dokumentace *TeX4ht* je poměrně spartánská. Obsáhlejší informace o konfiguraci příkazů lze nalézt v log souboru vytvořeném při kompilaci dokumentu s pomocí volby *info*.

Volby uvedené v prvním příkladu, *mathml* a *mathjax* poskytují obecně nejlepší výstup matematického obsahu. Značkovací jazyk MathML umožňuje zakódovat matematické informace, jeho podpora ve webových prohlížečích je však špatná. S využitím knihovny *MathJax*[4] je možné jej zobrazit ve všech moderních prohlížečích s podporou jazyka JavaScript.

Samotná volba *mathjax* bez *mathml* úplně vypne zpracování matematiky při kompilaci, veškerý matematický obsah zůstane v HTML dokumentu jako TEXová makra. *MathJax* poté dokument zpracuje a vykreslí matematiku správným způsobem. Nevýhodou tohoto způsobu je, že *MathJax* nepodporuje všechny balíčky a uživatelské příkazy.

Soukromý konfigurační soubor

Pomocí soukromého konfiguračního souboru je možné vkládat vlastní obsah do konfiguračních háčků. Tento soubor má zvláštní strukturu:

```
...Definice v~preamble...
\Preamble{volby pro tex4ht.sty}
... Normální konfigurace ...
\begin{document}
... Konfigurace pro hlavičku \HTML{} souboru
\EndPreamble
```

Příkazy `\Preamble`, `\begin{document}` a `\EndPreamble` musí být v konfiguračním souboru obsaženy. Cesta k souboru může být předána *make4ht* pomocí parametru `-c`:

```
make4ht -c myconfig.cfg filename.tex
```

Pokud konfigurační soubor není uložený v aktuálním pracovním adresáři, je nutno použít plnou cestu k souboru.

Existuje několik konfiguračních příkazů. Nejdůležitějším je `\Configure`, který nastavuje běžné konfigurace, dalšími jsou například `\ConfigureEnv` pro nastavení prostředí, nebo `\ConfigureList` pro seznamy.

[4] `https://www.mathjax.org/`

Příkaz `\HCode` se používá pro vkládání značek výstupního formátu. Součástí jeho argumentu může být příkaz `\Hnewline` pro vložení zalomení řádku. Instrukce `\Css` vloží kód pro kaskádové styly do CSS souboru.

Následující ukázka použije element `<em>` pro příkaz `\textit`:

```
\Configure{textit}{\HCode{<em>}\NoFonts}{\EndNoFonts\HCode{</em>}}
```

Příkaz `\Configure` podporuje variabilní počet argumentů, záleží na definici háčků, kolik argumentů je potřeba. Prvním argumentem je vždy název konfigurace, další argumenty poté vkládají kód do háčků. V typickém případě vyžaduje konfigurace dva háčky – jeden umístí kód před začátek příkazu, druhý za jeho konec. Tak je tomu v případě konfigurace pro `textit`. Název konfigurace se může shodovat s názvem konfigurovaného příkazu, jako v tomto případě, ale vždy záleží na konfiguračním *4ht* souboru pro daný balíček, jaký název použije.

Příkaz `\NoFonts` zakáže vkládání formátovacích elementů při zpracování DVI souboru. `tex4ht` vkládá automaticky tyto elementy při změně písma. Díky tomu je možné vytvořit základní formátování i pro nepodporované příkazy bez konfigurací, ovšem pro konfigurované příkazy je toto zbytečné.

Komplikovanou problematikou jsou odstavce. *TeX4ht* je někdy vkládá na nevhodná místa. Týká se to především konfigurací prostředí, která mohou obsahovat několik odstavců a celý svůj obsah umísťují do jednoho elementu. Může se stát, že počáteční značka odstavce je umístěna před začátkem tohoto elementu, správně by ovšem měla následovat až po něm. Pro tento případ existují příkazy `\IgnorePar`, který zabrání vložení značky pro následující odstavec, a `\EndP`, který vloží zavírací značku pro předešlý odstavec. Existuje více příkazů pro práci s odstavci, ale tyto jsou nejdůležitější.

V následujícím příkladu použijeme hypotetické prostředí `rightaligned`, které by mělo být umístěno v elementu `<article>`.

```
\ConfigureEnv{rightaligned}
{\HCode{<section class="right">}}
{\HCode{</section>}}{}{}
```

Příkaz `\ConfigureEnv` očekává pět parametrů, prvním je název konfigurovaného prostředí, druhým je kód vložený na začátku prostředí a třetím kód vložený na jeho konci. Zbylé dva argumenty se používají pouze pokud je konfigurované prostředí založeno na seznamu, ve většině případů mohou zůstat prázdné. HTML kód vytvořený touto konfigurací může vypadat následovně:

```
<p class="indent" >   <section class="right">
...
</p><p class="indent></section>
```

Tento kód je nevalidní, protože ukončovací značka pro element `<p>` je umístěna na špatné úrovni zanoření. Nevalidní kód může způsobit přerušení běhu DOM filtrů a jiných nástrojů, které zpracovávají výsledné XML soubory. Této situaci je třeba předcházet.

Správná konfigurace je poněkud komplikovanější:

```
\ConfigureEnv{rightaligned}
{\ifvmode\IgnorePar\fi\EndP\HCode{<section class="right">}\par}
{\ifvmode\IgnorePar\fi\EndP\HCode{</section>}}{}{}
```

V tomto případě řídíme vkládání značek pro odstavce sami a výsledek je v pořádku:

```
<section class="right">
<!--l. 9--><p class="indent" >
...
</p></section>
```

V konfiguracích můžeme také vyžádat konverzi části dokumentu na obrázek. Toho se dá docílit pomocí příkazů `\Picture*`, respektive `\Picture+`. Rozdíl mezi nimi je ten, že první zpracovává svůj obsah jako vertikální box. V každém případě se obsah mezi těmito příkazy a ukončovacím příkazem `\EndPicture` převede na obrázek.

Těchto příkazů lze využít například pro konverzi složitější matematiky nebo diagramů, ale čistě technicky podporují veškerý obsah, který dokáže zpracovat použitý konvertor z DVI do obrazových souborů, typicky *Dvipng* nebo *Dvisvgm*.

Následující příklad vytvoří obrázek pro text obsažený v prostředí *topicture*:

```
\documentclass{article}
\newenvironment{topicture}{\bfseries}{}
\begin{document}
\begin{topicture}
Obsah tohoto prostředí by měl být zobrazen jako obrázek
\end{topicture}
\end{document}
```

Konfigurace pro prostředí *topicture* využije příkazu `\Picture*`:

```
\ConfigureEnv{topicture}{\Picture*{}}{\EndPicture}{}{}
```

Závěr

Možnosti konfigurace *TeX4ht* jsou rozsáhlé, v předešlém textu jsme se dotkli pouze základů, které by však měly vystačit pro řešení nejběžnějších otázek, které uživatelé tohoto systému řeší. Vynechali jsme také ukázky, jakým způsobem lze přidat konfiguraci pro nový LaTeXový balíček. Tato témata by vydala na rozsáhlé samostatné články.

Díky sestavovacímu systému `make4ht` je použití celého systému snazší a efektivnější než tomu bylo v minulosti.

V současné době probíhá za finanční podpory poskytnuté *Československým sdružením uživatelů TeXu* tvorba nové dokumentace pro *TeX4ht*, kde budou

témata tohoto článku probrána do větší hloubky, spolu s konkrétními příklady užití.

Odkazy

[1] HOFTICH, Michal. Elektronické knihy a systém TeX4ebook. *Zpravodaj Československého sdružení uživatelů TEXu*. 2016, roč. 26, č. 1–4, s. 94–105. Dostupné také z: `http://bulletin.cstug.cz/doi.php/10.5300/2016-1-4/94`.

[2] HOFTICH, Michal. *The Make4ht package: A build system for tex4ht* [online]. 2018. Verze 0.2c [cit. 2018-12-14]. Dostupné z: `http://www.ctan.org/pkg/make4ht`.

[3] HOFTICH, Michal. *The Luaxml package: Lua library for reading and serialising XML files* [online]. 2018. 0.1g [cit. 2018-12-14]. Dostupné z: `http://www.ctan.org/pkg/luaxml`.

Summary: LaTeX to Web Publishing using TeX4ht

The article gives overview of the current state of development of TeX4ht, LaTeX to XML convertor. It introduces `make4ht`, a build system for TeX4ht as well as basic ways how to configure TeX4ht.

Keywords: TeX4ht, html

Michal Hoftich

Tabulky v dobře dokumentovaných statistických výpočtech

Marek Pomp

Článek popisuje metodu publikování dobře dokumentovaných statistických výpočtů s pomocí programu R. Speciálně se věnuje vytváření tabulek za pomocí balíčků `knitr` a `kableExtra`. Článek volně navazuje na [4].

Úvod

Při psaní textů obsahující statistické výpočty je vhodné použít metodu tzv. *reprodukovatelného výzkumu* [2], to znamená publikovat výsledky výzkumu včetně získaných dat a přesného popisu metod při zpracování dat. Vytvářet a uchovávat text publikace, tabulky a ilustrace včetně přesného popisu použitých statistických výpočtů můžeme kombinací LaTeXovského zdrojového textu a zdrojového kódu v některém statistickém software, například v programu R. Jedna z možností, jak začlenit zdrojový kód jazyka R do TeXovského zdrojového souboru, použití balíku `Sweave` [3], bylo popsáno i v tomto časopise [4]. Přibližně v roce 2012 se balíčkem `Sweave` inspiroval Yihui Xie a začal s vývojem `knitr`u, viz [6]. Na rozdíl od `Sweave`, který řeší jen export z R do LaTeXu, je `knitr` obecnější a zvládá spolupráci programu R[1] s mnoha dalšími formáty dokumentů. V dalším textu bude stručně popsáno použití balíku `knitr` zejména s ohledem na sazbu tabulek v LaTeXu.

Sazbu tabulek z programu R v LaTeXu řeší velmi sofistikovaný balíček `xtable`. S rozšířenými možnostmi `knitr`u je ale vhodné používat nástroje, které se nespecializují jen na tabulky ve formátu LaTeX, ale dokáží vytvořit tabulky použitelné také v jiných značkovacích jazycích. Tyto podmínky splňuje funkce kable z balíku `knitr` podpořená rozšířením z balíku `kableExtra`.

Poznámka pro čtenáře: Každá tabulka v textu je komentovaná a poblíž tabulky, obvykle na konci sekce, je uveden kód, jak příslušnou tabulku vytvořit. Naneštěstí není vždy možné, aby zobrazení tabulky, komentář a její kód byly na stejné straně.

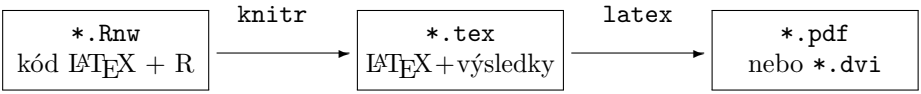
`Knitr` – principy

Použití balíku `knitr` je v základě stejné jako u balíčku `Sweave`. Do LaTeXovského dokumentu se vkládají tzv. *chunks*, bloky kódu v jazyce R. Smíšený dokument s

[1] Nejen R ale i jiných programovacích jazyků.

 doi: 10.5300/2018-1-4/22

příponou `Rnw` se kompiluje nejprve programem R. Během kompilace v programu R se vyhodnotí všechny bloky kódu jazyka R a výsledky se spolu s původním kódem LaTeXu ukládají to čistého LaTeXovského souboru. Tomuto postupu se říká *knitting* (v balíčku `Sweave` se tomuto procesu říkalo *weaving*). Z výsledného `*.tex` dokumentu lze vytvářet výstupy běžným LaTeXovským překladačem, viz obr. 1.



Obrázek 1: Schéma práce s `knitr`em.

Doplňkový proces nazvaný *purl* (pro Sweave se nazýval *tangle*) z původní směsi kódů pro program R a LaTeX vybírá jen bloky s kódem pro R a ty ukládá do zvláštního souboru.

Bloky kódu programu R jsou ve zdrojovém dokumentu `*.Rnw` od LaTeXovského kódu odděleny hlavičkou ve tvaru
`<<`*návěští bloku*`,` *volby a parametry bloku*`>>=`
a za kódem v programu R následuje uzavírací znak bloku, zavináč `@`. Hlavička bloku i uzavírací zavináč musí být uvedeny na samostatném řádku, bezprostředně na jeho začátku (bez mezer a tabulátorů).

Soubor `*.Rnw` budeme v programu R kompilovat funkcí `knit` z balíku `knitr`. na příkazové řádce v linuxu[2] by příkazy pro kompilaci mohly vypadat následovně:[3]

```
$ Rscript -e "knitr::knit('soubor.Rnw')"; pdflatex soubor.tex
```

Pro větší komfort uživatelů programu R je v balíčku `knitr` také funkce `knit2pdf`, která pouští postupně funkci `knit` a poté, aniž by musel uživatel opustit prostředí programu R, spustí rovnou překladač TeXu zvolený v parametru `compiler`. Na příkazové řádce programu R bychom mohli spouštět např.

```
> knitr::knit2pdf("soubor.Rnw", compiler = "xelatex")
```

Funkce `knit2pdf` kompiluje překladačem TeXu tolikrát, aby byly správně vyhodnoceny křížové odkazy, a pokud je to potřeba, zavolá v pravý čas také BiBTeX a MakeIndex[4].

Většina uživatelů používá k vytváření dokumentů nějaké komplexní vývojové prostředí, např. Emacs, vim, WindEdt, TeXnicCenter a další.

[2] Příkazovou řádku operačního systému značíme promptem `$`, příkazovou řádku v programu R znakem `>`.

[3] Nebo v přehlednější formě

```
$ Rscript -e "require(knitr); knit('soubor.Rnw')"; pdflatex soubor.tex
```

[4] Funkce `knit2pdf` spouští LaTeX prostřednictvím systémové funkce `latexmk`.

V komplexních editačních systémech je obvykle volba pro použití `knitru` přednastavená. Samozřejmostí je podpora `knitru` pro LyX.

Protože kompilace celého dokumentu bývá zdlouhavá[5], je vhodným zvykem ladit výsledné výpočty po částech, spouštět výpočet jen pro vybrané bloky kódu a celý dokument sestavit až pro správně probíhající výpočty. Tuto možnost nabízí např. editační prostředí RStudio (viz [5]), pluginy pro editory Emacs nebo vim a další nástroje.

Balík `knitr` není navržen jen pro spolupráci s TEXem, ale také pro psaní dokumentů v jazyce Markdown. Proto jsou některé jeho vlastnosti a funkce specializované na jeden z těchto jazyků a nemusí fungovat obecně. Autor textu, který nepočítá jen s využitím `knitru` v LATEXovském dokumentu, by tedy měl mít na paměti, že je třeba zvážit, zda využít voleb a vlastností odkazující na LATEXovské speciality. Na tyto vlastnosti se pokusíme upozornit na příslušných místech.

Volby v hlavičce bloku

Volby v hlavičce bloku kódu určují chování programu R při vyhodnocování bloku a zobrazení obsahu bloku a jeho výsledků. Nejdůležitější volby pro bloky vytvářející tabulky jsou:

návěští bloku — je první[6] volbou v hlavičce bloku a je to volba nepovinná. Návěští, pokud je uvedeno, musí být pro každý blok jedinečné. Pro *návěští bloku* se doporučuje použít sekvenci alfanumerických znaků a pomlček, bez mezer. *Návěští bloku* slouží k odkazování na blok ve výpočtech, `knitr` umí vyhodnotit blok znovu na jiném místě dokumentu. *Návěští bloku* je také využito k pojmenování souborů s vygenerovanými obrázky. Dále je *návěští bloku* využito jako základ křížových odkazů na nadpisy tabulek. Křížový odkaz na tabulky bude mít v LATEXovském souboru návěští označené `tab:`*návěští bloku* (obrázky mají automaticky vytvářené návěští `fig:`*návěští bloku*).

`echo` — určuje, zda se ve výsledném dokumentu objeví zdrojový kód zapsaný v bloku. Při čtení publikace asi čtenáře naše výpočty do detailu nezajímají a je vhodné nastavit tuto volbu na `FALSE` (a nechat zvídavé, ať detaily výpočtů studují ve zdrojovém kódu publikace).

`results` — popisuje, jakou formou budou zobrazeny výsledky. Protože výsledky výpočtů v tom formátu, v jakém je produkuje program R, nejsou pro

[5] Ti, kteří měli možnost překládat TEXovské dokumenty na PC AT 286 rychlostí jedna strana za cca 15 s, mohou slovo „zdlouhavý“ relativizovat.

[6] `Knitr` umí v bloku volat i kód jiných jazyků, než programu R, takže úplně první volbou by mohl být název volaného jazyka např. `bash`, nebo `perl`.

nepoučeného čtenáře moc čitelné, formátujeme výsledky obvykle do vhodně popsaných tabulek. Pro tabulky exportované do formátu LaTeX používáme volbu `results='asis'` (je možné použít i dvojité uvozovky `"asis"`).

Celý blok tedy tedy může vypadat takto:

```
<<prvni_chunk, echo = FALSE, results = 'asis'>>=
... kód programu R
@
```

V dalším textu už bude vždy ukázán jen kód pro program R bez hlaviček bloku.

Volby pro hlavičku bloku lze nastavit globálně pro celý dokument. V některém z úvodních bloků může být uvedeno např.

```
knitr::opts_chunk$set(echo = FALSE, results = 'asis')
```

aby se v celém dokumentu neukazoval zdrojový kód, ale jen tabulky.

Tabulky obecně

Výstupy statistických výpočtů jsou nejčastěji prezentovány ve formě tabulek. Sazba tabulek je komplikovaná oblast typografie, ve které je velký prostor pro vedení sporů o to, co je správné. Přesto se asi většina sazečů shodne, že běžná tabulka z manuálů pro LaTeX mívá

- příliš mnoho čar a linek,
- nepřehledné záhlaví a
- malou výšku řádků.

Vylepšení vzhledu tabulek řeší v LaTeXu balíček `booktabs` [1]. V balíčku `booktabs` jsou zavedeny nové příkazy pro horizontální linky v tabulkách, je nastaven širší řádkový rejstřík tabulek a několik dalších drobností tak, aby tabulky vyhovovaly stylu University of Chicago Press.

Pro sazbu tabulek přesahujících jednu stranu je v LaTeXu často používaný balíček `longtable`. Balíček `booktabs` je kompatibilní s balíkem `longtable`.

Jednoduché tabulky, funkce `kable`

Funkce pro tvorbu tabulek `kable` z balíku `knitr` [6] je velmi jednoduchá funkce a autor se podle vlastních slov nesnažil nahradit zavedený a sofistikovaný balíček `xtable` pro sazbu LaTeXovských tabulek v prostředí R. Funkce `kable` tak poskytuje jen základní možnosti sazby tabulek z objektů třídy `data.frame` nebo `matrix`[7].

[7] Ve skutečnosti tato funkce dokáže vytvořit tabulky i z vektorů nebo objektů třídy `array` nebo `list`, ale tyto objekty je lépe předem upravit na obdélníková schémata typu `data.frame`.

Mějme jednoduchá data, několik řádků vybraných z datového souboru, který uvádí IQ dvojčat vychovávaných odděleně a jejich sociální status.

```
mydata <- carData::Burt[c(5,10,12,19,26),]
```

ze kterých vyrobíme tabulku funkcí `kable`.

```
kable(mydata, booktabs = TRUE,
  col.names = c("První", "Druhý", "Třetí"),
  caption = "Jednoduchá tabulka vytvořená funkcí \\texttt{kable}"
  )
```

Tabulka 1: Jednoduchá tabulka vytvořená funkcí `kable`

	První	Druhý	Třetí
5	116	115	high
10	93	82	medium
12	88	100	medium
19	97	87	low
26	107	106	low

Prvním parametrem funkce `kable` je datový soubor. Parametr `booktabs` udává, zda má být tabulka formátována ve stylu *Chicago*, jen s vodorovnými linkami omezujícími tabulku a s vyznačeným záhlavím. Do záhlaví tabulky vstupuje parametr `col.names`. Pokud `col.names` není uveden, budou v záhlaví názvy sloupců z datového souboru. V tabulce budou uvedeny názvy řádků v závislosti na parametru `row.names`, přednastaveno je `row.names = TRUE`, takže se v tabulce objeví názvy řádků z datového souboru i když tato volba nebude uvedena.

Tabulka bude umístěna do plovoucího prostředí `table`, pokud obsahuje parametr `caption`. Návěští pro odkazy je automaticky přidáno podle parametru *návěští bloku* ve formě `\label{tab:`*návěští bloku*`}`. Na tabulku pak kdekoliv v textu odkazujeme pomocí `\ref{tab:`*návěští bloku*`}`.

Malá poznámka ke zdvojování zpětných lomítek. Většina poučených čtenářů ví, že zpětné lomítko bývá interpretováno jako speciální znak (escape), který mění význam pro ostatní znaky. Pokud chceme v regulárním výrazu vložit znak „zpětné lomítko“, zapisujeme jej jako escape sekvenci dvojicí zpětných lomítek. Příkazy LaTeXu jsou proto v argumentech funkcí programu R uvozeny dvojicí zpětných lomítek. Ovšem, jestliže se regulární výraz vyhodnocuje vícekráte, je třeba znovu zdvojovat řetězec escape sekvencí, a proto např. v argumentu `big.mark` musí být malá LaTeXovská mezera zapsána pomocí čtyř zpětných lomítek a čárky, viz kód na str. 28.

Oddělení řádků mezerou, zarovnání sloupců

V tabulkách `kable`, které mají více než šest řádků, jsou, pro přehlednost, automaticky po pěti řádcích přidány vertikální mezery příkazem `\addlinespace`. Toto chování se dá změnit nastavením parametru `linesep`. Následující nastavení vkládá mezery už po třech řádcích.

Funkce `kable` má přednastaveno, že sloupce s čísly budou zarovnány vpravo a ostatní sloupce vlevo. Defaultní nastavení měníme v parametru `align` pomocí příznaků `c`, `r`, `l`. Tyto parametry můžeme zadat jako položky vektoru pro každý sloupec zvlášť, nebo zadáme jediný společný parametr pro stejné zarovnání všech sloupců. Ve druhém případě doopravdy využíváme schopnosti programu R recyklovat krátké vektory na potřebnou délku, takže např. pokud bychom chtěli zarovnat sudé sloupce tabulky jinak než liché, zadáme pouze dvě volby. Návěští řádků je zarovnáno vždy doleva.

V parametru `align` je možné přidávat také vertikální linky a další formátování sloupců podle zvyku LaTeXovských tabulek. S ohledem na využití tabulek také v dokumentech v jazyce Markdown je systémovější způsob namísto speciálních LaTeXovských sekvencí (např. `|` pro vertikální linku) použít pro formátování sloupců tabulky balík `kableExtra` a jeho funkce, např. funkci `column_spec` a její parametry `border_left`, `border_right`. Přesto je v následující tabulce poslední sloupec oddělen svislou linkou a formátován parametrem `p` na šířku 2 cm.

	IQbio	IQfoster	class
5	116	115	high
10	93	82	medium
12	88	100	medium
19	97	87	low
26	107	106	low

```
kable(mydata, booktabs = TRUE,
  linesep = c("", "", "\\addlinespace"),
  align = c("c","l","|p{2cm}")
  )
```

Čísla v tabulkách

Číselné výstupy programu R jsou nastaveny tak, aby maximální počet číslic ve výpisech byl sedm, bez ohledu na desetinnou čárku. Změnit globální nastavení *počtu číslic* na výstupech je možné funkcí programu R `options` v parametru `digits`, např. `options(digits = 6)`

Tabulky vytvářené funkcí `kable` respektují nastavení počtu výstupních číslic, ale navíc je možné určit počet desetinných míst, pro zaokrouhlení výstupů v tabulce. Volba počtu *desetinných* míst pro zaokrouhlení se jmenuje `digits` (stejně jako globální volba počtu *všech* číslic na výstupu, což může být trochu matoucí).

Pokud bychom chtěli každý sloupec tabulky zaokrouhlit jiným způsobem, je možné ve volbě digits použít vektor. V následující tabulce je první sloupec zaokrouhlený na dvě a druhý na jedno desetinné místo.

Jako oddělovač desetinných míst slouží podle české typografické tradice desetinná čárka. Nastavit desetinnou čárku pro všechny výstupy lze globálně volbou `options(OutDec = ",")` Lokální nastavení v tabulce můžeme měnit v parametru `format.args` jako volbu `decimal.mark`.

	IQbio	IQfoster
5	36 442,47	36 128,3
10	29 216,81	25 761,1
12	27 646,02	31 415,9
19	30 473,45	27 331,9
26	33 615,04	33 300,9

Pro čtenáře je nepříjemné, pokud by měl číst dlouhá čísla, která postrádají jakékoliv formátování. Dobrým zvykem je oddělit každé tři číslice v číslovkách mezerou (nebo jiným znakem). Oddělovač tisíců se nastaví ve volbě `format.args`, parametrem `big.mark`. Protože argument funkce `big.mark` prochází dvojí expanzí, musíme zpětná lomítka zmnožit na čtyři. Funkce `kable` je nastavena tak, aby speciální znaky v buňkách expandovala do formátu stravitelného LaTeXem. Např. zpětné lomítko transformuje na sekvenci `\textbackslash`, znak procenta na `\%` apod. Pokud ale vkládáme speciální LaTeXovské znaky přímo do buněk, musíme toto chování potlačit volbou `escape = FALSE`.

```
kable(mydata[c(1, 2)] * 100 * pi,
  booktabs = TRUE,
  digits = c(2, 1),
  escape = FALSE,
  format.args = list(decimal.mark = ",", big.mark = "\\\\,")
  )
```

Rozšíření funkce `kable`, balík `kableExtra`

Používání tabulek vytvářených funkcí `kable` se ujalo, (snad) hlavně proto, že `knitr` podporuje vytváření dokumentů v odlehčeném značkovacím jazyce R Markdown. Proto brzy vzniklo rozšíření možností pro formátování tabulek, balík `kableExtra`. Při použití balíku `kableExtra` by zdrojový text měl obsahovat řádek:

```
require(kableExtra)
```

Balík `kableExtra` zavádí několik nových funkcí, jejichž vstupem je tabulka `kable`, kterou na výstupu modifikují. Při práci s balíkem `kableExtra` se doporučuje využívat pro přesměrování výstupů softwarovou rouru, operátor `%>%`[8]. Roura

[8] Roura `%>%` není základní funkcí programu R, ale je balíkem `kableExtra` načtena z balíčku `magrittr`.

přesměruje výstup jedné funkce na vstup druhé. Např. v R (po načtení balíčku `kableExtra`) jsou tyto dva řádky

```
kable(mydata)
mydata %>% kable()
```

ekvivalentní způsoby k zobrazení tabulky pomocí funkce `kable`.

Modifikace celých řádků a sloupců tabulky

Následující příklad ukazuje použití funkcí pro modifikaci řádků nebo sloupců `row_spec` resp. `column_spec`. Prvním z argumentů těchto funkcí je číslo řádku (sloupce), který máme modifikovat. Záhlaví je označeno jako řádek 0, ovšem názvy řádků, pokud se zobrazují, jsou počítány jako první sloupec.

Funkce `row_spec` a `column_spec` mohou modifikovat typ písma, přičemž názvy parametrů jsou samovysvětlující (`bold`, `italic`, `monospace`, `underline`, `strikeout`[9]). Změna barvy písma a pozadí se provede nastavením parametrů `color` a `background`. Barvy lze zadávat ve formátu RGB nebo lze použít standardní názvy barev z LaTeXovského balíku `xcolor`. Balík `xcolor` je třeba mít načtený v hlavičce dokumentu, s příslušnou volbou (např. `x11names`).

Další parametry už mají funkce `row_spec` a `column_spec` rozdílné a zmíníme je pro každou funkci zvlášť.

Funkce `row_spec` může měnit velikost písma na vybraném řádku tabulky parametrem `font_size`. Tento parametr vyžaduje číslo, velikost fontu v bodech. Pro LaTeXovskou funkci `\fontsize`, která potom skutečně mění velikost fontu v řádku, je automaticky dopočítána výška řádku pro zvolenou velikost písma.

V parametru `align` je možné měnit zarovnání buněk v řádku písmenem `c`, `l`, `r`. Ve výsledném LaTeXovském souboru se tato volba projeví nastavením LaTeXovského makra `\multicolumn` pro všechny buňky na řádku (je tedy možné použít také volbu svislé čáry `|` pro vložení vertikální linky). Jestliže v této volbě uvedeme jediné písmeno, zarovnají se stejně všechny buňky na řádku, vektor písmen může nastavit speciální zarovnání pro každou buňku. Volba `align` zruší speciální volby ve sloupcích. Na řádku 3 ve sloupci s

	IQ		
	IQbio	IQfoster	class
~~*5*~~	~~116~~	~~115~~	~~high~~
10	**93**	**82**	**medium**
12	88	100	medium
19	97	87	low
26	107	106	low

[9] Vyžaduje LaTeXovský balík `ulem`.

označením řádků tabulky podle nastavení sloupců by měla dvanáctka být vysázena kurzívou. Podobně je zrušena volba svislé linky za posledním sloupcem.

Parametr `angle` nastaví pootočení nápisu na řádku. Úhel otočení se uvádí ve stupních. Pootočení lze nastavit pro každou buňku řádku zvlášť tím, že uvedeme v parametru `angle` vektor.

Pokud je parametr `hline_after` nastaven na `TRUE`, bude za označeným řádkem vložena horizontální linka.

Pro sloupce jsou další volbou nastavenou ve funkci `column_spec` parametry `border_left` a `border_right` pro zobrazení vertikálních linek z jedné nebo druhé strany sloupce (to už umíme zařazením znaku `|` do volby `align` funkce `kable`, ale volby `border_*` jsou obecnější, použitelné nejen pro LaTeX také pro jazyk Markdown).

Poslední funkcí použitou v tomto příkladu je `add_header_above`, která přidá další řádek nad běžné záhlaví tabulky. Také v záhlaví musíme mezi sloupce počítat i sloupec s názvy řádků, pokud je zobrazen volbou `row_names = TRUE`. Nové záhlaví bude odděleno od původního vodorovnou linkou. V LaTeXu je tato linka vytvořená makrem `\cline`, které má v argumentu pomlčku, v české lokalizaci LaTeXu aktivní znak. Před tabulkou musíme nastavit `\shorthandoff{-}` a za tabulkou zase vrátit znaku pomlčka aktivní příznak příkazem `\shorthandon{-}`.

```
kable(mydata, booktabs = TRUE, row.names = TRUE) %>%
row_spec(0, angle = -45) %>%
row_spec(1, underline = TRUE, strikeout = TRUE) %>%
row_spec(2, bold = TRUE,
  color = "white", background = "#959595") %>%
row_spec(3, align = c("c", "l", "l", "r")) %>%
row_spec(4, font_size = 8, hline_after = TRUE) %>%
column_spec(1, italic = TRUE) %>%
column_spec(4, border_right = TRUE) %>%
add_header_above(c("", "IQ" = 2), bold = TRUE)
```

Formátování jednotlivých buněk

Speciální formátování některých vybraných buněk v tabulce se nastavuje funkcí `cell_spec`. Funkce `cell_spec` má podobné parametry jako `row_spec`, ale na rozdíl od funkcí `row_spec` a `column_spec` je třeba tuto funkci použít ještě před formátováním tabulky pomocí funkce `kable`. To znamená, že musíme modifikovat data, která poté budou zpracována funkcí `kable`. Protože vlastní data by měla zůstávat po celou práci nedotčená, je vhodné pro zpracování do výstupů použít jen kopii dat.

V tabulce jsou buňky sloupce „IQfoster“ vysázeny menším osmibodovým písmem, pokud je hodnota v tomto sloupci menší než ve sloupci „IQbio“, jinak jsou vysázeny větším, dvanáctibodovým písmem. V této tabulce opět vkládáme speciální LaTeXovské sekvence, volbu fontu, přímo do buněk tabulky, takže musíme nastavit volbu `escape = FALSE`.

	IQbio	IQfoster	class
5	**116**	115	**high**
10	93	82	medium
12	88	100	medium
19	97	87	low
26	107	106	low

Funkce `cell_spec` mění datový soubor, proto musíme dodržovat pravidla programu R pro práci s datovými typy. Výsledek, který vrací funkce `cell_spec` je vždy typu „character“. Program R konvertuje na typ „character“ automaticky typ „numeric“, ale třeba typ „factor“ takové automatické konverzi nepodléhá. Protože třetí sloupec našeho datového souboru je zrovna typu „factor“, je nutné jej nejprve konvertovat na „character“, aby výstup funkce `cell_spec` byl kompatibilní s celým sloupcem. Použití funkce `cell_spec` na buňky prvního a druhého sloupce díky automatické konverzi nepůsobí problémy.

```
# nechceme měnit původní data, ale jen jejich kopii
cdata <- mydata
cdata$IQfoster <- cell_spec(cdata$IQfoster,
  font_size = ifelse(cdata$IQfoster < cdata$IQbio, 8, 12)
  )
cdata[1,1] <- cell_spec(cdata[1,1], bold = TRUE)
# třetímu sloupci musíme změnit typ z faktoru na character
cdata[,3] <- as.character(cdata[,3])
cdata[1,3] <- cell_spec(cdata[1,3], bold = TRUE)
kable(cdata, booktabs = TRUE, escape = FALSE)
```

Poznámky v tabulce

Poznámky se do tabulky vkládají bez automatického propojení značky a textu poznámky, tj. vkládá se samostatně značka do tabulky a samostatně značka a text poznámky za tabulku. Značku pro poznámky do políčka nebo popisu tabulky vkládáme funkcemi `footnote_marker_*` (hvězdička nahrazuje volbu typu označení poznámky: `alphabet`, `number`, `symbol`, `none`). Značku do buňky v tabulce vložíme přímo do dat ještě před zpracováním tabulky funkcí `kable`. Samozřejmě, že připojení této značky k datům mění typ dat v buňce na typ „character“, a musíme stejně jako u funkce `cell_spec` věnovat pozornost tomu, aby s typem „character“ byl v souladu typ příslušného sloupce.

Parametrem funkcí `footnote_marker_*` je číslo, podle kterého se vybere

konkrétní značka pro daný typ markeru, např. pro marker typu `symbol` jsou prvními čtyřmi značkami znaky *, †, ‡, §. V příkladu je nalezeno maximum a označeno poznámkou pod tabulkou. Další dvě poznámky jsou vloženy do buněk zadaných řádkovým a sloupcovým indexem.

Vlastní poznámky, tj. značku a text poznámky, vkládá funkce `footnote` za tabulku. Značky poznámek pod tabulkou se vyhodnocují v pořadí, v jakém jsou poznámky zapsány funkcí `footnote` a v závislosti na parametru `footnote_order`, který určuje pořadí pro zobrazení poznámek podle typu značky. V příkladu jsou nejprve zobrazeny poznámky značené písmeny a poté poznámka označená číslem.

Ve výsledném LaTeXovském souboru je poznámka za tabulkou vložena jako další řádek tabulky makrem `\multicolumn`. Protože šířka tabulky se řídí šířkou nejširšího řádku, což může být řádek s poznámkou, může dlouhá poznámka pod tabulkou ovlivnit šířku celé tabulky, jak tomu je v našem příkladu (viz tab. 2).

```
# nechceme měnit původní data, ale jen jejich kopii
cdata <- mydata
# nalezneme souřadnice pro maximum v numerických datech
maximum <- which(mydata[1:2] == max(mydata[1:2]), arr.ind=TRUE)
cdata[1:2][maximum] <- paste(mydata[1:2][maximum],
  footnote_marker_alphabet(2), sep="")
cdata[2,1] <- paste(cdata[2,1],
  footnote_marker_alphabet(3), sep="")
cdata[,3] <- as.character(cdata[,3])
cdata[1,3] <- paste(cdata[1,3],
  footnote_marker_number(1), sep="")
kable(cdata, booktabs = TRUE, row.names = TRUE, escape = FALSE,
  caption = paste("Poznámky v tabulce",
  footnote_marker_alphabet(1), sep="")) %>%
footnote(alphabet=c("Dlouhá poznámka způsobí roztažení tabulky",
   "Zde je maximum.", "Další poznámka v pořadí."),
    number = ("Poznámka označená číslem."),
    footnote_order = c("alphabet", "number")
   )
```

Vylepšené chování poznámek v tabulkách zavádí LaTeXovský balík se jménem `threeparttable`. Tento balík vkládá poznámky pod tabulku do samostatné části, kterou zalamuje podle šířky tabulky. Balík `threeparttable` se aktivuje stejnojmennou volbou funkce `footnote`.

Funkce `footnote` umožňuje značit před poznámky také titulky.

	IQbio	IQfoster	class
5	116	115	high
10	93	82	medium

Číslované poznámky:

[1] Poznámka zalomená na šířku tabulky.

[a] Další poznámka.

Tabulka 2: Poznámky v tabulce[a]

	IQbio	IQfoster	class
5	116[b]	115	high[1]
10	93[c]	82	medium
12	88	100	medium
19	97	87	low
26	107	106	low

[a] Dlouhá poznámka způsobí roztažení tabulky
[b] Zde je maximum.
[c] Další poznámka v pořadí.
[1] Poznámka označená číslem.

```
kable(mydata[1:2, ], booktabs = TRUE) %>%
footnote(threeparttable=TRUE, escape=FALSE,
  number = "Poznámka zalomená na šířku tabulky.",
  number_title = "Číslované poznámky:",
  alphabet = "Další poznámka."
  )
```

Seskupení řádků, přídavné linky a zvýrazněné řádky

Řádky tabulky lze sdružovat funkcí `group_rows`. Její první parametr je označení skupiny řádků. Další dva parametry jsou číselné a ukazují na první a poslední řádek dané skupiny. V následujícím kódu jsou parametry funkce `group_rows` zadány buď napevno, nebo jsou vypočteny podle obsahu posledního sloupce tabulky.

```
kable(mydata, row.names=FALSE,
  booktabs = TRUE) %>%
group_rows("Medium class", 2, 3) %>%
group_rows("Low class",
  min(which(mydata$class == "low")),
  max(which(mydata$class == "low")))
```

IQbio	IQfoster	class
116	115	high
Medium class		
93	82	medium
88	100	medium
Low class		
97	87	low
107	106	low

Pokud je třeba v tabulce výrazně oddělit řádky, přidáme linky do funkce `row_spec` nastavením parametru `hline_after` resp. `extra_latex_after`. Už víme, že funkce `row_spec` má jako první parametr číslo řádku, na který se mají aplikovat dané speciality. Parametr `extra_latex_after` může obsahovat jakoukoliv sekvenci LaTeXovských příkazů, které budou vloženy za daný řádek. Pokud použijeme např. `\cline` pro LaTeX s českou volbou pro `babel`, musíme samozřejmě před tabulkou nastavit pomlčku z aktivního znaku na písmeno.

IQbio	IQfoster	class
116	115	high
93	82	medium
88	100	medium
97	87	low
107	106	low

```
kable(mydata, booktabs = TRUE,
  row.names=FALSE) %>%
row_spec(3, hline_after = TRUE) %>%
row_spec(1, extra_latex_after = "\\cline{1-2}")
```

Volba `latex_options = "striped"` pro funkci `kable_styling` barevně zvýrazní řádky tabulky. Toto zvýraznění provede příkaz `\rowcolors` z balíku `xcolor`, ve spolupráci s příkazem `\showrowcolors`. Oba tyto příkazy vkládá `knitr` do TeXovského souboru za hlavičku tabulky. Barva zvýraznění je zvolena v parametru `stripe_color` a volí se z deklarovaných barev v závislosti na balíku `xcolor`.

IQbio	IQfoster	class
116	115	high
93	82	medium
88	100	medium
97	87	low
107	106	low

```
kable(mydata, booktabs = TRUE,
  row.names = FALSE) %>%
kable_styling(latex_options = "striped",
  stripe_color="gray!20")
```

Plovoucí prostředí pro tabulky

Do plovoucího prostředí `table` jsou tabulky umístěny, pokud obsahují parametr `caption` nebo pokud má funkce `kable_styling` nastavenu volbu `latex_options`. Parametry pro umístění tabulky na stránce jsou brány z defaultního nastavení LaTeXu[10]. Změnit nastavení pro umístění jediné tabulky můžeme funkcí `kable_styling` volbou `latex_options`. Parametr `[h]` bude zapsán nastavením `hold_position` a parametr `[H]` bude nastaven volbou `HOLD_position`. Pro druhou možnost potřebujeme mít v LaTeXu načtený balíček `float`.

[10] Globálně se tyto volby se mění definicí v hlavičce dokumentu, např. `\def\fps@table{htbp}`.

Tabulky obtékané textem zvládá LaTeXovské prostředí `wraptable` z balíku `wrapfig`. Do tohoto prostředí se tabulka zapisuje volbou `position` nastavenou buďto na `float_right`, nebo `float_left`. Tabulka bude zařazena do následujícího odstavce. Prostředí `wraptable` vynechává pro tabulku místo podle rozměrů tabulky. Obvykle je ale vhodné nastavit počet vynechaných řádků pro tabulku ručně. Např. předefinování `\def\WF@wli{7}` vyhradí pro tabulku místo vysoké jako sedm řádků. Samozřejmě musíme předem změnit kategorii pro zavináč na kategorii písmene makrem `\makeatletter` a potom zase zpět na kategorii makrem `\makeatother`.

IQbio	IQfoster	class
116	115	high
93	82	medium
88	100	medium

```
kable(mydata[1:3,], booktabs = TRUE, row.names=FALSE) %>%
kable_styling(position = "float_right")
```

Naneštěstí funkce `kable_styling` zapíše pro prostředí `wraptable` parametr pro šířku tabulky 0pt (sázej tabulku podle její šířky), což působí kolize s nadpisem před tabulkou, a funkce `kable` neumí zařadit popisek až za tabulku, takže s popisky obtékaných tabulek nastávají potíže. Podobné potíže způsobí, jestliže byla při vytváření tabulky dvakrát použita funkce `kable_styling` (z toho jednou s volbou `latex_options`).

Tabulky s volbou `longtable`

Tabulky, které se rozměrem nevejdou na jedinou stranu, je vhodné sázet za pomoci prostředí `longtable`. V následujícím příkladu je tabulka se zvoleným prostředím `longtable`, ve které je přehlednost zajištěna podbarvením sudých řádků. Protože řádky jsou podbarveny, není zapotřebí oddělovat skupiny řádků separátorem `\addlinespace` z parametru `linesep`. Parametr `repeat_header_text` obsahuje text, který v záhlaví na dalších stranách oznamuje pokračování tabulky. Parametr `repeat_header_method` je defaultně nastaven na `append`, což znamená, že se na dalších stranách opakuje celý popisek z `caption`, nastavení `replace` zobrazí na další straně jen text z `repeat_header_text`.

Tabulka 3: Dlouhá tabulka

IQbio	IQfoster	class
82	82	high
80	90	high
88	91	high

Tabulka 3: Pokračování tabulky

IQbio	IQfoster	class
108	115	high
116	115	high
117	129	high
132	131	high
71	78	medium
75	79	medium
93	82	medium

```
mydata <- carData::Burt[1:10,]
kable(mydata, longtable = TRUE, linesep = "", booktabs = TRUE,
  caption = "Dlouhá tabulka") %>%
kable_styling(latex_options = c("striped", "repeat_header"),
  repeat_header_method = "replace",
  repeat_header_text = "Pokračování tabulky",
  stripe_color="gray!20")
```

Nastavení LaTeXu

V předchozím textu bylo zmíněno několik balíků, které jsou zapotřebí pro sazbu tabulek produkovaných v programu R funkcí `kable`. Pro shrnutí jsou zde uvedeny balíky, které by při překladu dokumentu mohly být zapotřebí.

```
\usepackage{booktabs}       % tabulky oddělené linkami
\usepackage[table]{xcolor}  % podbarvené řádky
\usepackage{makecell}       % speciální formát jednotlivých buněk
\usepackage{threeparttable} % poznámky pod tabulkou
\usepackage{float}          % volba [H] pro plovoucí tabulky
\usepackage{wrapfig}        % obtékané tabulky
\usepackage[normalem]{ulem} % přeškrtnuté písmo
\usepackage{longtable}      % dlouhé tabulky
```

Summary: Tables in well-documented statistical calculations

The article describes the method of publishing well-documented statistical calculations using software R. It is specially about creating tables using `knitr` and `kableExtra`.

Reference

[1] Fear, S. Publication quality tables in LaTeX, 2016. https://ctan.org/pkg/booktabs.

[2] Gandrud, C. *Reproducible Research with R and R Studio.* Chapman and Hall/CRC, Boca Raton, 2013.

[3] Leisch, F. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat 2002 — Proceedings in Computational Statistics* (2002), W. Härdle and B. Rönz, Eds., Physica Verlag, Heidelberg, pp. 575–580. http://www.stat.uni-muenchen.de/~leisch/Sweave.

[4] Pomp, M. Dobře dokumentované statistické výpočty. *Zpravodaj Československého sdružení uživatelů TEXu*, 1–4 (2016), 62–78.

[5] RStudio Team. *RStudio: Integrated Development Environment for R.* RStudio, Inc., Boston, MA, 2015. http://www.rstudio.com/.

[6] Xie, Y. *Dynamic documents with R and knitr.* Chapman and Hall/CRC, Boca Raton, 2015.

[7] Zhu, H. Create awesome latex table with knitr::kable and kableextra, 2018. https://haozhu233.github.io/kableExtra/awesome_table_in_pdf.pdf.

Poděkování: Autor by rád poděkoval recenzentovi za cenné podněty a připomínky, které pomohly tento text vylepšit.

Marek Pomp, EkF VŠB-TU Ostrava

LuaTEX version 1.0.0

Hans Hagen

After ten years of development, the first stable version of the LuaTEX engine was released at the 10th International ConTEXt Meeting 2016. The article describes the beginnings, the development, and the future of LuaTEX.

Keywords: Lua, LuaTEX, ConTEXt

The release

After some ten years of development and testing, on September 9, 2016, we released LuaTEX 1.0.0! It happened at the tenth meeting of the ConTEXt users and developers group in the Netherlands.

Instead of staying below one and ending up with versions like 0.99.1234, we decided that the moment was there to show the TEX audience that LuaTEX is stable enough to lose its beta status. Although functionality has evolved and sometimes been replaced, we have been using LuaTEX ourselves in production right from the start. Of course there are bugs and for sure we will fix them.

Our main objective was and still is to provide a variant of TEX that permits user extensions without the need to adapt the inner workings. We did add a

doi: 10.5300/2018-1-4/38

few things here and there but they mostly relate to opening up the inner parts and/or the wish to influence some hard-coded behaviour. Via Lua we managed to support modern functionality without bloating the code or adding more and more dependencies on foreign code. In the process, a stable and flexible MetaPost library became part of the engine.

The functionality as present now will stay. We might open up some more parts, we will stepwise clean up the code base while staying as close as possible to the Knuthian original, we will try to document bits and pieces. We might also experiment a bit with better isolation of the backend, and simplify some internals. For that we can use the experimental version but if we diverge too much we may need to give that another name.

We want to thank all those who have tested the betas and helped to make LuaTeX better.

Hans Hagen
Hartmut Henkel
Taco Hoekwater
Luigi Scarso

The past

Originally we planned to release the first version a few years ago but our ambitions didn't work out well with that schedule so we finally took a decade to get there. For the record it is good to summarize what happened during those years.

- Around 2005, after we talked a bit about extending TeX in a flexible way and Hartmut added the Lua scripting language to pdfTeX as an experiment. This add-on was inspired by the Lua extension to the SciTe editor that I (still) use.
- At that time one could query counter registers and box dimensions and print strings to the TeX input buffer.
- The Oriental TeX project then made it possible to go forward and come up with a complete interface. For this, Taco converted the code base from Pascal to C, quite an impressive effort.
- We spent more than a year intensively discussing, testing and implementing the interface between TeX and Lua. Many binaries and lots of test code were flying between Taco and my machine as we progressed and decided what directions to go. These were really interesting times.
- In successive years, we polished things and extended bits and pieces and in recent years we cleaned up interfaces, polished some code, filled in gaps and reached the point where we were more or less satisfied.
- The core is still traditional TeX, but has been extended with pdfTeX protrusion and expansion (reworked) and directional features from Aleph (cleaned

up). We did add some extensions (in ε-TEX fashion) but removed most of the ones that we inherited from pdfTEX because LUA could do better.

- The backend and extension interfaces are now mostly separated and although we don't expect to add more backends, it makes the code somewhat cleaner because all kinds of PDF-related issues are no longer mixed with front-end mechanisms.
- The font subsystem is no longer limited to 8-bit fonts. It must be noted that, for instance, OpenType support is done in LUA, which provides a lot of flexibility. This also serves as an example of extensibility. A small TEX core, independent of libraries, was definitely an objective and it works out well.
- The (rewritten but compatible) hyphenation machinery can use runtime loaded (and extended) patterns. There are a few extensions and, of course, one can revert to LUA for more.
- Already at an early stage, hyphenation, ligaturing and kerning were separated, which was one step in adding callbacks to nearly every stage in the typesetting process.
- Math supports wide (more than 8-bit) characters too so that one can implement Unicode math easily. The machinery has OpenType math code paths because there are some fundamental differences with traditional TEX math fonts.
- Although the kpse library is still the default interface to the file system, all in- and output can be controlled and intercepted, for instance for input filtering or re-encoding on the fly.
- The token scanner has been opened up so that one can write (simple) parsers. Experimental interception code didn't prove to be useful, so that interface has been dropped. We kept it simple and efficient.
- During callbacks related to the node lists, individual nodes can be accessed and manipulated at will. Of course, one needs to know a bit about the internals and not mess up the lists to the extent that TEX will choke on it: things that 'can't happen' now can. Most of the original documentation of the code by Don Knuth still applies (which was another objective) but of course directional support and such go beyond that. And it's surprisingly fast.
- Images and reusable boxes are now native nodes; they travel through the system as special kinds of rules instead of whatsits with dimensions. Users can define their own rule types too.

There is more to say but much has been reported already in articles in this and other journals. In the CONTEXT distribution, there are four documents describing aspects of the development and choices we have made (`mkiv.pdf`, `hybrid.pdf`, `about.pdf` and `still.pdf`) and we keep writing (`onandon.pdf`). One thing will hopefully be clear by now: the choice of LUA was a good one.

The future

The project is driven by ConTeXt users and ConTeXt development, which is why we found it proper to release version one at the tenth meeting. Right from the start, ConTeXt supported LuaTeX and this means that most mechanisms have been tested in production. There is some risk in this as users then are always forced to update the binary with the macros, but the ConTeXt garden provides easy ways to deal with this. In fact, most users switched to the new engine pretty soon after we started rewriting ConTeXt. We greatly appreciate their patience with us.

Raw performance of LuaTeX is of course less than 8-bit pdfTeX but in practice and on modern machines LuaTeX behaves well. In fact, many mechanisms, like native xml handling and MetaPost processing, are way faster in ConTeXt MkIV then in the now frozen MkII version. Given the fact that we're using Unicode and more complex fonts, one can safely assume that in ConTeXt, the overhead due to delegation to Lua has no real drawbacks.

We will continue development, but functionality will stay stable within versions. The code will be further streamlined and documented. We deliberately postponed some cleanup till after version one. And, of course, bugs will be fixed. We hope to stepwise improve the manual too. So what will the future bring?

- So far, we have managed to avoid extensions beyond those needed as part of the opening up. We stick close to Don Knuth's concepts so that existing documentation still conceptually applies. We keep our promise of not adding to the core. But we might open up (make configurable) some of the remaining hard-coded properties.
- Some node lists can use a bit of (non-critical) cleanup, for instance passive nodes, localpar nodes, and other leftovers. Maybe we should add missing left/right skips.
- We can optimize some callback resolution (more direct) so that we can gain a little performance.
- Inheritance of attributes needs checking and maybe we need to permit some more explicit settings.
- We will move some more code to the api file and plan to update the global pdf and Lua states consistently (there are some leftovers from the early days). Some C macros can probably go away.
- We can possibly minimize some return values of Lua functions and only return nil when we expect multiple calls in line. This might be more efficient. We plan to look into Lua 5.3 but we might well conclude that it's better to stick with 5.2.
- We have to figure out a way to deal with literals in virtual characters. This relates to font switching in the result.

- Maybe we will reorganize some code, so that documentation is easier. We hope to continue to stick close to what Don Knuth documents.
- We can clean up and isolate the backend a bit more. We could also add a few more options to delegate actions to Lua and we should get rid of some historic pdf artifacts.

Of course, we have some ideas of what to do next but these don't need an extension to the engine because we can use Lua for that.

In that perspective, it is tempting to think of a (lean and mean) LuaTEX variant for ConTEXt: one close to the traditional core with many hooks and a minimal number of dependencies on libraries and such. In a ConTEXt setup, a user only needs LuaTEX because all (workflow) related scripts are written in Lua and if additional functionality (like graphic conversions) is needed, it can easily be provided by external programs.

We will not touch the stable version unless it concerns bug fixes and/or simple extensions, but we will keep exposing ConTEXt users to the experimental branch (as we do now). Of course users of other macro packages can pick up binaries from the compile farm that has been set up by Mojca and friends.

So ... be prepared.

Verze 1.0.0 stroje LuaTEX

Po deseti letech vývoje byla na jubilejním desátém Mezinárodním setkání uživatelů ConTEXtu v roce 2016 vydána první stabilní verze TEXového stroje LuaTEX. Článek popisuje začátky, vývoj a budoucnost LuaTEXu.

Klíčová slova: Lua, LuaTEX, ConTEXt

Hans Hagen, pragma@wxs.nl

Poznámka redakce: Začátkem roku 2019 dojde k vydání verze 1.1.0 stroje LuaTEX. Tato verze bude součástí distribuce TEX Live 2019. Změny od verze 1.0.0 zahrnují přechod z verze 5.2 jazyka Lua na verzi 5.3, náhradu knihovny poppler pro čtení pdf dokumentů za knihovnu pplib a podporu procesorové architektury arm.

Emoji Again

Hans Hagen

Since the 10th International ConTeXt Meeting in 2016, ConTeXt has supported the OpenType `colr` and `cpal` tables that are used in color fonts and also to produce emoji. The article introduces emoji and uses the Microsoft's seguiemj font to show how emoji are constructed from glyphs, how emoji can be stacked into sequences, and how the palettes of a color font can be changed in ConTeXt.

Keywords: Lua, LuaTeX, ConTeXt, OpenType, emoji

Because at the ConTeXt 2016 meeting color fonts[1] were on the agenda, some time was spent on emoji (these colorful small picture glyphs). When possible I bring kids to the BachoTeX conference so for the 2017 BachoTUG I decided to do something with emoji that, after all, are mostly used by those younger than I am. So, I had to take a look at the current state. Here are some observations.

The Unicode standard defines a whole lot of emoji and if mankind manages to survive for a while one can assume that a lot more will be added. After all, icons as well as variants keep evolving. There are several ways to organize these symbols in groups but I will not give grouping a try. Just visit `emojipedia.org` and you get served well. For this story I only mention that:

- There are quite some shapes and nearly all of them are in color. The yellow ones, smilies and such, are quite prominently present but there are many more.
- A special subset is filled by persons: man, woman, girl, boy and recently a baby.
- The grown ups can be combined in loving couples (either or not kissing) and then can form families, but only upto 2 young kids or gender neutral babies.
- All persons can be flagged with one of five skin tones so that not all persons (or heads) look bright yellow.
- Interesting is that girls and boys are still fond of magenta (pinkish) and cyan (blueish) cloths and ornaments. Also haircuts are rather specific to the gender.

For rendering color emojis we have a few color related OpenType font properties available: bitmaps, svg, and stacked glyphs. Now, if you think of the

[1] For that occasion the cowfont, a practical joke concerning Dutch 'koeieletters', were turned into a color font and presented at the meeting.

doi: 10.5300/2018-1-4/43

combinations that can be made with skin tones, you realize that fonts can become pretty large if each combination results in a glyph. In the first half of 2017 Microsoft released an update for its emoji font and the company took the challenge to provide not only mixed skin tone couples, but also supported skin tones for the kids, including a baby.

This recent addition already adds over 25,000 additional glyphs[1] so imagine what will happen in the future. But, instead of making a picture for each variant, a different solution has been chosen. For coloring this seguiemj font uses the (very flexible) stacking technology: a color shape is an overlay of colored symbols. The colors are organized in palettes and it's no big deal to add additional palettes if needed. Instead of adding pre-composed shapes (as is needed with bitmaps and svg) snippets are used to build alternative glyphs and these can be combined into new shapes by substitution and positioning (for that kerns, mark anchoring and distance compensation is used).

So, a family can be constructed of composed shapes (man, woman, etc) that each are composed of snippets (skull, hair, mouth, eyes). So, effectively a family of four is a bunch of maybe 25 small glyphs overlayed and colored. In Figure 1 we see how a shape is constructed out of separate glyphs. Figure 2 shows how they can be overlayed with colors (we use a dedicated color set).

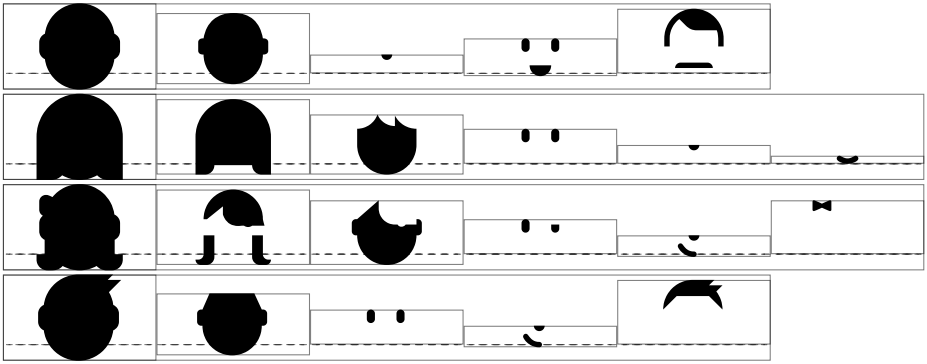


Figure 1 Emoji snippets.

When a font supports it, a sequence of emoji can be turned into a more compact representation. In Figure 3 we see how skin tones are applied in such combinations. Figure 4 shows the small snippets.

[1] That is the amount I counted when I added all combinations runtime but the emojipedia mentions twice that amount. Currently in ConTeXt we resolve such combinations when requested.

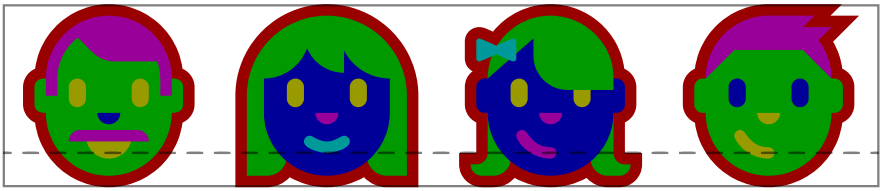


Figure 2 Emoji snippets overlayed.

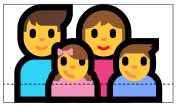
When we have to choose a font we need to take the following criteria into account:

- What is the quality of the shapes? For sure, outlines are best if you want to scale too.
- How efficiently is a shape constructed? In that respect a bitmap or SVG image is just one entity.
- How well can a (semi)arbitrary combinations of emoji be provided? Here the glyph approach wins.
- Are all skin colors for all human relates shapes supported? Actually it opens the possibility for racist fonts.
- Are all reasonable combinations of persons supported? It looks like (depending on time and version) kissing men or women can be missing, maybe because of social political reasons.
- Are black and white shapes provided alongside color shapes?

Maybe an SVG or bitmap image can have a lot of detail compared to a stacked glyph but, when we're just using pictographic representations, the latter is the best choice.

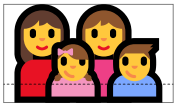
When I was playing a bit with the skin tone variants and other combinations that should result in some composed shape, I used the UNICODE test files but I got the impression that there were some errors in the test suite, for instance with respect to modifiers. Maybe the fonts are just doing the wrong thing or maybe some implement these sequences a bit inconsistent. This will probably improve over time but the question is if we should intercept issues. I'm not in favour of this because it adds more and more fuzzy code that not only wastes cycles (energy) but is also a conceptual horror. So, when testing, imperfection has to be accepted for now. This is no big deal as until now no one ever asked for emoji support in ConTeXt.

When no combined shape is provided, the original sequence shows up. A side effect can be that zero-width-joiners and modifiers become visible. This

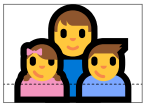


family man
woman
girl boy

family
woman
girl boy



family woman
woman
girl boy



family man
girl boy

family man
dark skin
tone woman
girl baby

family
man light
skin tone
woman
light
skin tone
girl dark
skin tone

family
woman
girl boy

family man
light skin tone
woman dark
skin tone
girl medium
skin tone
boy medium
skin tone

Figure 3 Emoji families and such with skin tones.

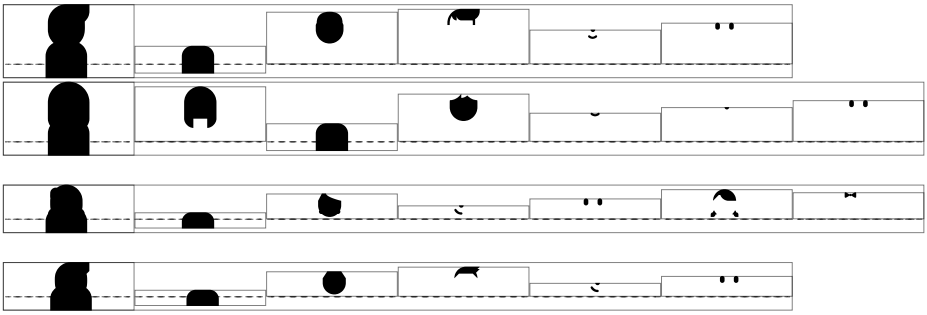


Figure 4 Emoji glyphs.

depends on the fonts. Users probably don't care that much about it. Now how do we suppose that users enter these emoji (sequences) in a document source? One can imagine a pop up in the editor but TEXies are often using commands for special cases.

We already showed some combined shapes. The reader might appreciate the outcome but getting there from the input takes a bit of work. For instance a two person `man light skin tone woman medium skin tone girl medium-light skin tone baby medium-light skin tone` involves this:

font 11: seguiemj.ttf @ 10.0pt

features [**basic**: ccmp=yes, dist=yes, mark=yes, mkmk=yes, script=dflt, tlig=yes, trep=yes] [**extra**: analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, colr=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mathkerns=yes, mathrules=yes, mode=node, spacekern=yes]

step 1  [+TLT] U+1F468: U+1F3FB: U+200D: ‍ U+1F469: U+1F3FD: U+200D: ‍ U+1F467: U+1F3FC: U+200D: ‍ U+1F476: U+1F3FC:

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0', replacing U+1F468 upto U+1F3FB by ligature U+F01C5 case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0', replacing U+1F469 upto U+1F3FD by ligature U+F01D2 case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0', replacing U+1F467 upto U+1F3FC by ligature U+F01BC case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0', replacing U+1F476 upto U+1F3FC by ligature U+F021A case 2

step 2  [+TLT] U+F01C5: U+200D: ‍ U+F01D2: U+200D: ‍ U+F01BC: U+200D: ‍ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_2', replacing single U+F01C5 by U+F15C4

step 3  [+TLT] U+F15C4: U+200D: ‍ U+F01D2: U+200D: ‍ U+F01BC: U+200D: ‍ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_3', index 1, replacing character U+200D upto U+F01D2 by ligature U+F15EC case 4

step 4  [+TLT] U+F15C4: U+F15EC: U+200D: ‍ U+F01BC: U+200D: ‍ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_5', index 1, replacing character U+200D upto U+F01BC by ligature U+F15B9 case 4

step 5  [+TLT] U+F15C4: U+F15EC: U+F15B9: U+200D: U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_6', index 1, replacing character U+200D upto U+F021A by ligature U+F1607 case 4

step 6  [+TLT] U+F15C4: U+F15EC: U+F15B9: U+F1607:

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_7', replacing single U+F15B9 by U+F15AC

step 7  [+TLT] U+F15C4: U+F15EC: U+F15AC: U+F1607:

feature 'dist', type 'gpos_single', lookup 'p_s_0', shifting single U+F15C4 by single xy (1.25pt,0pt) and wh (0pt,0pt)

step 8  [+TLT] U+F15C4: U+F15EC: U+F15AC: U+F1607:

feature 'dist', type 'gpos_single', lookup 'p_s_1', shifting single U+F15EC by single xy (0pt,0pt) and wh (1.25pt,0pt)

step 9  [+TLT] U+F15C4: U+F15EC: [kern] U+F15AC: U+F1607:

feature 'dist', type 'gpos_contextchain', chain lookup 'p_s_2', shifting single U+F15C4 by single (0pt,0pt) and correction (1.25pt,0pt)

step 10  [+TLT][kern] U+F15C4: U+F15EC: [kern] U+F15AC: U+F1607:

feature 'dist', type 'gpos_contextchain', chain lookup 'p_s_3', shifting single U+F15EC by single (4.76074pt,0pt) and correction (0pt,0pt)

feature 'dist', type 'gpos_contextchain', chain lookup 'p_s_3', shifting single U+F15EC by single (0pt,0pt) and correction (8.27148pt,0pt)

step 11  [+TLT][kern] U+F15C4:  [kern] U+F15EC: [kern] U+F15AC:  U+F1607: 

feature 'dist', type 'gpos_contextchain', chain lookup 'p_s_5', shifting single U+F15C4 by single (0pt,0pt) and correction (-4.76074pt,0pt)

step 12  [+TLT][kern] U+F15C4:  [kern][kern] U+F15EC:  [kern] U+F15AC:  U+F1607: 

feature 'mark', type 'gpos_mark2base', lookup 'p_s_27', bound 1, anchoring mark U+F15AC to basechar U+F15EC => (6.32812pt,0pt)

step 13  [+TLT][kern] U+F15C4:  [kern][kern] U+F15EC:  [kern] U+F15AC:  U+F1607: 

feature 'mark', type 'gpos_mark2base', lookup 'p_s_28', bound 2, anchoring mark U+F1607 to basechar U+F15EC => (0.00977pt,0pt)

result  [+TLT][kern] U+F15C4:  [kern][kern] U+F15EC:  [kern] U+F15AC:  U+F1607: 

A black and white example is the following `family woman girl`:

font 17: seguiemj.ttf @ 10.0pt

features **[basic:** ccmp=yes, dist=yes, mark=yes, mkmk=yes, script=dflt, tlig=yes, trep=yes] **[extra:** analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mathkerns=yes, mathrules=yes, mode=node, spacekern=yes]

step 1  [+TLT] U+1F469:  U+200D:  U+1F467: 

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_4', replacing single U+1F469 by U+F15E2

step 2  [+TLT] U+F15E2:  U+200D:  U+1F467: 

feature 'ccmp', type 'gsub_contextchain', chain lookup 's_s_5', index 1, replacing character U+200D upto U+1F467 by ligature U+F15B0 case 4

step 3 [+TLT] U+F15E2: U+F15B0:

```
feature 'dist', type 'gpos_single', lookup 'p_s_1',
  shifting single U+F15E2 by single xy (0pt,0pt) and
  wh (1.25pt,0pt)
```

step 4 [+TLT] U+F15E2: [kern] U+F15B0:

```
feature 'dist', type 'gpos_contextchain', chain lookup
  'p_s_4', shifting single U+F15E2 by single
  (1.25pt,0pt) and correction (0pt,0pt)

feature 'dist', type 'gpos_contextchain', chain lookup
  'p_s_4', shifting single U+F15E2 by single (0pt,0pt)
  and correction (4.76074pt,0pt)
```

step 5 [+TLT][kern] U+F15E2: [kern] U+F15B0:

```
feature 'mark', type 'gpos_mark2base', lookup
  'p_s_28', bound 1, anchoring mark U+F15B0 to
  basechar U+F15E2 => (0.00977pt,0pt)
```

result [+TLT][kern] U+F15E2: [kern] U+F15B0:

I will not show all emoji, just the subset that contains the word `woman` in the description. As you can see the persons in the sequences are separated by a zero-width-joiner. There are some curious ones, for instance a `woman wearing turban` which in terms of UNICODE input is a female combine with a turban wearing man becomes a beardless woman wearing a turban. Woman vampires and zombies are not supported so these are male properties.

		`blondhaired woman`
		`couple with heart woman man`
		`couple with heart woman woman`
		`family man woman boy`
		`family man woman boy boy`
		`family man woman girl`
		`family man woman girl boy`
		`family man woman girl girl`
		`family woman boy`
		`family woman boy boy`
		`family woman girl`

  family woman girl boy

  family woman girl girl

  family woman woman boy

  family woman woman boy boy

  family woman woman girl

  family woman woman girl boy

  family woman woman girl girl

  kiss woman man

  kiss woman woman

  man and woman holding hands

  old woman

  pregnant woman

  woman

  woman artist

  woman astronaut

  woman biking

  woman boot

  woman bouncing ball

  woman bowing

  woman cartwheeling

  woman climbing

  woman clothes

  woman construction worker

  woman cook

  woman dancing

  woman detective

  woman elf

  woman facepalming

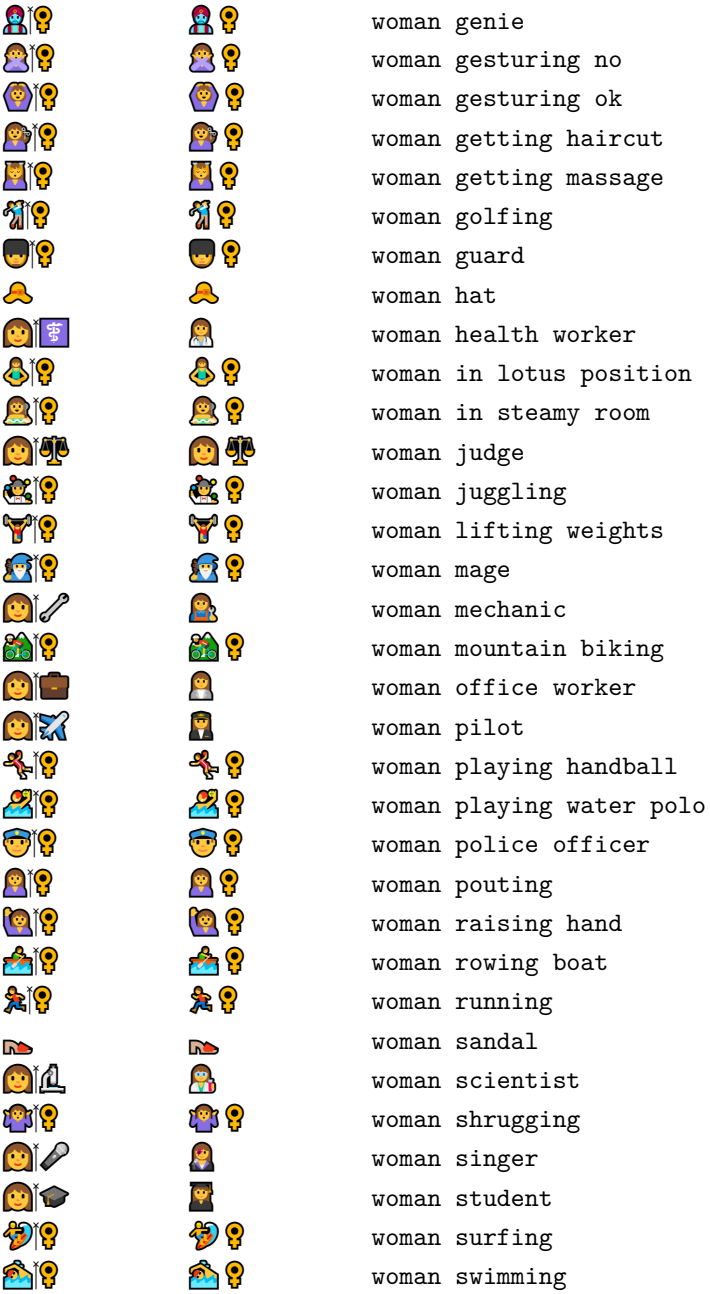
  woman factory worker

  woman fairy

  woman farmer

  woman firefighter

  woman frowning


woman genie
woman gesturing no
woman gesturing ok
woman getting haircut
woman getting massage
woman golfing
woman guard
woman hat
woman health worker
woman in lotus position
woman in steamy room
woman judge
woman juggling
woman lifting weights
woman mage
woman mechanic
woman mountain biking
woman office worker
woman pilot
woman playing handball
woman playing water polo
woman police officer
woman pouting
woman raising hand
woman rowing boat
woman running
woman sandal
woman scientist
woman shrugging
woman singer
woman student
woman surfing
woman swimming

woman teacher
woman technologist
woman tipping hand
woman vampire
woman walking
woman wearing turban
woman with headscarf
woman zombie

So what if you don't like these colors? Because we're dealing with TEX you can assume that if there is some way around the fixed color sets, then it will be provided. So, when you use CONTEXT, here is a way to overload them:

```
\definecolor[emoji-red][r=.4]
\definecolor[emoji-green][g=.4]
\definecolor[emoji-blue][b=.4]
\definecolor[emoji-yellow][r=.4,g=.4]
\definecolor[emoji-gray][s=1,t=.5,a=1]

\definefontcolorpalette[emoji-s]
  [black,emoji-gray]
\definefontcolorpalette[emoji-r]
  [emoji-red,emoji-gray]
\definefontcolorpalette[emoji-g]
  [emoji-green,emoji-gray]
\definefontcolorpalette[emoji-b]
  [emoji-blue,emoji-gray]
\definefontcolorpalette[emoji-y]
  [emoji-yellow,emoji-gray]

\definefontfeature[seguiemj-s]
  [ccmp=yes,dist=yes,colr=emoji-s]
\definefontfeature[seguiemj-r]
  [ccmp=yes,dist=yes,colr=emoji-r]
\definefontfeature[seguiemj-g]
  [ccmp=yes,dist=yes,colr=emoji-g]
\definefontfeature[seguiemj-b]
  [ccmp=yes,dist=yes,colr=emoji-b]
\definefontfeature[seguiemj-y]
  [ccmp=yes,dist=yes,colr=emoji-y]

\definefont[MyEmojiS]
```

```
  [seguiemj*seguiemj-s]
\definefont[MyEmojiR]
  [seguiemj*seguiemj-r]
\definefont[MyEmojiG]
  [seguiemj*seguiemj-g]
\definefont[MyEmojiB]
  [seguiemj*seguiemj-b]
\definefont[MyEmojiY]
  [seguiemj*seguiemj-y]
```

Figure 5 Overloading colors by plugging in a sequence of alternate colors.

In Figure 5 we see how this is applied. You can provide as many colors as needed but when you don't provide enough the last one is used. This way we get the overlayed transparent colors in the examples. By using transparency we don't obscure shapes.

The emojipedia mentions "Asked about the design, MICROSOFT told emojipedia that one of the reasons for the thick stroke was to allow each emoji to be easily read on any background color." The first glyph in the stack seems to do the trick, so just make sure that it doesn't become white. And, before I read that remark, while preparing a presentation with a colored background, I had

already noticed that using a background was no problem. This font definitely sets the standard.

How do we know what colors are used? The next table shows the first color palette of seguiemj. There are quite some colors so defining your own definitely involved some studying.

1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121
122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154
155	156	157	158	159	160	161	162	163	164	165
166	167	168	169	170	171	172	173	174	175	

Normally special symbols are accessed in ConTeXt with the `symbol` command where symbols are organized in symbol sets. This is a rather old mechanism and dates from the time that fonts were limited in coverage and symbols were collected in special fonts. The emoji are accessed by their own command: `\emoji`. The font used has the font synonym `emoji` so you need to set that one first:

```
\definefontsynonym
  [emoji]
  [seguiemj*seguiemj-cl]
```

Here is an example:

```
\emoji{woman light skin tone}\quad
\emoji{woman scientist}\quad
{\bfd bigger
 \emoji{man health worker}}
```

or typeset: 👩🏻 👩‍🔬 **bigger** 👨‍⚕️

The emoji symbol scales with the normal running font. When you ask for a family with skin toned members the lookup can result in another match (or no match) because one never knows to what extend a font supports it.

`\expandedemoji`	the sequence constructed from the given string
`\resolvedemoji`	a protected sequence constructed from the given string
`\checkedemoji`	a typeset sequence with unresolved modifiers and joiners removed
`\emoji`	a typeset resolved sequence using the `emoji` font synonym
`\robustemoji`	a typeset checked sequence using the `emoji` font synonym

In case you wonder how some of the details above were typeset, there is a module `fonts-emoji` that provides some helpers for introspection.

`\ShowEmoji`	show all the emoji in the current font
`\ShowEmojiSnippets`	show the snippets of a given emoji
`\ShowEmojiSnippetsOverlay`	show the overlayed snippets of a given emoji
`\ShowEmojiGlyphs`	show the snippets of a typeset emoji
`\ShowEmojiPalettes`	show the color palettes in the current font

Examples of usage are:

```
\ShowEmojiSnippets
  [family man woman girl boy]
\ShowEmojiGlyphs
  [family man woman baby girl]
\ShowEmoji        [^man]
\ShowEmoji
\ShowEmojiPalettes
\ShowEmojiPalettes[1]
```

A good source of information about emoji is the mentioned `emojipedia.org` website. There you find not only details about all these symbols but also has some history. It compares updates in fonts too. It mentions for instance that in the creative update of Windows 10, some persons grew beards in the seguiemj font and others lost an eye. Now, if you look at the snippets shown before, you can wonder if that eye is really gone. Maybe the color is wrong or the order of stacking is not right. I decided not to waste time looking into that.

Another quote: "Support for color emoji presentation on MS WINDOWS is limited. Many applications on MS WINDOWS display emojis with a black and white text presentation instead of their color version." Well, we can do better with TEX, but as usual not that many people really cares about that. But it's fun anyway.

We end with a warning. When you use 'ligatures' like this, you really need to check the outcome. For instance, when MICROSOFT updated the font in 2017, same gender couples got different hair style for the individuals so that one can still distinguish them. However, kissing couples and couples in love (indicated by a heart) seem to be removed. Who knows how and when politics creep into fonts:

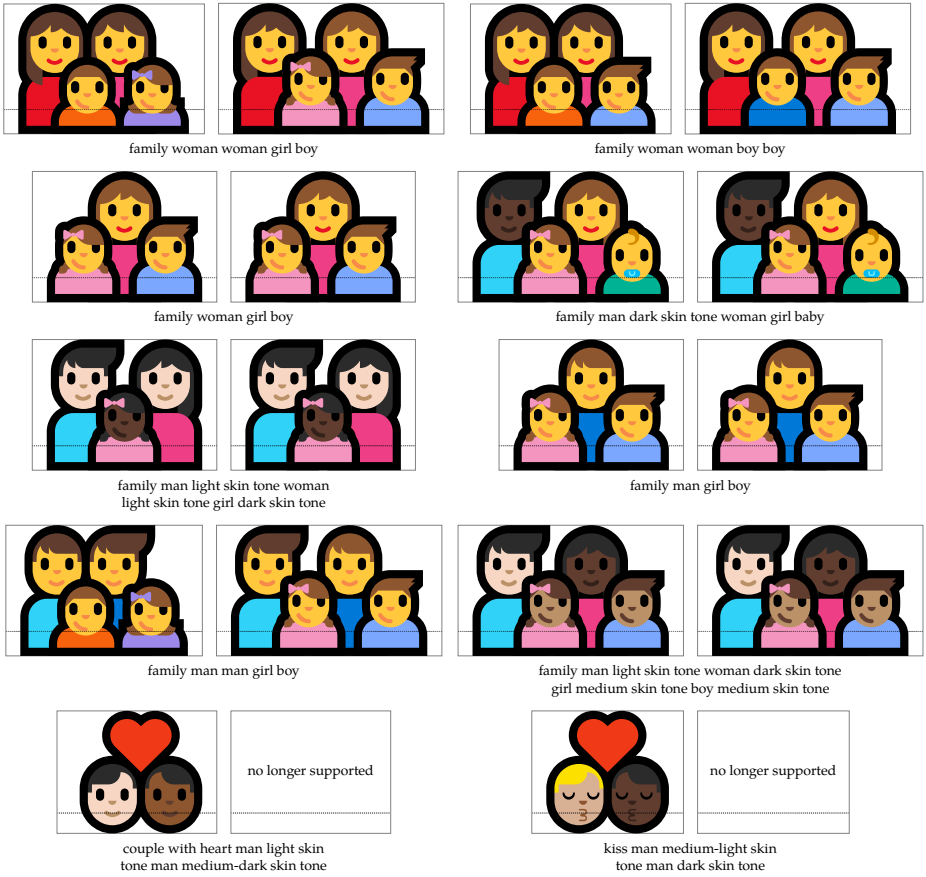


Figure 6 Incompatible updates.

is public mixed couple kissing permitted, do we support families with any mix of gender, is associating pink with girls okay or not, how do we distinguish male and female anyway? In Figure 6 we see the same combination twice, the early 2017 rendering versus the late 2017 rendering. Can you notice the differences?

Zase emoji

Od desátého Mezinárodního setkání uživatelů ConTeXtu v roce 2016 podporuje ConTeXt opentypové tabulky `colr` a `cpal`, které se používají v barevných písmech a také pro přípravu emoji. Článek představuje emoji a na příkladu písma

seguiemj od firmy Microsoft ukazuje, jak jsou znaky emoji sestaveny, jak je lze spojovat do posloupností a jak lze v ConTEXTu upravit barevnou paletu písma.

Klíčová slova: Lua, LuaTEX, ConTEXt, OpenType, emoji

Hans Hagen, pragma@wxs.nl

ConTEXt Performance

Hans Hagen

The processing speed of a TEX engine is affected by a number of factors, such as the format, macros, scripting, fonts, microtypographic extensions, SyncTEX, and command-line redirection. The article discusses the individual factors from the perspective of a ConTEXt user. The article also measures the overhead of ConTEXt MkII and MkIV, the impact of command-line redirection on the speed of ConTEXt MkIV, the impact of fonts on the speed of typesetting with ConTEXt MkIV, and the speed of typesetting with ConTEXt MkII and MkIV.

Keywords: Lua, LuaTEX, LuajitTEX, ConTEXt MkII, ConTEXt MkIV

Introduction

This article is about performance. Although it concerns LuaTEX this text is only meant for ConTEXt users. This is not because they ever complain about performance, on the contrary, I never received a complain from them. No, it's because it gives them some ammunition against the occasionally occurring nagging about the speed of LuaTEX (somewhere on the web or at some meetings). My experience is that, in most such cases, those complaining have no clue what they're talking about, so effectively we could just ignore them, but let's, for the sake of our users, waste some words on the issue.

What performance

So, what exactly does performance refer to? If you use ConTEXt, there are probably only two things that matter:

- How long does one run take?
- How many runs do I need?

Processing speed is reported at the end of a run in terms of seconds spent on the run, but also in pages per second. The runtime is made up of three components:

- start-up time,
- processing pages, and
- finishing the document.

The startup time is rather constant. Let's take my 2013 Dell Precision with i7-3840QM as reference. A simple

```
\starttext
\stoptext
```

doi: 10.5300/2018-1-4/59

document reports 0.4 seconds but, as we wrap the run in an `mtxrun` management run, we have an additional 0.3 overhead (auxiliary file handling, PDF viewer management, etc). This includes loading the Latin Modern font. With LuajitTeX, these times are below 0.3 and 0.2 seconds. It might look like a lot of overhead, but it feels snappy in an edit-preview runs. One can try this:

```
\stoptext
```

which bring down the time to about 0.2 seconds for both engines, but it doesn't do anything useful in practice.

Finishing a document is not that demanding, because most gets flushed as we go. The more (large) fonts we use, the longer it takes to finish a document, but, on the average, that time is not worth noticing. The main runtime contribution comes from processing the pages.

Okay, this is not always true. For instance, if we process a 400 page book from 2500 small XML files with multiple graphics per page, there is a little overhead in loading the files and in constructing the XML tree as well as in inserting the graphics, but in such cases one expects a few seconds longer runtime. The METAFUN manual has some 450 pages with over 2500 runtime-generated METAPOST graphics. It has color, uses quite some fonts, has lots of font switches (verbatim, too), but, still, one run takes only 18 seconds in stock LuaTeX and less than 15 seconds with LuajitTeX. Keep these numbers in mind if a non-ConTeXt user barks against the performance tree that his few-page mediocre document takes 10 seconds to compile: the content, styling, quality of macros and whatever one can come up with all play a role. Personally, I find any rate between 10 and 30 pages per second acceptable, and, if I get the lower rate, then I normally know pretty well that the job is demanding in all kind of aspects.

Over time, the ConTeXt–LuaTeX combination, in spite of the fact that more functionality has been added, has not become slower. In fact, some subsystems have been sped up. For instance, font handling is very sensitive to adding functionality. However, each version so far performed a bit better. Whenever some neat new trickery was added, at the same time improvements were made thanks to more insight in the matter. In practice, we're not talking of changes in speed by large factors but more by small percentages. I'm pretty sure that most ConTeXt users never noticed. Recently, a 15–30% speed up (in font handling) was realized (for more complex fonts), but only when you use such complex fonts and pages full of text will you see a positive impact on the whole run.

There is one important factor I didn't mention yet: the efficiency of the console. You can best check that by making a format (`context --make en`). When that is done by piping the messages to a file, it takes 3.2 seconds on my laptop and about the same when done from the editor (SCITE), maybe because the LuaTeX run and the log pane run on a different thread. When I use the standard console, it takes 3.8 seconds in Windows 10 Creative update (in older versions it took 4.3 seconds and slightly less when using a console wrapper). The

powershell takes 3.2 seconds, which is the same as piping to a file. Interesting is that in Bash on Windows, it takes 2.8 seconds and 2.6 seconds when piped to a file. Normal runs are somewhat slower, but it looks like the 64 bit Linux binary is somewhat faster than the 64 bit mingw version.[1] Anyway, it demonstrates that when someone yells a number, you need to ask what the conditions were.

At a ConTeXt meeting, there has been a presentation about possible speedups of a run by using, for instance, a separate syntax checker to prevent a useless run. However, the use case concerned a document that took a minute on the machine used, while the same document took a few seconds on mine. At the same meeting, we also did a comparison of speed for a LaTeX run using PdfTeX and the same document migrated to ConTeXt MkIV using LuaTeX (Harald Königs xml torture and compatibility test). Contrary to what one might expect, the ConTeXt run was significantly faster; the resulting document was a few gigabytes in size.

Bottlenecks

I will discuss a few potential bottlenecks next. A complex integrated system like ConTeXt has lots of components and some can be quite demanding. However, when something is not used, it has no (or hardly any) impact on performance. Even when we spend a lot of time in Lua, that is not the reason for a slowdown. Sometimes using Lua results in a speedup, sometimes it doesn't matter. Complex mechanisms like natural tables, for instance, will not suddenly become less complex. So, let's focus on the "aspects" that come up in those complaints: fonts and Lua. Because I only use ConTeXt and occasionally test with the plain TeX version that we provide, I will not explore the potential impact of using truckloads of packages, styles, and such, which I'm sure of plays a role, but one neglected in my discussion.

Fonts

According to the principles of LuaTeX, we process (OpenType) fonts using Lua. That way, we have complete control over any aspect of font handling, and can, as expected in TeX systems, provide users with what they need, now and in the future. In fact, if we didn't have that freedom in ConTeXt, I would probably have already quit using TeX a decade ago and found myself some other (programming) niche.

[1] Long ago, we found that LuaTeX is very sensitive to for instance the cpu cache, so maybe there are some differences due to optimization flags and/or the fact that bash runs in one thread, and all file io takes place in the main Windows instance. Who knows.

After a font has been loaded, part of the data gets passed to the TEX engine, so that it can do its work. For instance, in order to be able to typeset a paragraph, TEX needs to know the dimensions of glyphs. Once a font has been loaded (that is, the binary blob) it's fetched from a cache the next time. Initial loading (and preparation) takes some time, depending on the complexity and the size of the font. Loading from cache is close to instantaneous. After loading, the dimensions are passed to TEX, but all data remains accessible for any desired usage. The OpenType feature processor, for instance, uses that data, and ConTEXt, for sure, needs that data (quickly accessible) for different purposes, too.

When a font is used in a so-called base mode, we let TEX do the ligaturing and kerning. This is possible with simple fonts and features. If you have a critical workflow, you might enable base mode, which can be done per font instance. Processing in node mode takes some time, but how much depends on the font and script. Normally, there is no difference between ConTEXt and generic usage. In ConTEXt, we also have dynamic features, and the impact on performance depends on usage. In addition to base and node, we also have plug mode, but that is only used for testing and therefore not advertised.

Every `\hbox` and every paragraph goes through the font handler. Because we support mixed modes, some analysis takes place, and because we do more in ConTEXt, the generic analyzer is more lightweight, which again can mean that a generic run is not slower than a similar ConTEXt one.

Interesting is that added functionality for variable and/or color fonts had no impact on performance. Runtime-added user features can have some impact, but, when defined well, it can be neglected. I bet that when you add additional node list handling yourself, its impact on performance will be larger. But, in the end, what counts is that the job gets done and the more you demand the higher the price you pay.

Lua

The second possible bottleneck when using LuaTEX can be in using Lua code. However, using that is laughable as an argument for slow runs. For instance, ConTEXt MkIV can easily spend half its time in Lua, and that is not making it any slower than MkII using PdfTEX doing equally complex things. For instance, the embedded metapost library makes MkIV way faster than MkII, and the built-in xml processing capabilities in MkIV can easily beat MkII xml handling, apart from the fact that it can do more, like filtering by path and expression. In fact, files that take, say, half a minute in MkIV, could as well have taken 15 minutes or more in MkII (and imagine multiple runs then).

So, for ConTEXt, using Lua to achieve its objectives is mandatory. The combination of TEX, metapost and Lua is pretty powerful! Each of these

components is really fast. If TEX is your bottleneck, review your macros! When LUA seems to be the bad, go over your code and make it better. Much of the LUA code I see flying around doesn't look that efficient, which is okay, because the interpreter is really fast, but don't blame LUA beforehand, blame your coding (style) first. When METAPOST is the bottleneck, well, sometimes not much can be done about it, but when you know that language well enough, you can often make it perform better.

For the record: every additional mechanism that kicks in, like character spacing (the ugly one), case treatments, special word and line trickery, marginal stuff, graphics, line numbering, underlining, referencing, and a few dozen more will add a bit to the processing time. In that case, in CONTEXT, the font related runtime gets pretty well obscured by other things happening, just that you know.

Some timing

Next, I will show some timings related to fonts. For this, I use stock LUATEX (second column) as well as LUAJITTEX (last column), which, of course, performs much better. The timings are rounded to three decimal places, but, as the system load is usually only consistent in a set of test runs, the last two decimals only matter in relative comparison. So, for comparing runs over time, round to the first decimal. Let's start with loading a bodyfont. This happens once per document, and one usually only has one bodyfont active. Loading involves definitions as well as setting up math, so a couple of fonts are actually loaded even if they're not used later on. A setup normally involves a serif, sans, mono and math setup (in CONTEXT).[2]

bodyfont		
modern	0.023 s	0.019 s
pagella	0.127 s	0.079 s
termes	0.128 s	0.087 s
cambria	0.180 s	0.123 s
dejavu	0.140 s	0.092 s
ebgaramond	0.142 s	0.093 s
lucidaot	0.146 s	0.120 s

There is a bit of a difference between the font sets, but a safe average is 150 milliseconds, and this is rather constant over runs.

[2]The timing for Latin Modern is so low, because that font is loaded already.

An actual font switch can result in loading a font, but this is a one-time overhead. Loading four variants (regular, bold, italic and bold italic) roughly takes the following time:

bodyfont switch and 4 style changes (first time)		
modern	0.028 s	0.028 s
pagella	0.035 s	0.031 s
termes	0.036 s	0.069 s
cambria	0.052 s	0.047 s
dejavu	0.091 s	0.069 s
ebgaramond	0.022 s	0.016 s
lucidaot	0.017 s	0.031 s

Using them again later on takes no time:

bodyfont switch and 4 style changes (follow-up)		
modern	0.000 s	0.000 s
pagella	0.001 s	0.000 s
termes	0.000 s	0.001 s
cambria	0.000 s	0.000 s
dejavu	0.001 s	0.000 s
ebgaramond	0.000 s	0.000 s
lucidaot	0.000 s	0.000 s

Before we start timing the font handler, a few baseline benchmarks are shown. When no font is applied and nothing else is done with the node list, we get:

100 hboxes with 4 texts and no font handling		
baseline	0.142 s	2.343 s

A simple monospaced no-features-applied run takes a bit more:

100 hboxes with 4 texts and no features		
baseline	0.275 s	0.220 s

Now, we show a one-font typesetting run. As with the two benchmarks before, we just typeset a text in a `\hbox`, so no par builder interference happens. We use the `sapolsky` sample text and typeset it 100 times 4, first without font switches.

100 hboxes with 4 texts using one font		
modern	0.933 s	0.591 s
pagella	1.027 s	0.660 s
termes	1.032 s	0.604 s
cambria	1.483 s	0.862 s
dejavu	1.009 s	0.581 s
ebgaramond	3.240 s	1.774 s
lucidaot	0.699 s	0.444 s

Much more runtime is needed when we typeset with four font switches. Ebgaramond is the most demanding. Actually, we're not doing 4 fonts there because ebgaramond has no bold, so the numbers are a bit lower than expected for this example. One reason for it being demanding is that it has lots of (contextual) lookups. Combining lookups saves space and time, so complexity of a font is not always a good predictor for performance hits.

100 hbox with 4 texts using 4 font switches		
modern	1.611 s	0.946 s
pagella	1.697 s	0.975 s
termes	1.727 s	1.038 s
cambria	2.815 s	1.626 s
dejavu	1.946 s	1.087 s
ebgaramond	5.445 s	2.899 s
lucidaot	1.288 s	0.746 s

If we typeset paragraphs, we get the following:

100 times 4 texts on pages (Figure 1)		
modern	1.377 s	0.904 s
pagella	1.523 s	0.961 s
termes	1.453 s	0.898 s
cambria	1.901 s	1.138 s
dejavu	1.437 s	0.917 s
ebgaramond	3.714 s	2.133 s
lucidaot	1.117 s	0.767 s

We're talking of some 275 pages here.

100 times 4 texts on pages using 4 styles (Figure 2)		
modern	2.074 s	1.307 s
pagella	2.155 s	1.338 s
termes	2.153 s	1.373 s
cambria	3.349 s	2.012 s
dejavu	2.408 s	1.453 s
ebgaramond	4.368 s	2.512 s
lucidaot	1.682 s	1.056 s

There is, of course, overhead in handling paragraphs and pages:

100 times 4 texts on pages with no features (Figure 3)		
baseline	0.825 s	0.559 s

Before I discuss these numbers in more detail, two more benchmarks are shown. The next table concerns a paragraph with only a few (bold) words.

100 times 1 text on pages with bold font switches (Figure 4)		
modern	0.409 s	0.263 s
pagella	0.445 s	0.281 s
termes	0.432 s	0.300 s
cambria	0.606 s	0.368 s
dejavu	0.465 s	0.295 s
ebgaramond	0.922 s	0.530 s
lucidaot	0.345 s	0.220 s

The next table concerns a paragraph with a few monospaced words using `\type`.

100 times 1 text on pages with word verbatim switches (Figure 5)		
modern	0.380 s	0.255 s
pagella	0.396 s	0.266 s
termes	0.384 s	0.278 s
cambria	0.535 s	0.355 s
dejavu	0.366 s	0.247 s
ebgaramond	0.939 s	0.533 s
lucidaot	0.322 s	0.216 s

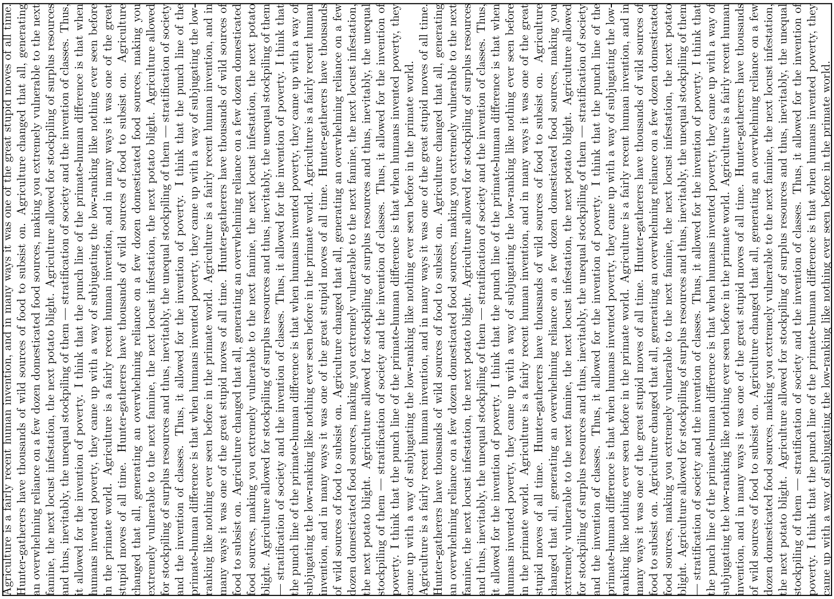


Figure 1 100 times 4 texts on pages

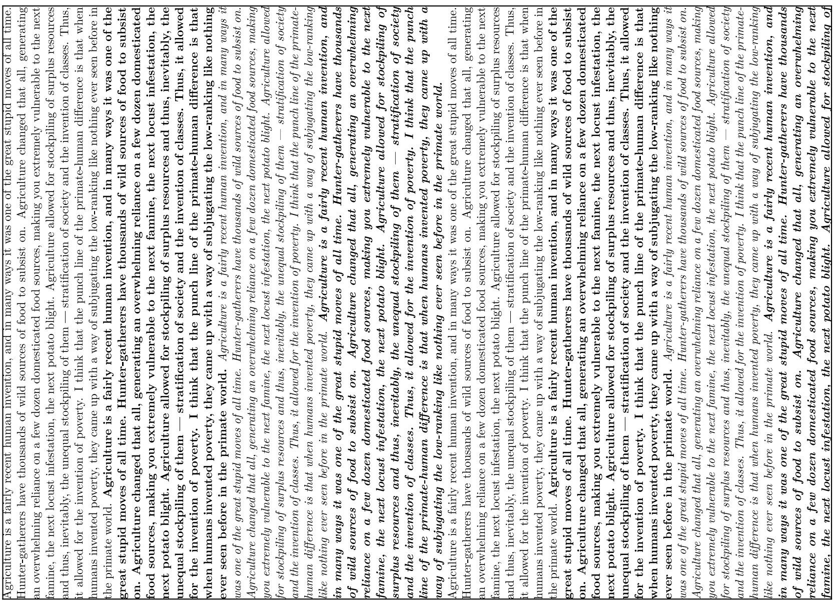


Figure 2 100 times 4 texts on pages using 4 styles

```
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid mov
of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture cha
that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you
tremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agricul
ture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of
--- stratification of society and the invention of classes. Thus, it allowed for the invention of p
I think that the punch line of the primate-human difference is that when humans invented poverty, t
came up with a way of subjugating the low-ranking like nothing ever seen before in the primate worl
```

Figure 3 100 times 4 texts on pages with no features

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

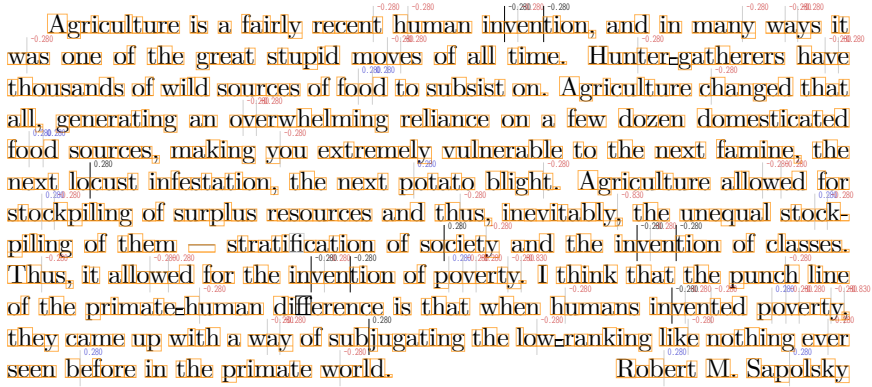
Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, **inevitably**, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like **nothing ever seen before** in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great **stupid** moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating

Figure 4 100 times 1 text on pages with bold switches

Figure 5 100 times 1 text on pages with word verbatim switches

When a node list (hbox or paragraph) is processed, each glyph is looked at. One important property of LuaTeX (compared to PdfTeX) is that it hyphenates the whole text, not only the most feasible spots. For the `sapolsky` snippet, this results in 200 potential breakpoints registered in an equal number of discretionary nodes. The snippet has 688 characters grouped into 125 words and, because it's an English quote, we're not hampered with composed characters or complex script handling. And, when we mention 100 runs, then we actually mean 400 ones when font switching and bodyfonts are compared.



In order to get substitutions and positioning right, we need not only to consult streams of glyphs but also combinations with preceding pre or replace, or trailing post and replace texts. When a font has a bit more complex substitutions, as ebgaramond has, multiple (sometimes hundreds of) passes over the list are made. This is why the more complex a font is, the more runtime is involved.

Another factor, one you could easily deduce from the benchmarks, is intermediate font switches. Even a few such switches (in the last benchmarks) already result in a runtime penalty. The four switch benchmarks show an impressive increase of runtime, but it's good to know that such a situation seldom happens. It's also important not to confuse, for instance, a verbatim snippet with a bold one. The bold one is indeed leading to a pass over the list, but verbatim is normally skipped, because it uses a font that needs no processing. That verbatim or bold have the same penalty is mainly due to the fact that verbatim itself is costly: the text is picked up using a different catcode regime and travels through TEX and LUA before it finally gets typeset. This relates to special treatments of spacing, syntax highlighting, and such.

Also, keep in mind that the page examples are quite unreal. We use a layout with no margins, just text from edge to edge.

So, what is a realistic example? That is hard to say. Unfortunately, no one has ever asked us to typeset novels. They are rather brain dead-products for a machinery, so they process fast. On the mentioned laptop, 350 word pages in Dejavu fonts can be processed at a rate of 75 pages per second with LUATEX and over 100 pages per second with LUAJITTEX. On a more modern laptop or a professional server, the performance is of course better. And, for automated flows, batch mode is your friend. The rate is not much worse for a document in a language with a bit more complex character handling, take accents or ligatures. Of course, PDFTEX is faster on such a dumb document, but kick in some more functionality, and the advantage quickly disappears. So, if someone complains that LUATEX needs 10 or more seconds for a simple few page document ... you can bet that when the fonts are seen as reason, then the setup is pretty bad. Personally I would not waste time on such a complaint.

Valid questions

Here are some reasonable questions that you can ask when someone complains to you about the slowness of LUATEX:

What engines do you compare?
If you come from PDFTEX, you come from an 8-bit world: input and font handling are based on bytes, and hyphenation is integrated into the par builder.

If you use UTF-8 in PdfTeX, the input is decoded by TeX macros, which carries a speed penalty. Because in the wide engines macro names can also be UTF sequences, construction of macro names is less efficient too.

When you try to use wide fonts, there is, again, a penalty. Now, if you use XeTeX or LuaTeX, your input is UTF-8, which becomes something 32-bit internally. Fonts are wide, so more resources are needed, apart from these fonts being larger and in need of more processing due to feature handling. Where XeTeX uses a library, LuaTeX uses its own handler. Does that have a consequence for performance? Yes and no. First of all, it depends on how much time is spent on fonts at all, but even then, the difference is not that large. Sometimes XeTeX wins, sometimes it's LuaTeX. One thing is clear: LuaTeX is more flexible as we can roll out our own solutions and therefore do more advanced font magic. For ConTeXt, it doesn't matter as we use LuaTeX exclusively, and we rely on the flexible font handler, also for future extensions. If really needed, you can kick in a library-based handler but it's (currently) not distributed as we lose other functionality, which would, in turn, result in complaints about that fact (apart from conflicting with the strive for independence).

There is no doubt that PdfTeX is faster, but, for ConTeXt, it's an obsolete engine. The hard-coded-solutions engine XeTeX is not feasible for ConTeXt either. So, in practice, ConTeXt users have no choice: LuaTeX is used, but users of other macro packages can use the alternatives if they are not satisfied with performance. The fact that ConTeXt users don't complain about speed is a clear signal that this is a no-issue. And, if you want more speed, you can always use LuajitTeX.[3] In the last section, the different engines will be compared in more detail.

Just that you know, when we do the four-switches example in plain TeX on my laptop, I get a rate of 40 pages per second, and, for one font, 180 pages per second. There is, of course, a bit more going on in ConTeXt in page building and so, but the difference between plain and ConTeXt is not that large.

What macro package is used?

When plain TeX is used, a follow up question is: what variant? The ConTeXt distribution ships with `luatex-plain`, and that is our benchmark. If there really is a bottleneck, it is worth exploring, but keep in mind that, in order to be plain, not that much can be done. The LuaTeX part is just an example of an implementation. We already discussed ConTeXt, and for LaTeX, I don't want to

[3] In plug mode, we can actually test a library and experiments have shown that performance on the average is much worse, but it can be a bit better for complex scripts, although a gain gets unnoticed in normal documents. So, one can decide to use a library but at the cost of much other functionality that ConTeXt offers, so we don't support it.

speculate where performance hits might come from. When we're talking fonts, ConTeXt can actually be a bit slower than the generic (or LaTeX) variant, because we can kick in more functionality. Also, when you compare macro packages, keep in mind that, when node list processing code is added in that package, the impact depends on interaction with other functionality and depends on the efficiency of the code. You can't compare mechanisms or draw general conclusions when you don't know what else is done!

What do you load?
Most ConTeXt modules are small and load fast. Of course, there can be exceptions when we rely on third party code; for instance, loading tikz takes a bit of time. It makes no sense to look for ways to speed that system up, because it is maintained elsewhere. There can probably be gained a bit, but, again, no user has complained so far.

If ConTeXt is not used, one probably also uses a large TeX installation. File lookup in ConTeXt is done differently, and can be faster. Even loading can be more efficient in ConTeXt, but it's hard to generalize that conclusion. If one complains about loading fonts being an issue, just try to measure how much time is spent on loading other code.

Did you patch macros?
Not everyone is a TeXpert. So, coming up with macros that are expanded many times and/or have inefficient user interfacing, can have some impact. If someone complains about one subsystem being slow, then honesty demands to complain about other subsystems as well. You get what you ask for.

How efficient is the code that you use?
Writing super-efficient code only makes sense when it's used frequently. In ConTeXt, most code is reasonably efficient. It can be that in one document, fonts are responsible for most runtime, but in another document, table construction can be more demanding while yet another document puts some stress on interactive features. When hz or protrusion is enabled, then you run substantially slower anyway, so when you are willing to sacrifice 10% or more of runtime, don't complain about other components. The same is true for enabling SyncTeX: if you are willing to add more than 10% of runtime for that, don't wither about the same amount for font handling.[4]

[4] In ConTeXt, we use a SyncTeX alternative that is somewhat faster, but it remains a fact that enabling more and more functionality will make the penalty of, for instance, font processing relatively small.

How efficient is the styling that you use?
Probably the most easily overlooked optimization is in switching fonts and colors. Although in ConTeXt, font switching is fast, I have no clue about it in other macro packages. But in a style, you can decide to use inefficient (massive) font switches. The effects can easily be tested by commenting out bits and pieces. For instance, sometimes you need to do a full bodyfont switch when changing a style, like assigning `\small\bf` to the `style` key in `\setuphead`, but often using e.g. `\tfd` is much more efficient and works quite as well. Just try it.

Are fonts really the bottleneck?
We already mentioned that one can look in the wrong direction. Maybe, once someone is convinced that fonts are the culprit, it gets hard to look at the real issue. If a similar job in different macro packages has a significantly different runtime, one can wonder what happens indeed.

It is good to keep in mind that the amount of text is often not as large as you think. It's easy to do a test with hundreds of paragraphs of text, but, in practice, we have whitespace, section titles, half empty pages, floats, itemize and similar constructs, etc. Often, we don't mix many fonts in the running text either. So, in the end, a real document is your best test.

If you use Lua, is that code any good?
You can gain from the faster virtual machine of LuajitTeX. Don't expect wonders from the jitting as that only pays off in long runs with the same code used over and over again. If the gain is high, you can even wonder how well-written your Lua code is anyway.

What if they don't believe you?
So, say that someone finds LuaTeX slow, what can be done about it? Just advice them to stick to their previously-used tool. Then, if arguments come that one also wants to use utf-8, OpenType fonts, a bit of metapost, and is looking forward to using Lua runtime, the only answer is: take it or leave it. You pay a price for progress, but, if you do your job well, the price is not that high. Tell them to spend time on learning and maybe adapting and to bark against their own tree before barking against those who took that step a decade ago. Most ConTeXt users took that step and someone still using LuaTeX after a decade can't be that stupid. It's always best to first wonder what one actually asks from LuaTeX, and if the benefit of having Lua on board has an advantage. If not, one can just use another engine.

Also think of this: when a job is slow, for me it's no problem to identify where the problem is. The question then is: can something be done about it?

Well, I happily keep the answer for myself. After all, some people always need room to complain, if only to hide their ignorance or incompetence. Who knows.

Comparing engines

The next comparison is to be taken with a grain of salt and concerns the state of affairs mid-2017. First of all, you cannot really compare MkII with MkIV: the latter has more functionality (or a more advanced implementation of functionality). And, as mentioned, you can also not really compare PdfTeX and the wide engines. Anyway, here are some (useless) tests. First, a bunch of loads. Keep in mind that different engines also deal differently with reading files. For instance, MkIV uses LuaTeX callbacks to normalize the input and has its own readers. There is a bit more overhead in starting up a LuaTeX run, and some functionality is enabled that is not present in MkII. The format is also larger, if only because we preload a lot of useful font, character and script related data.

```
\starttext
    \dorecurse {#1} {
        \input knuth
        \par
    }
\stoptext
```

When looking at the numbers, one should realize that the times include startup and job management by the runner scripts. We also run in batchmode to avoid logging to influence runtime. The average is calculated from 5 runs.

engine	**#1 = 50**	**#1 = 500**	**#1 = 2500**
pdftex	0.43 s	0.77 s	2.33 s
xetex	0.85 s	2.66 s	10.79 s
luatex	0.94 s	2.50 s	9.44 s
luajittex	0.68 s	1.69 s	6.34 s

The second example does a few switches in a paragraph:

```
\starttext
    \dorecurse {#1} {
        \tf \input knuth
        \bf \input knuth
        \it \input knuth
        \bs \input knuth
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.58 s	2.10 s	8.97 s
xetex	1.47 s	8.66 s	42.50 s
luatex	1.59 s	8.26 s	38.11 s
luajittex	1.12 s	5.57 s	25.48 s

The third example does more, resulting in multiple subranges per style:

```
\starttext
    \dorecurse {#1} {
        \tf \input knuth \it knuth
        \bf \input knuth \bs knuth
        \it \input knuth \tf knuth
        \bs \input knuth \bf knuth
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.59 s	2.20 s	9.52 s
xetex	1.49 s	8.88 s	43.85 s
luatex	1.64 s	8.91 s	41.26 s
luajittex	1.15 s	5.91 s	27.15 s

The last example adds some color. Enabling more functionality can have an impact on performance. In fact, as MkIV uses a lot of Lua and is also more advanced that MkII, one can expect a performance hit, but, in practice, the opposite happens, which can also be due to some fundamental differences deep down at the macro level.

```
\setupcolors[state=start] % default in MkIV
\starttext
    \dorecurse {#1} {
        {\red \tf \input knuth \green \it knuth}
        {\red \bf \input knuth \green \bs knuth}
        {\red \it \input knuth \green \tf knuth}
        {\red \bs \input knuth \green \bf knuth}
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.61 s	2.36 s	10.33 s
xetex	1.53 s	9.25 s	45.59 s
luatex	1.65 s	8.91 s	41.32 s
luajittex	1.15 s	5.93 s	27.34 s

In these measurements, the accuracy is a few decimals, but a pattern is visible. As expected, PdfTeX wins on simple documents but starts losing when things get more complex. For these tests, I used 64-bit binaries. A 32-bit XeTeX with MkII performs the same as LuajitTeX with MkIV, but a 64-bit XeTeX is actually quite a bit slower. In that case, the mingw cross-compiled LuaTeX version does pretty well. A 64-bit PdfTeX is also slower (it looks) than a 32-bit version. So, in the end, there are more factors that play a role. Choosing between LuaTeX and LuajitTeX depends on how well the memory-limited LuajitTeX variant can handle your documents and fonts.

Because in most of our recent styles we use OpenType fonts and (structural) features as well as recent metafun extensions only present in MkIV, we cannot compare engines using such documents. The mentioned performance of LuaTeX (or LuajitTeX) and MkIV on the metafun manual illustrate that, in most cases, this combination is a clear winner.

```
\starttext
    \dorecurse {#1} {
        \null \page
    }
\stoptext
```

This gives:

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.46 s	1.05 s	3.72 s
xetex	0.73 s	1.80 s	6.56 s
luatex	0.84 s	1.44 s	4.07 s
luajittex	0.61 s	1.10 s	3.33 s

That leaves the zero run:

```
\starttext
    \dorecurse {#1} {
        % nothing
    }
\stoptext
```

This gives the following numbers. In longer runs, the difference in overhead is negligible.

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.36 s	0.36 s	0.36 s
xetex	0.57 s	0.57 s	0.59 s
luatex	0.74 s	0.74 s	0.74 s
luajittex	0.53 s	0.53 s	0.54 s

It will be clear that when we use different fonts, the numbers will also be different. And, if you use a lot of runtime METAPOST graphics (for instance for backgrounds), the MkIV runs end up at the top. And, when we process XML, it will be clear that going back to MkII is no longer a realistic option. It must be noted that I occasionally manage to improve performance, but we've now reached a state where there is not that much to gain. Some functionality is hard to compare. For instance, in ConTeXt, we don't use much of the PDF backend features because we implement them all in Lua. In fact, even in MkII (already done in TeX), so in the end, the speed difference there is not large and often in favour of MkIV.

For the record, I mention that shipping out the about 1250 pages has some overhead too: about 2 seconds. Here, LuajitTeX is 20% more efficient, which is an indication of quite some Lua involvement. Loading the input files has an overhead of about half a second. Starting up LuaTeX takes more time than PdfTeX and XeTeX, but that disadvantage disappears with more pages. So, in the end, there are quite some factors that blur the measurements. In practice, what matters is convenience: does the runtime feel reasonable and, in most cases, it does.

If I would replace my laptop with a reasonable comparable alternative, then that one would be some 35% faster (single threads on processors don't gain much per year). I guess that this is about the same increase in performance that ConTeXt MkIV got in that period. I don't expect such a gain in the upcoming years, so, at some point, we're stuck with what we have.

Summary

So, how "slow" is LuaTeX really compared to the other engines? If we go back in time to when the first wide engines showed up, Omega was considered to be slow, although I never tested that myself. Then, when XeTeX showed up, there was not much talk about speed, just about the fact that we could use OpenType fonts and native UTF input. If you look at the numbers, for sure you can say that it was much slower than PdfTeX. So, how come that some people complain about LuaTeX being so slow, especially when we take into account that it's not that much slower than XeTeX, and that LuajitTeX is often faster than XeTeX? Also, computers have become faster. With the wide

engines, you get more functionality and that comes at a price. This was accepted for XeTeX and is also acceptable for LuaTeX. But the price is not that high if you take into account that hardware performs better: you just need to compare LuaTeX (and XeTeX) runtime with PdfTeX runtime 15 years ago.

As a comparison, look at games and video. Resolution became much higher as did color depth. Higher frame rates were in demand. Therefore, the hardware had to become faster, and it did, and, as a result, the user experience kept up. No user will say that a modern game is slower than an old one, because the old one does 500 frames per second compared to some 50 for the new game on the modern hardware. In a similar fashion, the demands for typesetting became higher: Unicode, OpenType, graphics, xml, advanced pdf, more complex (niche) typesetting, etc. This happened more or less in parallel with computers becoming more powerful. So, as with games, the user experience didn't degrade with demands. Comparing LuaTeX with PdfTeX is like comparing a low-res, low-framerate, low-color game with a modern one. You need to have up-to-date hardware and even then, the writer of such programs needs to make sure that they run efficiently, simply because hardware no longer scales like it did decades ago. You need to look at the bigger picture.

Rychlost ConTeXtu

Rychlost TeXového stroje je ovlivněna množstvím faktorů, jako je formát, makra, skripty, písma, mikrotypografická rozšíření, SyncTeX a přesměrování standardního chybového výstupu. Článek diskutuje jednotlivé faktory z pohledu uživatele ConTeXtu. Článek dále měří režii formátů ConTeXt MkII a MkIV, dopad přesměrování výstupu na rychlost ConTeXtu MkIV, dopad písem na rychlost sazby v ConTeXtu MkIV a rychlost sazby v ConTeXtu MkII a MkIV.

Klíčová slova: Lua, LuaTeX, LuajitTeX, ConTeXt MkII, ConTeXt MkIV

Hans Hagen, pragma@wxs.nl

Variable fonts

Hans Hagen

In September 2016, variable fonts were added to the OpenType 1.8 specification, reintroducing the ideas of Knuth's METAFONT and Adobe's multiple master to mainstream font design. The article explains the relevant parts of the OpenType specification, and describes the implementation of variable fonts in ConTEXt.

Keywords: variable fonts, OpenType, ConTEXt, LuaTEX

Introduction

History shows the tendency to recycle ideas. Often, quite some effort is made by historians to figure out what really happened, not just long ago, when nothing was written down and we have to do with stories or pictures at most, but also in recent times. Descriptions can be conflicting, puzzling, incomplete, partially lost, biased, ...

Just as language was invented (or evolved) several times, so were scripts. The same might be true for rendering scripts on a medium. Semaphores came and went within decades, and how many people know now that they existed and that encryption was involved? Are the old printing presses truly the old ones, or are older examples simply gone? One of the nice aspects of the internet is that one can now more easily discover similar solutions to the same problem but with a different (and independent) origin.

So, how about this "next big thing" in font technology: variable fonts? In this case, history shows that it's not that new. For most TEX users, the names METAFONT and MetaPost will ring bells. They have a very well-documented history, so there is not much left to speculation. There are articles, books, pictures, examples, sources, and more around for decades. So, the ability to change the appearance of a glyph in a font depending on some parameters is not new. What probably *is* new is that creating variable fonts is done in the natural environment, where fonts are designed: an interactive program. The METAFONT toolkit demands quite some insight into programming shapes in such a way that one can change look and feel depending on parameters. There are not that many metafonts made, and one reason is that making them requires a certain mind- and skill-set. On the other hand, faster computers, interactive programs, evolving web technologies, where real-time rendering and therefore more or less real-time tweaking of fonts is a realistic option, all play a role in acceptance.

But do interactive font design programs make this easier? You still need to translate ideas into usable beautiful fonts. Taking the common shapes of glyphs,

doi: 10.5300/2018-1-4/79

defining extremes and letting a program calculate some interpolations will not always give good results. It's like morphing a picture of your baby's face into yours or that of your grandparent: not all intermediate results will look great. It's good to notice that variable fonts are a revival of existing techniques and ideas used in, for instance, multiple master fonts. The details might matter even more as they can now be exaggerated when transformations are applied.

There is currently (March 2017) not much information about these fonts, so what I say next may be partially wrong or at least different from what is intended. The perspective will be one of a TEX user and coder. Whatever you think of them, these fonts will be out there, and, for sure, there will be nice examples circulating soon. And so, when I ran into a few experimental fonts with PostScript and TrueType outlines, I decided to have a look at what is inside. After all, because it's visual, it's also fun to play with. Let's stress that, at the moment of this writing, I only have a few simple fonts available, fonts that are designed for testing and not for usage. Some recommended tables were missing and no complex OpenType features were used in these fonts.

The specification

I'm not that good at reading specifications, first of all because I quickly fall asleep with such documents, but mostly because I prefer reading other stuff (I do have lots of books waiting to be read). I'm also someone who has to play with something in order to understand it: trial and error is my modus operandi. Eventually, it's my intended usage that drives the interface and that is when everything comes together.

Exploring this technology comes down to: locate a font, get the OpenType 1.8 specification from the Microsoft website, and try to figure out what is in the font. When I had a rough idea, the next step was to get to the shapes and see if I could manipulate them. Of course, it helped that we can already load fonts and play with shapes in ConTEXt using metapost. I didn't have to install and learn other programs. Once I could render them, in this case by creating a virtual font with inline pdf literals, a next step was to apply variation. Then came the first experiments with a possible user interface. Seeing more variation then drove the exploration of additional properties needed for typesetting, like features.

The main extension to the data packaged in a font file concerns the (to be discussed) axis along which variable fonts operate and deltas to be applied to coordinates. The `gdef` table has been extended and contains information that is used in `gpos` features. There are new `hvar`, `vvar` and `mvar` tables that influence the horizontal, vertical, and general font dimensions. The `gvar` table is used for TrueType variants, while the `cff2` table replaces the `cff` table for OpenType

PostScript outlines. The `avar` and `stat` tables contain some metainformation about the axes of variations.

It must be said that, because this is a new technology, the information in the standard is not always easy to understand. The fact that we have two rendering techniques, PostScript `cff` and TrueType `ttf`, also means that we have different information and perspectives. But this situation is not much different from OpenType standards a few years ago: it takes time, but, in the end, I will get there. And, after all, users also complain about the lack of documentation for ConTeXt, so who am I to complain? In fact, it will be those ConTeXt users who will provide feedback and make the implementation better in the end.

Loading

Before we discuss some details, it will be useful to summarize what the font loader does when a user requests a font at a certain size and with specific features enabled. When a font is used for the first time, its binary format is converted into a form that makes it suitable for use in ConTeXt and therefore in LuaTeX. This conversion involves collecting the properties of the font as a whole (official names, general dimensions like x-height and em-width, etc.), of glyphs (dimensions, Unicode properties, optional math properties), and all kinds of information that relate to (contextual) replacements of glyphs (small caps, oldstyle, scripts like Arabic) and positioning (kerning, anchoring marks, etc.). In the ConTeXt font loader, this conversion is done in Lua.

The result is stored in a condensed format in a cache, and, the next time the font is needed, it loads in an instant. In the cached version, the dimensions are untouched, so a font at different sizes has just one copy in the cache. Often, a font is needed at several sizes, and for each size, we create a copy with scaled glyph dimensions. The feature-related dimensions (kerning, anchoring, etc.) are shared and scaled when needed. This happens when sequences of characters in the node list get converted into sequences of glyphs. We could do the same with glyph dimensions, but one reason for having a scaled copy is that this copy can also contain virtual glyphs, and these have to be scaled beforehand. In practice, there are several layers of caching in order to keep the memory footprint within reasonable bounds.[1]

When the font is actually used, interaction between characters is resolved using the feature-related information. When, for instance, two characters need

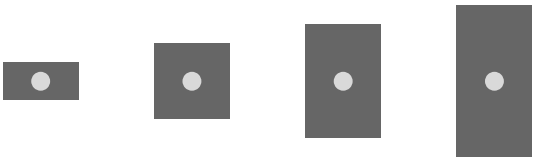
[1] In retrospect, one can wonder if that makes sense; just look at how much memory a browser uses when it has been open for some time. In the beginning of LuaTeX, users wondered about caching fonts, but again, just look at what amounts browsers cache: it gets pretty close to the average amount of writes that a ssd can handle per day within its guarantee.

to be kerned, a lookup results in the injection of a kern, scaled from general dimensions to the current size of the font.

When the outlines of glyphs are needed in METAFUN, the font is also converted from its binary form to something in LUA, but this time we filter the shapes. For a `cff`, this comes down to interpreting the `charstrings` and reducing the complexity to `moveto`, `lineto`, and `curveto` operators. In the process, subroutines are inlined. The result is something that METAPOST is happy with but that also can be turned into a piece of a PDF.

We now come to what a variable font actually is: a basic design which is transformed along one or more axes. A simple example is wider shapes:

We can also go taller and retain the width:



Here we have linear scaling, but glyphs are not normally done that way. There are font collections out there with lots of intermediate variants (say from light to heavy) and it's more profitable to sell each variant independently. However, there is often some logic behind it, probably supported by programs that designers use, so why not build that logic into the font and have one file that represents many intermediate forms. In fact, once we have multiple axes, even when the designer has clear ideas of the intended usage, nothing will prevent users from tinkering with the axis properties in ways that will fulfil their demands (and hurt the designer's eyes). I will not discuss that dilemma here.

When a variable font follows the route described above, we face a problem. When you load a TRUETYPE font, it will just work. The glyphs are packaged in the same format as static fonts. However, a variable font has axes, and, on each axis, a value can be set. Each axis has a minimum, a maximum, and a default. It can be that the default instance also assumes some transformations are applied. The standard recommends adding tables to describe these things, but the fonts that I played with each lacked such tables. So that leaves some guesswork. But still, just loading a TRUETYPE font gives some sort of outcome, although the dimensions (widths) might be weird due to the lack of a (default) axis being applied.

An OPENTYPE font with POSTSCRIPT outlines is different: the internal `cff` format has been upgraded to `cff2`, which on the one hand is less complicated,

but on the other hand has a few new operators, which results in programs that have not been adapted complaining or simply crashing on them.

One could argue that a font is just a resource and that one only has to pass it along, but that's not what works well in practice. Take LuaTEX. We can, of course, load the font and apply axis values, so that we can process the document as we normally do. But, at some point, we have to create a PDF file. We can simply embed the TrueType files, but no axis values are applied. This is because, even if we add the relevant information, there is no way in the current PDF formats to deal with it. For that, we should be able to pass all relevant axis-related information as well as specify what values to use along these axes. And, for TrueType fonts, this information is not part of the shape description so then we need to filter and pass more. An OpenType PostScript font is much cleaner, because there we have the information needed to transform the shape mostly in the glyph description. There, we only need to carry some extra information on how to apply these so-called blend values. The region/axis model used there only demands passing a relatively simple table (stripped down to what we need). But, as said above, `cff2` is not backward-compatible, so a viewer will (currently) simply not show anything.

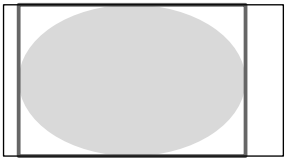
Recalling how we load fonts, how does that change with variable changes? If we have two characters with glyphs that get transformed and that have a kern between them, the kern may or may not transform. So, when we choose values on an axis, then not only glyph properties change but also relations. We can no longer share positional information and scale afterwards, because each instance can have different values to start with. We could carry all that information around and apply it at runtime, but, because we're typesetting documents with a static design, it's more convenient to just apply it once and create an instance. We can use the same caching as mentioned before, but each chosen instance (provided by the font or made up by user specifications) is kept in the cache. As a consequence, using a variable font has no overhead, apart from initial caching.

So, having dealt with that, how do we proceed? Processing a font is not different from what we already had. However, I would not be surprised if users are not always satisfied with, for instance, kerning, because in such fonts, a lot of care has to be given to this by the designer. Of course, I can imagine that programs used to create fonts deal with this, but even then, there is a visual aspect to it, too. The good news is that in ConTEXt we can manipulate features, so, in theory, one can create a so-called font goodie file for a specific instance.

Shapes

For OpenType PostScript shapes, we always have to do a dummy rendering in order to get the right bounding box information. For TrueType, this in-

formation is already present but not when we use a variable instance, so I had to do a bit of coding for that. Here we face a problem. For TEX, we need the width, height and depth of a glyph. Consider the following case:



The shape has a bounding box that fits the shape. However, its left corner is not at the origin. So, when we calculate a tight bounding box, we cannot use it for actually positioning the glyph. We do use it (for horizontal scripts) to get the height and depth, but for the width, we depend on an explicit value. In OpenType PostScript, we have the width available, and how the shape is positioned relative to the origin doesn't much matter. In a TrueType shape, a bounding box is part of the specification, as is the width, but for a variable font, one has to use so-called phantom points to recalculate the width, and the test fonts I had were not suitable for investigating this.

At any rate, once I could generate documents with typeset text using variable fonts, it was time to start thinking about a user interface. A variable font can have predefined instances, but, of course, a user also wants to mess with axis values. Take one of the test fonts: Adobe Variable Font Prototype. It has several instances:

extralight	It looks like this!	`weight=0`	`contrast=0`
light	It looks like this!	`weight=150`	`contrast=0`
regular	It looks like this!	`weight=394`	`contrast=0`
semibold	**It looks like this!**	`weight=600`	`contrast=0`
bold	**It looks like this!**	`weight=824`	`contrast=0`
black high contrast	**It looks like this!**	`weight=1000`	`contrast=100`
black medium contrast	**It looks like this!**	`weight=1000`	`contrast=50`
black	**It looks like this!**	`weight=1000`	`contrast=0`

Such an instance is accessed with:

```
\definefont
  [MyLightFont]
  [name:adobevariablefontprototypelight*default]
```

The Avenir Next variable demo font (currently) provides:

regular	It looks like this!	`weight=400`	`width=100`
medium	It looks like this!	`weight=500`	`width=100`
bold	**It looks like this!**	`weight=700`	`width=100`

heavy	**It looks like this!**	`weight=900`	`width=100`
condensed	It looks like this!	`weight=400`	`width=75`
medium condensed	It looks like this!	`weight=500`	`width=75`
bold condensed	**It looks like this!**	`weight=700`	`width=75`
heavy condensed	**It looks like this!**	`weight=900`	`width=75`

Before we continue, I will show a few examples of variable shapes. Here, we use some METAFUN magic. Just take these definitions for granted.

```
\startMPcode
draw outlinetext.b
 ("\definedfont[name:adobevariablefontprototypeextralight]foo@bar")
 (withcolor "gray")
 (withcolor red withpen pencircle scaled 1/10)
 xsized .45TextWidth ;
\stopMPcode
\startMPcode
draw outlinetext.b
 ("\definedfont[name:adobevariablefontprototypelight]foo@bar")
 (withcolor "gray")
 (withcolor red withpen pencircle scaled 1/10)
 xsized .45TextWidth ;
\stopMPcode
\startMPcode
draw outlinetext.b
 ("\definedfont[name:adobevariablefontprototypebold]foo@bar")
 (withcolor "gray")
 (withcolor red withpen pencircle scaled 1/10)
 xsized .45TextWidth ;
\stopMPcode
\startMPcode
draw outlinetext.b
 ("\definefontfeature[whatever][axis={weight:350}]%
   \definedfont[name:adobevariablefontprototype*whatever]foo@bar")
 (withcolor "gray")
 (withcolor red withpen pencircle scaled 1/10)
 xsized .45TextWidth ;
\stopMPcode
```

The results are shown in Figure 1. What we see here is that as long as we fill the shape everything will look as expected, but only using an outline won't. The crucial (control) points are moved to different locations and, as a result, they can end up inside the shape. Giving up outlines is the price we evidently need

to pay. Of course this is not unique for variable fonts, although, in practice, static fonts behave better. To some extent, we're back to where we were with METAFONT and (for instance) Computer Modern: because these originate in bitmaps (and probably use similar design logic) we also can have overlap and bits and pieces pasted together and no one will notice that. The first outline variants of Computer Modern also had such artifacts, while in the static Latin Modern successors, outlines were cleaned up.

Figure 1 Four variants

The fact that we need to preprocess an instance, but that we only know how to do that after we have retrieved information about axis values from the font means that the font handler has to be adapted to keep caching correct. Another definition is:

```
\definefontfeature
  [lightdefault]
  [default]
  [axis={weight:230,contrast:50}]
\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*lightdefault]
```

Here, the complication is that where normally features are dealt with after loading, the axis feature is part of the preparation (and caching). If you want the virtual font solution, you can do this:

```
\definefontfeature
  [inlinelightdefault]
  [default]
  [axis={weight:230,contrast:50},
   variableshapes=yes]
\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*inlinelightdefault]
```

When playing with these fonts, it was hard to see if loading was done right. For instance, not all values make sense. It is beyond the scope of this article, but axes like weight, width, contrast, and italic values get applied differently to so-called regions (subspaces). So, say that we have an x coordinate with the value 50. This value can be adapted in, for instance, four subspaces (regions), so we actually get $x' = x + s_1 \times x_1 + s_2 \times x_2 + s_3 \times x_3 + s_4 \times x_4$. The (here four) scale factors s_n are determined by the axis value. Each axis has some rules about how to map the values 230 for weight and 50 for contrast to such a factor. Each region has its own translation from axis values to these factors. The deltas $x_1, ..., x_4$ are provided by the font. In a PostScript-based font, we find sequences like:

```
1 <setvstore>
120 [10 -30 40 -60] 1 <blend> ... <operator>
100 120 [10 -30 40 -60] [30 -10 -30 20] 2 <blend> .. <operator>
```

A store refers to a region specification. From there, the factors are calculated using the chosen values on the axis. The deltas are part of the glyphs specification. Officially, there can be multiple region specifications, but how likely it is that they will be used in real fonts is an open question.

In TrueType fonts, the deltas are not in the glyph specification but in a dedicated `gvar` table.

```
apply x deltas [10 -30 40 -60] to x 120
apply y deltas [30 -10 -30 20] to y 100
```

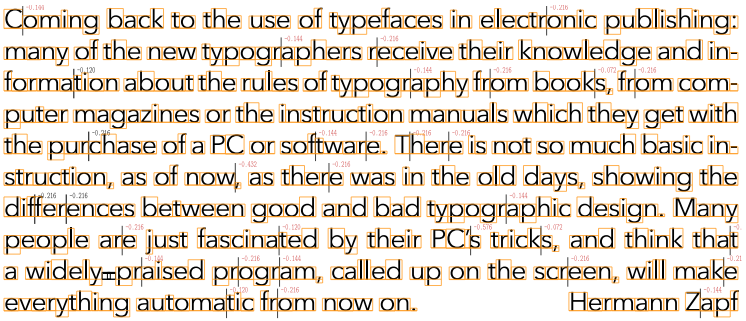
Here, the deltas come from tables outside the glyph specification and their application is triggered by a combination of axis values and regions.

The following two examples use Avenir Next Variable and demonstrate that kerning is adapted to the variant.

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:10}]
\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]
```

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on. Hermann Zapf

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:100}]
\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]
```



Embedding

Once we're done typesetting and a PDF file has to be created, there are three possible routes:

- We can embed the shapes as PDF images (inline literal) using virtual font technology. We cannot use so-called xforms here, because we want to support color selectively in text.
- We can wait till the PDF format supports such fonts, which might happen, but even then we might be stuck for years with viewers getting there. Also, documents need to be printed, and when printer support might arrive is another unknown.
- We can embed a regular font with shapes that match the chosen values on the axis. This solution is way more efficient than the first.

Once I could interpret the right information in the font, the first route was the way to go. A side effect of having a converter for both outline types meant that it was trivial to create a virtual font at runtime. This option will stay in ConTeXt as a pseudo-feature `variableshapes`.

When trying to support variable fonts, I tried to limit the impact on the backend code. Also, processing features and such was not touched. The inclusion of the right shapes is done via a callback that requests the blob to be injected in

the `cff` or `glyf` table. When implementing this, I actually found out that the LuaTeX backend also does some juggling of charstrings to inline subroutines. In retrospect, I could have learned a few tricks faster by looking at that code, but I never realized that it was there. Looking at the code again, it strikes me that the whole inclusion could be done with Lua code, and, some day, I will give that a try.

Conclusion

When I first heard about variable fonts, I was confident that when they showed up, they could be supported. Of course, a specimen was needed to prove this. A first implementation demonstrates that, indeed, it's no big deal to let ConTeXt with LuaTeX handle such fonts. Of course, we need to fill in some gaps, which can be done once we have complete fonts. And then, of course, users will demand more control. In the meantime, the helper script that deals with identifying fonts by name has been extended, and the relevant code has been added to the distribution. At some point, the ConTeXt Garden will provide the LuaTeX binary that has the callback.

I end on a warning note. On the one hand, this technology looks promising, but on the other hand, one can easily get lost. Most such fonts probably operate over a well-defined domain of values, but, even then, one should be aware of complex interactions with features like positioning or replacements. Not all combinations can be tested. It's probably best to stick with fonts that have all the relevant tables and don't depend on properties of a specific rendering technology.

Variabilní fonty

V září roku 2016 přibyly do specifikace OpenType 1.8 tzv. variabilní fonty jakožto znovuoživení myšlenek Knuthova METAFONTu a multiple master fontů od Adobe. Článek vykládá relevantní části specifikace OpenTypea popisuje implementaci variabilních fontů v ConTeXtu.

Klíčová slova: variabilní fonty, OpenType, ConTeXt, LuaTeX

Hans Hagen, pragma@wxs.nl

Mělo by to fungovat VII – Makra

Peter Wilson

Abstrakt

V článku jsou ukázány další možnosti nastavení tvaru a zarovnání odstavce. Dále jsou zde ukázány různé možnosti definování maker v LaTeXu.

Klíčová slova: LaTeX, odstavec, makro, `\parshape`, `\newcommand`, `\def`.

> The raging waves doth belching upwardcast
> The wretched wrackes that round about doe fleete,
> The silken sayles and glistering golden Mast,
> Lies all to torne and trodden under feete.
>
> *The Ship of safegarde*
> Barnabe Googe

Cílem tohoto seriálu je ukázat čtenáři krátké kousky kódu, které mohou vyřešit některé z jeho problémů. Doufám, že situaci ještě více nezkomplikuji v důsledku mých chyb. Opravy, poznámky a návrhy na změny budou vždy vítány.

Hlavním tématem tohoto dílu je definování maker. Otázky tohoto typu, a to zejména v LaTeXu, se ve skupině `comp.text.tex` vyskytují opravdu často.

> The lines are fallen unto me in pleasant places;
> yea I have a goodly heritage.
>
> *Psalm 16, verse 6*

1. Více o odstavcích

Donald Knuth mi poslal následující zdrojový kód se slovy

> „Vytvořil jsem toto makro, které může být užitečné při kontrole parametrů příkazu `\parshape` předtím, než se v odstavci použije skutečný text.“

```
% parshape.tex
%    autor Don Knuth, duben 2007
% \parshapetest{n} nakreslí n řádků horizontálních linek
%    při aktuálně nastavených parametrech odstavce
%    (například \hangindent, \hangafter, \parshape)
```

Z anglického originálu *Glisterings* [9] přeložil Jan Šustek.

 doi: 10.5300/2018-1-4/90

```
\def\parshapetest#1{%
  \leavevmode%% DEK původně použil \indent
  \count255=1 \loop
    \ifnum\count255<#1
    \null\leaders\hrule\hfil\null\break
    \advance\count255 by 1   \repeat
  \null\leaders\hrule\hfil\hskip-\parfillskip
  \null\par}
```

Bohužel už bylo příliš pozdě na to, abych makro využil při přípravě článku [8], který se zabýval sazbou různě tvarovaných odstavců. Byla to škoda i z toho důvodu, že když jsem použil makro `\parshapetest` na některé uvedené příklady, zjistil jsem, že jsem ne zcela pochopil význam některých parametrů odstavce.

Makro `\parshapetest{`⟨*počet*⟩`}` nakreslí ⟨*počet*⟩ horizontálních linek do jednotlivých řádků podle aktuálně nastavených parametrů odstavce. Na první pohled to nezní nijak užitečně. Nicméně makro může ušetřit spoustu času, protože nemusíme vymýšlet vhodný text, na němž by byl dobře vidět výsledný tvar odstavce.

Vyzkoušel jsem například tento příklad z [8].

```
\begingroup
\hangindent=3pc \hangafter=-2
\parshapetest{4}
\endgroup
```

K mému překvapení byl výsledek jiný, než jsem plánoval.

Co jsem si původně neuvědomil, bylo, že i při nastavení `\hangindent` a `\hangafter` se pořád na začátku odstavce použije `\parindent`. Původně plánovaného výsledku lze docílit následujícím nastavením.

```
\begingroup
\parindent=0pt
\hangindent=3pc \hangafter=-2
\parshapetest{4}
\endgroup
```

Výsledek je již správný.

Dále jsem vyzkoušel příklad s makrem `\hangfrom` ze stejného článku, které se používá k sazbě odstavců s více odsazenými řádky. Makro bylo definováno následovně.

```
\newcommand*{\hangfrom}[1]{%
  \setbox\@tempboxa\hbox{{#1}}%
  \hangindent \wd\@tempboxa
  \noindent\box\@tempboxa}
```

Při použití[1]

```
\hangfrom{$\Rightarrow$\space}
\parshapetest{3}
```

dostaneme

$\Rightarrow$

Následuje nastavení zajímavějšího tvaru odstavce a ukázka jeho otestování.

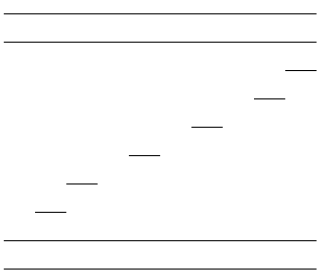
```
\newdimen\delka \delka=\baselineskip
\newcommand*{\zparshape}{%
  \parshape=10 0pt 10\delka % 1
               0pt 10\delka % 2
               9\delka \delka % 3
               8\delka \delka % 4
               6\delka \delka % 5
               4\delka \delka % 6
               2\delka \delka % 7
               \delka \delka % 8
               0pt 10\delka % 9
               0pt 10\delka} % 10
\zparshape
\noindent\parshapetest{10}
```

Výstup je na obrázku 1.

Zkuste si vysázet odstavec s tímto nastavením a textem ze 76 znaků „z“.

```
\zparshape
\noindent
z z z z z z z z z z z z z z z z z z
atd.
```

[1] Všimněte si, že při použití příkazů `\hangindent`, `\hangafter` a `\parshape` není nutné příklady uzavírat mezi `\begingroup` a `\endgroup`. Argumenty těchto příkazů se totiž na konci každého odstavce nulují. (pozn. překl.)



Obrázek 1: Výstup řádků 41–42

Ve skupině `comp.text.tex` položil Stephen Moye dotaz, jak vysázet první řádek odstavce zarovnaný doleva, další řádky zarovnané na střed a poslední řádek zarovnaný doprava. Paul Vojta [7] v odpovědi definoval následující makro, které toto zarovnání nastaví.[2]

```
47 \newcommand*{\leftcenterright}{%
48   \leftskip=0pt plus 1fil
49   \rightskip=0pt plus 1fil
50   \parfillskip=0pt plus -1fil
51   \parindent=0pt
52   \everypar={\hskip0pt plus -1fil\relax}}
```

Jsou dvě možnosti, jak toto makro použít. První z nich je makro použít uvnitř skupiny. Druhou je použít následující makro, které nastaví zarovnání zpět. (Je třeba předem mít nastaven registr `\myparindent` na běžnou hodnotu `\parindent`.)

```
53 \newcommand*{\regularpar}{%
54   \leftskip=0pt
55   \rightskip=\leftskip
56   \parfillskip=0pt plus 1fil
```

[2]Nastavení jednotlivých registrů v makru `\leftcenterright` stojí za vysvětlení.

Na prvním řádku odstavce se vlevo vloží mezery z `\leftskip` a `\parindent` následované obsahem `\everypar`. Celkově tak bude vlevo mezera o velikosti `0pt`. Vpravo bude mezera z `\rightskip`, tj. `0pt plus 1fil`. Nekonečná pružnost mezery vpravo způsobí, že text bude zarovnaný doleva.

Na dalších řádcích odstavce se pouze vloží vlevo mezera z `\leftskip` a vpravo mezera z `\rightskip`. Protože obě mezery jsou stejné a mají nekonečnou pružnost, natáhnou se na stejnou šířku a příslušné řádky budou zarovnané na střed.

Na posledním řádku odstavce se vlevo vloží mezera z `\leftskip`, tj. `0pt plus 1fil`. Vpravo se vloží mezery z `\parfillskip` a `\rightskip`. Celkově tak vpravo bude mezera o velikosti `0pt`. Nekonečná pružnost mezery vlevo způsobí, že text bude zarovnaný doprava.

V původním řešení na řádku 52 chyběl příkaz `\relax`. Cvičením pro pokročilého čtenáře je zjistit, proč je na tomto místě příkaz `\relax` nezbytný. (pozn. překl.)

```
\parindent=\myparindent
\everypar{}}
```

Následuje odstavec vysázený s použitím makra `\leftcenterright`.

```
\leftcenterright
První řádek\break druhý řádek\break
třetí řádek\break poslední řádek\par
\regularpar
```

První řádek

druhý řádek
třetí řádek

poslední řádek

> Who will change old lamps for new?
> ... new lamps for old ones?
> *Arabian Nights: The History of Aladdin*

2. Definiční triumvirát v LaTeXu

LaTeX nabízí pro definování nových maker své makro `\newcommand`, které má trochu jednodušší syntaxi než příkaz TeXu `\def`, na němž je založeno. Syntaxe je

```
\newcommand{⟨název⟩}[⟨počet⟩][⟨arg1⟩]{⟨tělo⟩}
```

kde ⟨*název*⟩ je název definovaného makra včetně zpětného lomítka (například `\makro`) a ⟨*tělo*⟩ je tělo definice makra. To může být jednoduchý text, který se má vysázet, ale může to být také něco velmi složitého. Volitelný parametr ⟨*počet*⟩ určuje, kolik parametrů bude mít nové makro. Pokud se ⟨*počet*⟩ použije, musí mít hodnotu mezi 1 a 9. Pokud je použito ⟨*arg1*⟩, bude první parametr nového makra volitelný a pokud uživatel tento první parametr nepoužije, nastaví se ⟨*arg1*⟩ jako jeho hodnota. Makro definované pomocí `\newcommand` je, řečeno terminologií TeXu, makro typu `\long`, což znamená, že jeho argument může být delší než jeden odstavec nebo, což je ekvivalentní, může obsahovat příkaz `\par`. Varianta s hvězdičkou (`\newcommand*`) nadefinuje makro, které není typu `\long` a jehož argument nesmí obsahovat konec odstavce. Pokud makro ⟨*název*⟩ již dříve bylo definováno, ohlásí LaTeX chybu.

LaTeXové makro

```
\renewcommand{⟨název⟩}[⟨počet⟩][⟨arg1⟩]{⟨tělo⟩}
```

a jeho varianta `\renewcommand*` se chovají obdobně. Jediný rozdíl je, že makro ⟨*název*⟩ musí již dříve být definováno a nyní se předefinuje. Pokud dříve definováno nebylo, ohlásí LaTeX chybu.

Třetí LaTeXové makro pro definování maker je

```
\providecommand{⟨název⟩}[⟨počet⟩][⟨arg1⟩]{⟨tělo⟩}
```

a jeho varianta `\providecommand*`. Pokud makro ⟨*název*⟩ nebylo dříve definováno, chová se `\providecommand` stejně jako `\newcommand`. Pokud makro ⟨*název*⟩ bylo definováno, pak `\providecommand` neudělá nic a neohlásí chybu.

Někdy potřebujete nové makro nadefinovat nezávisle na tom, zda již předtím bylo či nebylo definováno. V tom případě je možné v LaTeXu použít

```
% zajistíme, že makro ⟨název⟩ je definováno
\providecommand{⟨název⟩}{}
% změníme definici
\renewcommand{⟨název⟩}[⟨počet⟩][⟨arg1⟩]{⟨tělo⟩}
```

Pokud se v těle definice vyskytují argumenty, pak se první z nich označuje `#1`, druhý `#2` až devátý `#9`. Argumenty mohou být použity v libovolném pořadí a mohou se libovolně opakovat.

Čas od času se ve skupině `comp.text.tex` objeví otázka, jak definovat makro s více než devíti argumenty. Řešením je rozdělit makro na dvě nebo více maker a každé z nich načte pouze část argumentů. Řekněme, že chceme makro s jedenácti argumenty. Pak použijeme

```
\newcommand{\jedenact}[9]{%
  % načteme 9 argumentů jako #1 až #9
  \zbytek}
\newcommand{\zbytek}[2]{%
  % desátý argument načteme jako #1
  % jedenáctý argument načteme jako #2
  }
```

Uživatel zavolá makro `\jedenact` zdánlivě s jedenácti argumenty. Přitom ve skutečnosti makro `\jedenact` načte pouze prvních devět argumentů a poté zavolá makro `\zbytek`, které zpracuje zbývající dva argumenty.

Pokud je třeba uvnitř makra `\zbytek` zpracovat například čtvrtý argument, lze jej makru `\zbytek` přenést následovně.

```
\newcommand{\jedenact}[9]{%
  % načteme 9 argumentů jako #1 až #9
  \zbytek{#4}}
\newcommand{\zbytek}[3]{%
  % čtvrtý argument načteme jako #1
  % desátý argument načteme jako #2
  % jedenáctý argument načteme jako #3
  }
```

Pro přenesení argumentu makru `\zbytek` lze také využít příkaz `\def` popsaný v následující sekci.

```
\newcommand{\jedenact}[9]{%
  % načteme 9 argumentů jako #1 až #9
  \def\argIV{#4}%
  \zbytek}
\newcommand{\zbytek}[2]{%
  % čtvrtý argument je uložen v makru \argIV
  % desátý argument načteme jako #1
  % jedenáctý argument načteme jako #2
  }
```

Výše uvedeným způsobem lze samozřejmě načíst libovolný počet argumentů. Při větším počtu argumentů se však tento postup stává nepřehledným. Úplně jiný přístup pak nabízí balíček keyval [3] nebo jeho pozdější rozšíření xkeyval [2]. Pomocí těchto balíčků je možné si jednotlivé argumenty makra pojmenovat a uživatel pak může makro zavolat s libovolným počtem těchto argumentů, například jen s některými z nich.

> He who can properly define and divide is to be considered a god.
>
> *Novum Organum*
> Francis Bacon quoting Plato

3. Diktátor v TEXu

TEX disponuje velmi obecným příkazem `\def` pro definování nových maker. Ukázat všechny jeho možnosti v tomto krátkém článku není možné. Knuthův TEXbook [5, kap. 20] nabízí úplný popis, užitečné však mohou být i knihy Eijkhouta [4, kap. 11] nebo Abrahamse a dalších [1, kap. 4 a 9], které jsou pro začínající uživatele lépe čitelné.

Syntaxe příkazu `\def` je úplně jiná, než na co jsou LATEXoví uživatelé zvyklí.

`\def`⟨*název*⟩⟨*parametry*⟩`{`⟨*tělo*⟩`}`

Stejně jako v případě LATEXu je ⟨*název*⟩ název definovaného makra včetně zpětného lomítka (například `\makro`) a ⟨*tělo*⟩ je tělo definice makra. Povšimněte si, že kolem ⟨*název*⟩ se nepoužívají složené závorky.

Část ⟨*parametry*⟩ popisuje výskyt parametrů makra ⟨*název*⟩. Zde se parametry označují `#1`, `#2` atd., musejí být číslovány ve vzestupném pořadí a navíc je rozdíl, zda mezi parametry použijeme mezeru, nebo ne. Pro srovnání uvádíme dva ekvivalentní řádky kódu.

```
\newcommand*{\makro}[2]{...} % LaTeX
\def\makro#1#2{...}          % TeX
... \makro{něco}{bla} ...    % (La)TeX
```

Pokud chceme, aby argument mohl obsahovat konec odstavce, pak musíme makro definovat jako makro typu `\long`. Navíc TEX nijak uživatele neinformuje, že příslušné makro již bylo definováno. Stará definice se jednoduše nahradí novou. Na toto je třeba dát si pozor, protože se může stát, že předefinujete nějaké důležité makro, o kterém jste ani nevěděli, že existuje. Opět uvádíme dva ekvivalentní řádky kódu.

```
\renewcommand{\makro}[2]{...}      % LaTeX
\long\def\makro#1#2{...}           % TeX
... \makro{Odstavec\par}{text} ... % (La)TeX
```

Pokud se v ⟨*parametry*⟩ vyskytují pouze parametry (tj. `#1` atd.), nazývají se tyto parametry neseparované. Odpovídají přesně LATEXovým povinným parametrům. Na druhou stranu, pokud se za `#`n vyskytuje něco jiného než další parametr nebo složená závorka zahajující ⟨*tělo*⟩, pak se parametr `#`n označuje jako separovaný. Příslušné další znaky (přesněji tokeny) se nazývají separátor. Při zavolání makra se příslušný argument načítá tak dlouho, než TEX narazí na tento separátor. Mechanismus volitelných parametrů v LATEXu interně používá právě separované parametry.

Předpokládejme, že chceme nadefinovat makro `\vlak`, které se bude volat

```
\vlak číslo(h:m)
```

kde `číslo`, `h` a `m` jsou argumenty makra `\vlak`. Výše uvedené LATEXové nástroje takové makro nadefinovat neumožňují. Na druhou stranu, příkaz `\def` ano. Pokud nadefinujeme

```
\def\vlak#1(#2:#3){Vlak~#1 jede v~#2:#3.}
```

a zavoláme

```
\textit{\vlak EC130(8:41)}
```

dostaneme výsledek *Vlak EC130 jede v 8:41.*

Může se stát, že budete potřebovat nadefinovat makro, které má dvě varianty, podobně jako makro nadefinované pomocí `\newcommand`. K tomu může pomoci makro `\@ifnextchar` nadefinované v jádru `\LaTeX`u. Syntaxi má následující

```
\@ifnextchar⟨znak⟩{⟨ano⟩}{⟨ne⟩}
```

Makro se podívá na následující znak na vstupu, který je různý od mezery. Pokud je tento znak shodný s ⟨*znak*⟩, pak se provede ⟨*ano*⟩, v opačném případě se provede ⟨*ne*⟩. Jádro LATEXu nabízí také makro

```
\@ifstar{⟨ano⟩}{⟨ne⟩}
```

které testuje, jestli následující znak je hvězdička. Pokud je, pak hvězdičku odstraní a provede ⟨*ano*⟩, v opačném případě provede ⟨*ne*⟩. Toto makro je definováno následovně.

```
\long\def\@firstoftwo#1#2{#1}
\def\@ifstar#1{%
  \@ifnextchar *{\@firstoftwo{#1}}}
```

Nyní můžete snadno definovat makra ve variantě s hvězdičkou i bez hvězdičky.

```
\makeatletter  % mimo soubory .cls a .sty
\def\hvezda{%
  \@ifstar{\@hvezdaS}{\@hvezdaBez}}
  % definice varianty s hvězdičkou
\def\@hvezdaS#1#2{S~hvězdičkou (#1) a (#2).}

  % definice varianty bez hvězdičky
\def\@hvezdaBez#1#2{Bez hvězdičky (#1) a (#2).}
\makeatother   % mimo soubory .cls a .sty
```

Naše makro bude mít dvě varianty a každá bude mít dva argumenty. Vyzkoušejme

```
\hvezda*{první}{druhý}
\hvezda{první}{druhý}
```

Dostaneme výsledek
S hvězdičkou (první) a (druhý).
Bez hvězdičky (první) a (druhý).

Pokud namísto hvězdičky chceme použít jiný znak, například otazník, můžeme makro definovat následovně.

```
\makeatletter  % mimo soubory .cls a .sty
\def\otaznik{%
  \@ifnextchar ?{\@otaznikS}{\@otaznikBez}}
  % definice varianty s otazníkem
\def\@otaznikS#1#2#3{S~otazníkem (#2) a (#3).}
  % definice varianty bez otazníku
\def\@otaznikBez#1#2{Bez otazníku (#1) a (#2).}
\makeatother   % mimo soubory .cls a .sty
```

V předchozím příkladě makro `\@ifstar` hvězdičku odstranilo. Tady musíme otazník odstranit sami. To zařídí naše makro `\@otaznikS` tak, že jej načte jako argument `#1`. Vyzkoušejme

```
\otaznik?{první}{druhý}
\otaznik{první}{druhý}
```

Dostaneme výsledek
S otazníkem (první) a (druhý).
Bez otazníku (první) a (druhý).

Nyní si ukážeme způsob, jak lze v LaTeXu nadefinovat makro, které bude mít dva volitelné parametry a jeden povinný parametr. Heiko Oberdiek k tomuto účelu vytvořil balíček twoopt [6]. My si ukážeme jednodušší způsob, který se při těchto nezvyklých situacích může hodit. Výsledkem bude makro `\dvavolitelne`, jehož volitelné parametry budou mít implicitní hodnotu `jedna` a `dvě`.[3]

```
\def\dvavolitelne{%
  \@ifnextchar [{\@dvavol}{\@dvavol[jedna]}}
\def\@dvavol[#1]{%
  \@ifnextchar [%
      {\@@dvavol{#1}}{\@@dvavol{#1}[dvě]}}
\def\@@dvavol#1[#2]#3{1 (#1) 2 (#2) 3 (#3)}
```

Připomínám, že tato makra je třeba definovat v místě, kde LaTeX považuje znak `@` jako písmeno, například uvnitř balíčku nebo po nastavení `\makeatletter`. Po použití

```
\dvavolitelne{ahoj}
\dvavolitelne[hello]{ahoj}
\dvavolitelne[hello][ahoj]{baf}
```

dostaneme
1 (jedna) 2 (dvě) 3 (ahoj)
1 (hello) 2 (dvě) 3 (ahoj)
1 (hello) 2 (ahoj) 3 (baf)

Seznam literatury

[1] Abrahams, Paul W., Berry, Karl, Hargreaves, Kathryn A. *TeX for the Impatient*. Addison-Wesley, 1990.
[2] Adriaens, Hendri. *The xkeyval package*. 2005. Dostupné na CTAN v adresáři `latex/macros/contrib/xkeyval`.
[3] Carlisle, David. *The keyval package*. 1999. Dostupné na CTAN v adresáři `latex/macros/required/graphics`.
[4] Eijkhout, Victor. *TeX by topic, A TeXnician's Reference*. Addison-Wesley, 1991. Dostupné na `www.wijkhout.net/tbt`.
[5] Knuth, Donald E. *The TeXbook*. Addison-Wesley, 1984.

[3] Doporučuji čtenáři, aby si sám na papíru vyzkoušel postupnou expanzi makra ve všech třech případech. (pozn. překl.)

[6] Oberdiek, Heiko. *The twoopt package: Definitions with two optional arguments.* 1999. Dostupné na CTAN v adresáři `latex/macros/contrib/oberdiek`.
[7] Vojta, Paul. *Re: New York Times headline style.* Příspěvek ve skupině `comp.text.tex`, 10. 7. 2007.
[8] Wilson, Peter. Glisterings. *TUGboat*, 28(2):229–232, 2007.
[9] Wilson, Peter. Glisterings. *TUGboat*, 29(2):324–327, 2008.

Summary

This paper demonstrates more possibilities of setting the paragraph shape and alignment in LaTeX. It also shows several possibilities to define macros in LaTeX.

Key words: LaTeX, paragraph, macro, `\parshape`, `\newcommand`, `\def`.

Peter Wilson, herries.press@earthlink.net
18912 8th Ave. SW
Normandy Park, WA 98166 USA

Zpravodaj Československého sdružení uživatelů TEXu
ISSN 1211-6661 (tištěná verze), ISSN 1213-8185 (online verze)

Vydalo:	Československé sdružení uživatelů TEXu vlastním nákladem jako interní publikaci
Obálka:	Antonín Strejc
Ilustrace na obálce:	Hans Hagen
Počet výtisků:	310
Uzávěrka:	14. 12. 2018
Odpovědný redaktor:	Jan Šustek
Redakční rada:	Pavel Haluza, Lukáš Novotný, Vít Novotný, Michal Růžička a Jan Šustek (šéfredaktor)
Vědecká rada:	Ján Buša (předseda), Jiří Demel, Jaromír Kuben (zástupce předsedy), Jiří Rybička a Petr Sojka
Technická redakce:	Vít Novotný
Evidenční číslo MK:	E 7629
Tisk:	ASMETI, Klášterní 1187, 735 11 Orlová
Adresa:	CSTUG, Nejedlého 373/1, 638 00 Brno
Email:	cstug@cstug.cz

Zřízené poštovní aliasy sdružení CSTUG:

`bulletin@cstug.cz`, `zpravodaj@cstug.cz`
korespondence ohledně Zpravodaje sdružení

`board@cstug.cz`
korespondence členům výboru

`cstug@cstug.cz`, `president@cstug.cz`
korespondence předsedovi sdružení

`gacstug@cstug.cz`
grantová agentura CSTUGu

`secretary@cstug.cz`, `orders@cstug.cz`
korespondence administrativní síle sdružení, objednávky CD a DVD

`cstug-members@cstug.cz`
korespondence členům sdružení

`cstug-faq@cstug.cz`
řešené otázky s odpověďmi navrhované k zařazení do dokumentu CSFAQ

`bookorders@cstug.cz`
objednávky tištěné TEXové literatury na dobírku

ftp server sdružení:
`ftp://ftp.cstug.cz`

www server sdružení:
`https://www.cstug.cz`

CONTENTS