Pablo Rodríguez

From pandoc to ConTEXt

A Method to Generate High–Quality PDF Documents

http://www.from-pandoc-to-context.tk

From pandoc to ConTEXt

A Method to Generate High–Quality PDF Documents

Pablo Rodríguez

2015

http://www.from-pandoc-to-context.tk

Dedicated to Hans Hagen
and John MacFarlarne.

This tiny contribution is a
small sign of my deep gratitude
for their huge achievements.

"If I have seen further it is by standing on the shoulders of giants."
(Isaac Newton, *Letter to R. Hooke*)

Contents

Foreword

Well, it seems that I haven't seen that far. But I can't doubt that I stand on the shoulders of giants. Without their extraordinary achievements, I could never have thought of using their software for my own needs.

These pages describe a method to typeset XHTML generated by `pandoc`. ConTEXt is the right tool, since it deals with XML natively.

`pandoc` generates a much more reliable XHTML output than its ConTEXt counterpart. This is also due to the fact that its light–weight markup format—*Markdown*—was designed with HTML in mind. *Markdown* was conceived as HTML without the tags.

The main advantage of using ConTEXt with `pandoc` is loosing the least part—or really nothing—in translation. Neither code snippets only aimed at ConTEXt nor tricky `pandoc` or ConTEXt invocations are neeeded. Not even special templates are required.

I hope this may be useful to you. I have already written a book with ePub and high–quality PDF outputs. This method works. And even if it needs improvements—at least, some polishing—, it is a way to use `pandoc` natively for PDF documents.

Before you start reading, one last warning. My background is in humanities, not in science. Besides computers, I'm not a technical person. This also means that my writings don't include math.

Last and I hope least, excuse my broken English. I probably have a mixture of British English—which I should have learnt in school—and U.S. English—which I should have learnt in life.

February 22nd, 2015

Introduction

First of all, I must warn you about the fact that I'm not a programer. I'm only an average computer user. So, if I was able to do this, probably anybody can do it. I'm not especially gifted with computers. I was only interested in finding a better method for digital editions.

I had been using LaTeX for almost a decade. I knew that it could generate high–quality PDF documents. It can do a great job, although the main shortcoming is the input format. From LaTeX sources you may expect PDF documents.[1]

After that decade, I started learning ConTeXt. I was curious about its features to typeset critical editions. And I wanted to learn how to typeset XML natively. Although I may have been using it over five years, I'm still a newbie in ConTeXt.

I started using `pandoc` to translate LaTeX sources into HTML files. The results weren't suitable for what I intended. But I realized that the best way to use `pandoc` was to use it natively. The native format in `pandoc`—besides its internal format—is *extended Markdown*.

`pandoc` may fullfill the ancient promise that reads:

> One source to generate them all.

Generating multiple formats from a single source imposes two requirements:

- A consistent input format to read from, such as *extended Markdown*—for the rest of the document simply named as *Markdown*.

- Well–implemented conversion formats to write to. In some cases—where pandoc can't handle the output format natively—, the right tool is needed to generate the final output in that format.

pandoc is able to write to many different formats. The most relevant output formats for digital editions are:

- XHTML, which includes ePub[2] (both versions 2 and 3).

- Source formats to generate PDF documents with further processing: LaTeX, ConTeXt or *Adobe InDesign*.

- RTF, *Office Open XML Document* and *OpenDocument Text*. The last two formats are the native formats of *Microsoft Word* and *LibreOffice Writer*, respectively.

  I don't think word processing formats are relevant for electronic editing, but many people find them useful. I won't discuss them here.

The previous list is by no means complete. It intends to group final output formats, which don't need futher processing to be experienced by the end–user.

pandoc doesn't generate PDF documents natively. It can generate them directly, if you have a TeX distribution installed on your computer. But this isn't native generation: pandoc generates a LaTeX source document and it commands the compilation of this source file.

Unlike the other two groups, PDF is the most cumbersome option to adapt to specific layouts with pandoc. Of course, one can always generate LaTeX or ConTeXt sources and tweak them. But then you have two sources—*Markdown* and TeX—instead of one.

Having two sources is a suboptimal solution, at best. Even if one is the main source and the other one is only for PDF generation. This scenario might work if you modify only the subordinated source—the one for PDF generation. If you have to modify the main source and keep the subsidiary source in sync, you are experiencing the pain of having to maintain more than one source from a given text.

One of the most striking features of free or open source software is that real–world achievements may remain untold. I'm not complaining about the lack of manuals. This is something different. General instructions have to differ from implementations. Manuals may contain code samples, but they aren't intended to explain complete user cases. People can learn a lot from existing implementations. And they come to understand when they start adapting other projects to their own needs.

These pages try to serve as a complete example to go from `pandoc` to ConTEXt. This is a complete document that can be tweaked and learnt from. It may be useful to others, because it was a way of solving issues to me, the original author.

Comments

If you find errors in this document or you want to comment anything related to the topic it handles, open an issue at `https://github.com/ousia/from-pandoc-to-context/issues/new`.

Typographic Conventions

These pages are typeset using both a Roman font family and a typewriter typeface.

The selected Roman font family is *TEX Gyre Pagella*. This typeface is completed by *GFS Didot* for the Greek glyphs.

Italics are used to refer to intellectual works and registered marks. These include corporation and device names, document titles and some computer programs. Italics also emphasize words or expressions.

The *Cousine* monospaced typeface is used for everything that has to be typewritten by the user. This includes computer commands, program options and source code in many different formats. It has support for both Latin and Greek glyph ranges.

As already shown, single programs—not as part of a command invocation—are written with a monospaced font. These are only programs that can only be executed by a typewritten command. All their interactivity is contained in the command line. Those programs have no graphical user interface. `pandoc` is the most obvious example.

Typewritten inline words will be hyphenated using an underscore instead of the standard hyphen. Underscore hyphenation aims to avoid confusion. Because `pandoc` and `pan-doc` would be two different computer programs.[3]

ConTEXt and related programs have special logotypes. This is the reason why they aren't typeset either in italics or monospaced. As part of a execution command, they are displayed as typewritten.

Acknowledgements

John MacFarlarne is the leading developer of `pandoc`. I want to thank him for this excellent software. He also coordinates all efforts from other programers and users to improve `pandoc`. I would like to thank him and Matthew Pickering also for their help programing filters.

I could have never thought of going from `pandoc` to ConTEXt without the extraordinary work of Hans Hagen. He develops this extraordinary tool for digital typesetting called ConTEXt. And he has patiently replied

all my questions about how to parse XML sources with ConTEXt. I thank him for both.

Without the help from Wolfgang Schuster, I wouldn't be using ConTEXt right now. His replies to the many questions I posted on the ConTEXt mailing list are both highly instructive and straight to the point. I want to thank him very much for his readiness to help any user in the ConTEXt mailing list.

1 *Why ConTEXt?*

If you want high–quality PDF documents from `pandoc`, you have these options:

- `groff`.
- LaTeX.
- *Adobe InDesign*.
- ConTeXt.

As explained before, `pandoc` needs an intermediate file to be processed by an external tool that generates the final PDF document. So, let's discuss the options.

I have never used `groff`. It might be similar to TeX, so with similar features and results. Reading its manual,[4] it seems that `groff` has a markup of its own.

LaTeX is much more widely used than `groff`. It is the most popular TeX macropackage. It reads its own markup as input format. Its modular nature may be a strong advantage. But this also poses a problem for the user: different packages might clash. Besides, the user needs an encyclopaedic knowledge of all packages (s)he might benefit from. This also has a consequence: installing a complete LaTeX distribution requires at least a couple of free gigabytes in your storage device.

I have never used *Adobe InDesign*. But I can see some problems deploying it with `pandoc`. First of all, it is a visual program. Its approach

to text editing is completely different from the one pandoc has. I firmly believe that the logical approach is much better for digital editions. At least, in the long run. Because, you need to change your mind first. You have to find the way in which you master and command computers best.

Adobe InDesign is also proprietary. This is the way it is—although I tend to think this is suboptimal. Licenses cost money. And it seems *Adobe* has changed its sales strategy. *InDesign* licenses are no more sold,[5] but rented. The licensed user gets the latest version available. But (s)he has to pay a monthly fee. In the EU, *InDesign* seems to be licensed for 20€/month. The main consequence is not the price, but the perpetual fee. Otherwise, the user won't be able to do what (s)he has been doing. They payment may buy new features, but it does purchase the right to use the program.

ConTEXt has many advantages. It is licensed under the GNU General Public License version 2, such as pandoc. It handles pure text files. Although it has some modules, it doesn't have an huge number of packages, such as LATEX. The *ConTEXt Suite* fits in less than 300MB, including cache files. After using both macropackages, I think that ConTEXt documents are clearer to write than those in LATEX. Don't forget it: this is my personal opinion. And even it has its own markup, it can handle XML natively. This is the key feature to use it with pandoc.

After repeating "native XML handling" many times, I have to describe what this feature promises. Plainly speaking, handling XML natively means no extra layer. Only a configuration file is needed to parse the XML code. This file both maps and formats XML elements and attributes to ConTEXt code. This configuration file is called "environment".

ConTEXt needs an environment to compile a PDF file from XML code, in a similar way a web browser uses a cascading style sheet to display XHTML code. The difference is the following: the environment file is needed to parse the code, but the cascading style sheet is used only to display the code with other layout than the default. A web browser displays XHTML code without cascading style sheets. ConTEXt is not able to compile any XML code without its proper environment.

The main purpose of this document is to provide the user with an environment to start with. This environment may serve as a template to tinker with ConTEXt and to adapt it to the particular needs. At least, this was the way I learn(t) ConTEXt: change things, compile the document and see what happens. As long as you play with duplicated files—and I warmly recommend learning `git`—, this is totally safe.

This approach has a drawback: you are new to ConTEXt and learning is required prior to usage. Don't worry, everyone has been new to ConTEXt once. Seriously, it isn't a bug, it's a feature. Learning is also a requirement with `groff`, *Adobe InDesign* or LATEX. And I don't think ConTEXt is harder to learn than the other three opitions.

I must admit I can't spare anyone the learning process. But this will be a much faster learning. Because you don't need to write full ConTEXt documents. Environments are all that you need. The only requirement is the ability to translate XML elements and attributes to its ConTEXt values. Although I'm still experimienting with environments, I think this is faster to learn than going the full path of ConTEXt.

pandoc

2 *Basic Markup*

In this part, I don't plan to explain how to use `pandoc`. There are awesome guides out there. I will only explain what I consider essential features in markup for text processing.[6] These are logical requirements to digitally edit texts.

Text markup requires two basic components: elements and attributes. All text processing needs at least elements. Every text has elements. Even if markup is already embedded in the text itself: such as spaces and newline characters. Two formatting features such as hyphenation or margins for paragraphs only work if words and paragraphs are present in the text to be processed.

Headings, footnotes, lists and block quotes are such a kind of elements that require markup to exist in a text. `pandoc` has an internal document model inspired in the HTML document model. Unless you compose extremely complex textual structures, `pandoc` provides enough elements for the vast majority of documents you may need.

Attributes extend elements, enabling them to be designed individually or in groups. Without attributes, an element would have the same layout in all its occurrences. In some cases, it is important to be able refer individually to an individual occurrence of an element. For example, the title of the first chapter could be the destination of a reference linked from the introduction. In other cases, some occurrences of a particular element—but not all of them—should have a different layout than the rest of the occurrences from the same element. For example, a first–level title requires a different layout when used as part or chapter heading than when it is used as book title in the cover page.

Attributes are an essential feature when dealing with digital texts. pandoc enables them in some of its elements, but unfortunately not in all of them. The elements granted attributes in pandoc—this is part of its internal document model—are: titles, code blocks and code snippets. As far as I can remember, these are all text elements that allow attributes.

The fact that only a tiny proportion of elements can enjoy attributes in pandoc imposes also a burden on users. Since generic block division and inline span elements are available, there is a workaround to virtually provide attributes for all elements. Wrapping any element into a division or a span—depending whether it is a block or an inline element—with attributes is very close to assign these attributes to the elements themselves. But this is a workaround: it needs extra tagging and more complex formatting.

I can't code, so I don't know how much effort would be involved in enabling attributes in for elements in pandoc's internal document model. I think this is the right thing to do. Mainly because speedy typing—this is the main reason behind light–weight markup languages—and easier layout formatting. And because attributes are emulated with the vast majority of elements, one may end up in a situation where attributes can't be emulated for a particular element.

3 Markdown

Markdown is a light–weight markup language. In its extended version, *Markdown* is the closest version to the native format in `pandoc`. It can't be its internal document format, but it is the most faithful format read from and written to.

As explained in the previous chapter, raw `<div>` and `<span>` XML tags should be used to enable attributes in block and inline elements, excepting titles and code snippets. From one exception and the general rule, attributes can help us to fully format books in both XHTML and ConTEXt.

The general rule is that you need divisions and spans to grant attributes to elements. Besides other features, this enables language markup in multilingual documents.

The exception to the general rule is that titles can have attributes. As explained in the next chapter, this enables special layout not only for titles themselves, but also for their text sections.

4 Titles

Titles are one of the few elements that are granted attributes in pandoc. Native attributes in titles enable the following possibilities in my documents:

- Hidden titles in some chapters, using classes.

  This makes sense in special chapters such as dedication, epigraph and colophon.

- Special formatting for single sections, with unique identifiers.

  Again, this is extremely helpful to set up different layouts for the dedication and the colophon.

- Special book parts, such as front, body and back matters, using identifiers.

Both classes and identifiers are ways to describe particular elements in the source document. They are designation modes for elements in the source. This enhanced designation enables special formatting beyond the general layout for the given element. Of course, this description requires further formatting with both cascading style sheets and within ConTEXt environments.

Attributes in titles can be also used to format the text they refer to. There is a way to wrap both the title and its text in a parent division element—a division that contains both title and its text section. In that case, attributes would belong to the parent division element. To get this

wrapping text and title divisions, `pandoc` should be invoked with the following argument:

```
--section-divs
```

This wrapping division is a key feature to be able to format the layout for whole text sections beyond its titles.

A Hidden Titles

Hidden titles are essential in some book sections. These sections provide information that has a separate page, but it lacks a title. One example would be a copyright information page.[7] Another would be an epigraph page.

The code in the source document would be:

```
# [Epigraph] {.hidden}
```

I recommend enclosing the hidden title in brackets so it would be displayed in a secondary way in the ePub file internal table of contents.

The previous sample can be formatted in CSS with the code:

```
h1.hidden {
    visibility: hidden;
    margin: 0%;
    padding: 0%;
    font-size: 0pt;
}
```

I recommend to remove any size from `margin`, `padding` and `font-size`—to set its value to zero—to avoid an unwanted extra blank space.

If you generate the XHTML code with `--section-divs`,[8] the CSS code should read:

```
.hidden > h1 {
    visibility: hidden;
    margin: 0%;
    padding: 0%;
    font-size: 0pt;
}
```

Of course, you might use other class name than `{.hidden}`. In that case, you should keep the same class name in the cascading style sheets.

B Special Sections

Special layouts for special chapters may be achieved by specifying the unique identifier in the book title.

A sample for a colophon would be:[9]

```
# Colophon {#colophon}

This book was generated with `pandoc`
(<http://pandoc.org/>) and typeset with <span
class="tex-logo">ConTeXt</span>
(<http://contextgarden.net/>).

_<span class="tex-logo">TeX</span> Gyre Pagella_, _GFS
Didot_ and _Cousine_ were the selected typefaces.
```

With the `--section-divs` argument, the whole section—part, chapter, section, or any heading—will be wrapped in a block division element. So you could format anything in that section.

A sample layout for the colophon would be:

```
#colophon {
    padding-top: 40%;
    margin-left: 10%;
```

```
    width: 80%;
    line-height: 115%;
    text-align: justify;
    font-size: 95%;
    text-indent: 0%;
    }

#colophon p {
    text-indent: 0%;
    padding-top: 0%;
    text-align: center;
    }
```

The option `--section-divs` is mandatory here. Without the division wrapper, it would be impossible to format anythning beyond the title itself.

C *Book Matters*

My first experience going from `pandoc` to ConTEXt was a real book. Books have at least three divisions above the part division. These are front matter, body matter and back matter.

These matters serve an important purpose: different numbering for pages and titles. Page numbers may be different in the front matter. Or they should be missing before the foreword or the introduction. Titles aren't numbered before the body matter. They follow a different numbering scheme in the appendices. And they aren't usually numbered in the back matter.

Book matters only make sense in the PDF output. Page numbering isn't configurable in ePub. And automatic title numbering isn't an option in ePub either. I'll explain why.

The second version of ePub lacks counters in its specification. But to display the counters, `:before` and `:after` pseudo–elements are required too. Although included in the specification for ePub version 2, they aren't honored by *Adobe Reader Mobile*.[10] So, they aren't available to the vast majority of e–ink readers. And even when *Adobe* fixes the issue, the only way to apply the fix is to—physically—upgrade the reader. I may be a harsh critic, but these upgrades are too expensive for anyone.

Since book matters are only relevant for ConTEXt, I don't recommend the following markup:

```
<div id="frontmatter">
# Foreword

# Acknowledgements

# Introduction

</div>

<div id="bodymatter">
# First Chapter

# Second Chapter

</div>

<div id="backmatter">
# Conclusions

</div>

<div id="appendices">
# First Appendix

# Second Appendix

</div>
```

The reason is very simple: this won't help the ePub conversion, but it may hurt it. As you may already know, ePub is only a `.zip` compressed file with many XHTML files inside. By default, `pandoc` splits XHTML files with the first–level title. This is fine for many documents. The behaviour can be changed with `--epub-chapter-level`. But division elements shouldn't be split. In fact, `pandoc` doesn't split them. At least, division elements shouldn't be split when they have unique identifiers. If you have two elements using the same identifier, the identifier can't be unique.

The real problem with unsplit divisions is that their XHTML files may be huge. Huge for a device—not a tablet, but an e–ink reader—with extremely limited computing resources. Although it might work fine with your device, you shouldn't expect that all electronic reading devices have the same computing resources. And you shouldn't expect either that your readers would upgrade their reading device to be able to read your works.

Book matter markup could be achieved by tagging the title that starts each book division. That way, ConTEXt would be served without interferring in the ePub generation. This would make a better version from the previous sample:[11]

```
# Foreword {.frontmatter}

# Acknowledgements

# Introduction

# First Chapter {.bodymatter}

# Second Chapter

# Conclusions {.backmatter}
```

```
# First Appendix {.appendices}

# Second Appendix
```

In the sample above, each first–level title—which is also a chapter—will have a separate XHTML file in the ePub output file. This will be faster to be processed in any computer. Even by an e–ink reader.

5 *Languages*

A Document Main Language

The language of a given document can be specified in `pandoc` with the `lang` YAML field. This works with HTML conversions. But LaTeX requires special tagging: the YAML `mainlang`. You may object that `lang` can be used for both HTML and LaTeX. It can be used, but since language codes in HTML and LaTeX differ, each value will clash when converting to the other format.[12]

The different YAML key for language in the previous paragraph may be a workaround to avoid changing the value from `lang`. Other formats won't have other document language—if any—than the default one. Since `pandoc` is modelled after XML, I'd say that LaTeX is wrong here. But don't get me wrong too. LaTeX language codes should be automatically translated from the XML values when writing to LaTeX. And vice versa.

Language markup has in hyphenation its main feature. But there are many other language–dependent features, such as indentation, quotation and other typographical conventions. Different languages solve typographical questions in different ways. Some of them may be hard–coded in the text itself. But others ones should be specified with markup.

B Language Markup

Language tagging in `pandoc` should be achieved by wrapping the element—excepting those ones which allow attributes—with a division

or span element. *Markdown* doesn't have any special markup for languages. This is due to the fact that `pandoc` lacks a special attribute for language in its internal document model. This means to the end–user that (s)he needs to hard–code languages.

Language tagging should be achived as in the following sample:

```
<span lang="de">_Begriff_</span> is the German word for
“concept”.
```

This markup has the following issues:

- Extra tagging and XML code which may even be problematic for the end user.
- Without special syntax, conversion to other formats would need extra parsing.
- Hard–coded syntax may be problematic even in HTML, if the `xml:lang` attribute is required instead.

C *Hyphenation*

ConTEXt requires language correctly specified for the whole document and the passages in foreign languages. This requirement comes from a feature which it inherits from TEX: automatic hyphenation enabled by default. In the previous sample, if you neglect to markup language, the first word may be wrong hyphenated. Just in case you wonder, US English is the default hyphenation language in TEX.

This isn't a feature for the German language only. If you happen to write your documents in UK English, your document may be wrong hyphenated. As far as I know, hyphenation rules for both US and UK English are sligthly different.[13] With any other Western language, hy-

phenation points are—high—probably wrong. Because their rules are too different.[14]

Language tagging is an essential attribute for whole documents. Otherwise, formatting some elements could be unnecessarily complicated. Hyphenation may be wrong. Not only when using ConTEXt. *iBooks* seems to activate hyphenation by default.[15] Hyphenation is an extremely useful feature: it improves text readability. There is nothing wrong in enabling it. What it may be wrong is specifying another language or using hyphenation rules from other language.

In multilingual documents, there is another issue besides hyphenation. Of course, proper markup is required to avoid wrong hyphenation points. Just as a comment, it makes a poor impression to have a high–quality typeset document with wrong hyphenations. Because people may think that the author doesn't know the hyphenation rules for the language.

D *Other Language–Dependent Issues*

Hyphenation isn't the only feature that is related to language. Indentation is a language–dependent issue. Even if continental European languages seem to neglect their own traditions in favour of typographical conventions comming from the US. Indentation in Spanish—as well as in other European languages—belongs to each paragraph, also after headings. Language markup makes easier to write source documents in different languages each and to layout them using a single format file—either cascading style sheet or ConTEXt environment.

Quotation marks are also a language–dependent issue. Imagine the following multilingual show:

English “quotes” are the Spanish «comillas», the French « guillemets » and the German »Anführungszeichen« or „Hochkommata“.

Of course, you can hard–code them. But it might not be a wise decision, because you might want—or need—to change the layout later.

You could soft–code the quotes with this markup:

```
English _quotes_{:en} are the Spanish _comillas_{:es},
the French _guillemets_{:fr} and the German
_Anführungszeichen_{:de .alt-quotes} or
_Hochkommata_{:de}.
```

In cascading style sheets, you would only need:

```
em {
    font-style: normal;
    }

em:lang(en):before {
    content: '“'
    }

em:lang(en):after {
    content: '”'
    }

em:lang(es):before {
    content: '«'
    }

em:lang(es):after {
    content: '»'
    }
```

```
em:lang(fr):before {
    content: '« '
    }
em:lang(fr):after {
    content: ' »'
    }
em:lang(de):before {
    content: '„'
    }
em:lang(de):after {
    content: '“'
    }
em.alt-quotes:lang(de):before {
    content: '»'
    }
em-alt-quotes:lang(de):after {
    content: '«'
    }
```

This approach may easily be reverted to standard emphasis layout, if required. It doesn't work with *Adobe Reader Mobile*. I agree, but I hope they will fix it one day. It can be also objected that it could be written with no special syntax for the language attribute. I agree. Only more typing and even more complicated formatting would be needed. Besides ignoring the simplicity of dealing with language as the special attribute it really is.

E Required Typefaces for Different Scripts

If you write in languages that use the Latin script—or the Latin alphabet—, you may safely ignore this section. But if you use other scripts

than Latin—and especially if you mix scripts in your documents—, reading this section might provide you some help.

Mixing scripts successfully in the same document requires typefaces with all glyphs used for all scripts contained in the document. But many typefaces only have glyphs in the Latin script. Then, defining typefaces for different scripts is the way to have all glyphs covered.[16]

CSS doesn't allow to define typefaces for scripts. It is possible to specify typefaces for languages. Imagine that you mix English and Greek in a document. A way to cover both languages in CSS would be:

```
body {
    font-family: GaramondNo8;
    }

:lang(el) {
    font-family: Theano Didot;
    }
```

Of course, you would need to have languages tagged in your document to make this work. If the document main language were English, this should be defined in the `lang` YAML field. And all Greek passages should be wrapped in `<span>` or `<div>` elements with the attribute `lang="el"`.

CSS doesn't provide a proper fallback mechanism. Or it is much simpler than you would need here. All what its fallback mechanism does is the following:

```
body {
    font-family: Palatino, Garamond, serif;
    }
```

The text would be displayed using *Palatino*. If not, *Garamond* would be used. If none of these typefaces were available, the system default serif font would be used to display the text.

This approach wouldn't solve the problem. Imagine that you have a document mixing German and Korean. If *Palatino* or *Garamond* were available, the German text would be displayed fine. But Korean passages would probably not displayed at all.

ConTEXt provides a fallback mechanism to specify typefaces for different scripts. And it has another method to hyphenate texts from different scripts, even with unmarked languages. The second chapter from the next part explains both.

6 *A Final Word on* Adobe Reader Mobile

> Systems *must* support all CSS2 selectors, including pseudo–elements and pseudo–classes.

I'm not a computer expert, but I have paged through the PDF specification[17]—reading would have required skills I'm afraid I don't have—and I warmly thank *Adobe* for making available such a clear document. I may be misled, but I tend to think that the specification is well written, if I understand some parts of it.

The opening quote in this chapter comes from the latest version of the *Open Publication Structure (OPS) 2.0.1*. I only replaced bold with italics.

The *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification* includes both pseudo–elements and pseudo–classes.[18] `:before` and `:after` are pseudo–elements. `:lang` is a pseudo–class.

Unfortunately, *Adobe Reader Mobile* doesn't seem to implement either pseudo–classes or pseudo–elements. At least, in its version 9.

As far as I know, either pseudo–classes and pseudo–elements can't be faked. Of course, you can hard–code quotes or any text string instead of using `:before` and `:after` in CSS. Hard–coding requires replacing all occurrences when requirements change

If you want to have different CSS properties for a language, you would need to add an extra class besides `:lang`. It is important to markup language for other ePub browsers and for ConTEXt. By the way, automatic hyphenation doesn't seem to be available in *Adobe Reader Mobile*

There are other basic non–working issues with *Adobe Reader Mobile*. But if your ePub document contains text in languages written from right to left, this feature is also not supported by *Adobe*. Unfortunately.

CONTEXT

7 A Comment About Versions

ConTEXt can be found in any popular TEX distribution out there. Even if you have your favourite distribution, I warmly recommend you to install the *ConTEXt Suite*. It has the latest features and it won't bother the ConTEXt version installed from your TEX distribution.

TEX distributions update ConTEXt a couple of times a year. The *ConTEXt Suite* may release improved versions many times a week. In these updates you get new features and old bugs fixed. Using a standard TEX distribution, you deal with old bugs and you also miss the new features.

Only the development version is updated. If you find bugs in an already released version, the fixes will be included in the next development version.

Both requirements—using the *ConTEXt Suite* and its development version—may sound crazy. They aren't. When I started using ConTEXt, I installed it from *TEX Live*—I don't remember which year. A participant in the mailing list recommended me to use the *ConTEXt Suite*. I installed the current version. But at some point, I realized that I was missing a feature included in the development version. So I installed it. Next time I updated my *Fedora* distribution, I didn't install *TEX Live* again. The reason was I simply didn't need it.

The ConTEXt wiki provides detailed instructions to install the *Suite*.[19] The *Suite* is a complete ConTEXt distribution. It is portable, so you can have both stable and development releases on the same computer. In fact, you can have as many releases as you want. And you will be able to use the one that best fits your needs.

If you plan to use the environment `pandoc-xhtml.tex`, please install the latest version of the *ConTEXt Suite*. And keep it updated.

8 *Invoking ConTEXt*

The standard way to invoke ConTEXt is:

```
context filename.tex
```

Of course, you should replace `filename.tex` with the name of your actual file name. You may know the following practices from `pandoc`, but I can't refrain from asking you to consider the following questions:

- Avoid spaces in file names. Hyphens and underscores may replace space characters.

  If your file includes spaces, the way to invoke ConTEXt would be:

  ```
  context "my file name.tex"
  ```

  Enclosing file names in quotes is mandatory when they contain spaces. Otherwise, any operating system would understand that the space separates two files, not two words within the same file name.

- Lowercase and uppercase characters differ. *Windows* ignores this difference in file names. Unless you know what you are doing, avoid uppercase letters. This might lead to confusion. Besides, lowercase letters are more readable.

- You can use other extension than `.tex`. Or even none. But using no extension at all makes it impossible to associate `.tex` files with an editor.

- Replacing `.tex` with another extension may not be a good choice. Only the `.tex` extension may be skipped when invoking ConTEXt. The next command is equivalent to the previous example:

  ```
  context filename
  ```

- Don't forget that the final point belongs to the extension. And it should be removed, when the extension is skipped. Otherwise, it won't compile any file which includes the extension `.tex`.[20]

For the rest of the chapter, there are some issues that you should remember:

- All execution commands—displayed in fixed–width font, except ConTEXt code—are to be typed in the console window in a single line. They have to be wrapped in this document to fit in the page size or your device screen.

- Arguments have a double hyphen preceding them. The double hyphen in the command line is what enables ConTEXt to recognize them as arguments. This recognition works, because there is no space between the double hyphen and the argument name.

  The PDF version from this document won't insert a line break between the double hyphen and the argument name. Your reading device may insert a line break after the double hyphen. Don't insert a blank space after the double hyphen, when typing arguments. Otherwise, they won't work.

A *Portable* ConTEXt Suite

If you have downloaded the *ConTEXt Suite*, you may be interested in not adding the path to your system. A reason could be having also another ConTEXt version from *TEX Live* or from the *ConTEXt Suite* itself installed.

The paths for the *ConTEXt Suite* can be loaded in *Windows* with:

```
"c:\path\where\you\have\context\tex\setuptex"
```

Paths can be loaded in any *Unix* flavour—*Linux* and *MacOS X*, among others—with:

```
source "/context/path/tex/setuptex"
```

Quotes are mandatory when there is a blank space somewhere in the path. This is a requirement for all operating systems. And you have to replace the path before `tex/setuptex`—or `tex\setuptex` in *Windows*—with the actual path where you saved *ConTEXt Suite*.

Paths should be loaded once per console window. Of course, there are better methods than typing these commands. I will explain the ones I use in the last section of this chapter. Please, read all sections inbetween to know how to activate different options in ConTEXt.

B Environments

The previous command is the standard way to compile ConTEXt source files. But to compile the XHTML output generated by `pandoc`, you need a different command:

```
contextjit --environment=environment source.xml
```

The argument can't have a blank space after or before the equal sign. You may safely remove the `.tex` extension from the `--environment` option. But the `.xml` extension from the XML file shouldn't be removed. Otherwise, compilation won't work.

There are two useful options when you compile documents with ConTEXt.

The first option is to use the `luajittex` engine, instead the `luatex` engine. The main reason is that it is slightly faster.[21] There are two ways to invoke it.

If you use *TEX Live* or even the stable version from the *ConTEXt Suite*, you should type:

```
context --engine=luajittex --environment=environment
source.xml
```

If you use the development version from the *ConTEXt Suite*, only a single command should be typed:

```
contextjit --environment=environment source.xml
```

ConTEXt generates two extra files—besides the PDF document—while compiling a source: `.log` and `.tuc` file. They won't be removed after the compilation.

The `.log` file contains the compilation log—as the extension name reveals. Unless compilation crashes, it contains information only relevant to experts. And even in that case, you have the compilation messages in the console. The `.log` file can be safely removed after compilation.

The `.tuc` file is an auxiliary file. It contains information that ConTEXt needs to generate the final PDF document. It can be safely removed. But if the file is kept, next compilation will be probably faster. For most documents, ConTEXt needs more than a run to generate the PDF document. In the second and next runs, compiling instructions are read

from auxiliary files—mainly from the `.tuc` file. If you already have the `.tuc` file, you may spare one or two ConTEXt runs.

Keeping auxiliary files is a personal option. It depends on your preferences. One has to find the balance between compilation speed and extra unnecessary files. If your computer is rather fast, removing auxiliary files may be a more suitable option.[22]

The option to remove extra files after compilation is `--purgeall`, such as in:

```
contextjit --purgeall --environment=environment
source.xml
```

A good compromise solution woud be removing only the `.log` file after compilation. `--purge` is the option to achieve it:

```
contextjit --purge --environment=environment source.xml
```

One last important consideration. When using the last two options, you should write a file name. `--purgeall` or `--purge` without a file name will remove all files in the directory with matching extensions that ConTEXt recognizes as belonging to auxiliary files.

Removal will include files with extensions from the current ConTEXt MkIV and the previous ConTEXt MkII. Please, don't forget that all files having those extensions—either generated by ConTEXt or not—will be removed. Handle with care.

D *Different Output File Name*

ConTEXt generates a PDF file with the same name than its source file. Sometimes, it is very helpful to be able to specify a different name for the PDF document.

The way to specify a different output filename is:

```
contextjit --result=my-new-filename
--environment=environment source.xml
```

It is essential to include the equal sign between the argument and the file name. No blank spaces are allowed. Otherwise, your operating system will understand that it has two files to be compiled by ConTEXt.

With this option, ConTEXt renames the PDF file after compilation. This is important to notice, since it may overwrite a previous PDF file with the source file name.

This feature is extremely handy for a script with different PDF version files from the same sources.

E Optional Compilation

Optional compilation in ConTEXt allows to have different from the same source. This is the only effective method to generate multiple versions from a single source file. When compiling XML sources—such as described in this document—, this information is contained in the environment files.

Modes are enabled from the command line with:

```
contextjit --mode=letter,footnotes
--environment=environment source.xml
```

Multiple modes can be invoked from the command line separated by commas. But in this case, no blank space is allowed after the comma. Otherwise, your operative system will assume that the mode name is a source file to be compiled.

F *Multiple Versions*

Multiple versions can be generated from the a single source file by mixing the previous features. Automatic output renaming can be adapted to optional compilation to generate different files with specific contents and layout each.

These are the commands to generate four different documents from the same source:

```
contextjit --mode=letter --result=document-letter
--environment=environment source.xml

contextjit --mode=A4 --result=document-A4
--environment=environment source.xml

contextjit --mode=letter,footnotes
--result=document-letter_footnotes
--environment=environment source.xml

contextjit --mode=A4,footnotes
--result=document-A4_footnotes
--environment=environment source.xml
```

Of course, it would be crazy to type all commands each time you want to generate the files. A script is your friend here.

G *A Single Keystroke*

The best method to write documents with `pandoc` is to type them with your favourite text editor and to visually check them in their final output format. This visual check should be done frequently, in order to see if everything works fine in your document. Unfortunately, both text and layout can go wrong.

Relevant output formats for digital editions are ePub and PDF. You don't have to check both all the time. But you have to check one of them frequently. And both format versions should pass the final check, once the document is finished.

Since the PDF document could be more problematic, I recommend to perform the frequent test on the PDF version. Contrary to ePub documents, generating a PDF file from `pandoc` with ConTEXt requires two steps. XHTML has to be generated first from the *Markdown* source file. And then, the PDF document can be compiled from the XHTML source.

Nothing prevents you from typing the two commands in your console window each time. But they are rather long and prone to typing errors. You can also make the computer invoke the complete compilation command for you. And you could completely compile a PDF file from *Markdown* sources with a single keystroke. For that, you need a programable text editor. There are many out there. Choose your favourite.

In order to generate a command that invokes both `pandoc` and then ConTEXt from any document you might want, your source files should be named after the following conventions:

- Both *Mardown* source file and ConTEXt environment should share the same name. The only difference would be their extensions: `.md` and `.tex`.

- As a general rule, I advise you to use the same name for all files related to the same *Markdown* source. Different extensions will distinguish the different files. The same name policy also includes the cascading style sheet, bibliography and cover image file. All these files will be easily identified by anyone as belonging to the same project or document.

- You may use the same ConTEXt environment file for all documents to be processed with `pandoc`. I think this is a wrong move. Unless you want all your documents with exactly the same layout. Each *Markdown* source should have its own environment.

- The previous requirement doesn't mean that you shouldn't share code among ConTEXt environments. It is wise to have all the parsing code that associates XML elements and attributes with ConTEXt elements in a single file. This file can be loaded in each single environment.

My system is *Linux* and I feel extremely comfortable with *geany*.[23] The complete command for PDF compilation from *Mardown* sources reads:[24]

```
pandoc -S -s --section-divs --template=xml.tpl -o
"%e.xml" -t html "%f" && source
/home/user/ctxbeta/tex/setuptex && contextjit
--purgeall --environment="%e" "%e.xml"
```

To avoid confusions, the command above generates a temporary XHTML file, forcing it to have a `.xml` extension. I prefer to avoid the standard `.html` extension, because I know it is only there for further PDF compilation. But this is a matter of personal taste. Remember to adjust it accordingly, if you change this.

The same compilation command can be adapted to *NotePad++*:[25]

```
c:\context\tex\setuptex &amp;&amp; cd /d
&quot;$(CURRENT_DIRECTORY)&quot; &amp;&amp; pandoc -S
-s --section-divs --template=xml.tpl -t html
&quot;$(FILE_NAME)&quot; -o
&quot;$(NAME_PART)&quot;.xml &amp;&amp; context
```

```
--engine=luajittex --purgeall
--environment=&amp;quot;$(NAME_PART)&amp;quot;
&amp;quot;$(NAME_PART)&amp;quot;.xml
```

9 *Simple Formatting*

This chapter doesn't pretend to be *A Crash Course in ConTEXt*. There would be too many casualties and few learning. It is mainly a light comment on the environment file that was used to generate the PDF version from this document.

A *Before You Start Typing*

Commands and options in ConTEXt are case–sensitive. This means that `\em`—the standard command for emphasis—isn't the same as `\EM`. The latter is an undefined command; or an undefined control sequence, as ConTEXt calls it. The same way, `a0` instead of `A0` is an invalid paper size.

The comment character in ConTEXt is the percentage sign—`%`—. Everthing you may write after the comment character in the same line is ignored by ConTEXt.

Paths within ConTEXt use the slash character—`/`—. This is also the case *Windows*. Path with backslashes—such as `c:\Documents\file.tex`—won't work in *Windows* either. Inside a ConTEXt document, it should read `c:/Documents/file.tex`, instead.

Backslash is exclusively used in ConTEXt to indicate that the following word is a command. `\chapter{My Chapter Title}` is a command to indicate a chapter title. The backslash character can be escaped with the `\letterbackslash` command.

Backslash is also used as the escape character in ConTEXt. So, the percentage sign can be included with `\%`. There is also a special command for it: `\letterpercent`.

Backslash can't be escaped with another backslash character. `\\` has a different meaning. It inserts a new line in the paragraph. The command to escape the backslash character is `\letterbackslash`, as already noted.

B The Most Basic Document

The most basic document in ConTEXt may be the following one:

```
\starttext
\ConTeXt\ is great!
\stoptext
```

Minimal required elements in any ConTEXt document are:

- `\starttext`[26] indicates where the document text starts.
- `\stoptext` indicates where the document itself stops.
- Some text or text commands in the document text body.

Setup commands are to be placed before `\starttext`. This is the document preamble.

C Language Tagging

Besides specifying the document main language in the preamble with `\mainlanguage`, ConTEXt can switch language anywhere in the text. When no language is specified for the document, the default language is U.S. English.

Language shortcuts are: `\uk`, `\de`, `\deo`, `\fr`, `\nl`, `\es`, `\sv`, `\gr`, `\pt`, `\agr`, `\la`, `\bg`, `\cz`, `\hr`, `\ro`, `\sk`, `\sl`, `\ja`, `\ua`, `\vi`.[27] The previous

list isn't complete. `\it` isn't a language command, but the command for italics. In that case, the full command has to be used: `\language[it]`.[28]

Language commands are switches. This means they change language from their point in the document. Multilingual documents may require this in some cases. The standard method to include foreign passages in a document is to limit the language command to these text portions. Nothing prevents you from switching languages as many times as you want. Alhough I think it is safer to enclose language switches. Because it may prevent some stupid mistakes.

For text passages not above a paragraph, enclosing braces may be the best option, such as in:[29]

```
{\de\em Wort} is a basic German word.
```

For text portions above a paragraph, any `\start...\stop` structure may be useful. The easiest structure might be a block quote:

```
\startnarrow\deo
„Ach! Du bist witzig!“ rief sie freundlich und
überrascht, als sie zum ersten Mal diese Äußerung
hörte.

„Ja, Miezchen, ich mache zuweilen einen recht guten
Witz,“ antwortete er trivial.
\stopnarrow
```

If we only need pure language markup for multiple paragraphs, we enclose it in the most basic structure:[30]

```
\start\deo
Ja, ja, man hat der Exempel mehrere, daß kranke Leute
gestorben sind.
```

```
Das ist dabei das allerwichtigste, was aber von gar
keiner Bedeutung ist.
\stop
```

The most important thing when working with switches is to know where they stop. It's even more important than to know where they start. Especially with languages, because wrong hyphenation may be easily overlooked.

D *Font Switches*

As it happens with languages, font commands are switches. They change the font or font family from their point to the end of the document. Unless they are enclosed.

Document typeface family can be specified with `\setupbodyfont` in the document preamble. In the document body—after `\starttext`—, there are two commands for typeface configuration: `\setupbodyfont` and `\switchtobodyfont`. The former replaces the document typeface from the point where is invoked, while the latter changes typeface only for the page body. Both commands can be enclosed the same way language commands are.

Font switching commands are:

`\em` enables emphasis.

`\it` switches to italics[31].

`\bf` activates to bold font.

`\bi` sets up the font to bold italics.

`\sc` switches to small caps.

`\tt` changes the font to monospace.

`\ss` activates to the sans–serif font.

`\tfa` increases the font size by 1.2 times. Range values go from `a` to `d`.

`\tfx` decreases the font size substracting 0.2. Possible values are `x` and `xx`).

These commands need the following considerations:

- Emphasis can be configured to use any font.[32] By default, emphasis will use the slanted font with *Latin Modern* and italics for any other typeface.

- Double emphasis—such as in `{\em this wasn't yours, it was {\em mine}}`—disables italics in the second emphasis.

- Either a font is provided, or ConTEXt can't fake it. This applies to italics, bold and small caps glyphs.

  ConTEXt is able to emulate an automatically slanted font, if required by the user. But unless you know why you are doing it, please do use a typeface that contains proper italics.[33]

- This is a personal opinion: avoid using bold fonts. They mark a too strong emphasis and distract from the rest of the text. When strictly required for emphasis, use italics.

  Good typography improves readability.

- All font or typeface commands from `\it` to `\ss` allow font size combinations. `\ssxx` would be sans–serif font with a 0.6 size

from the standard text. Or `\itd` would be italic font with a size which is 2.074 times the size of standard text.

My final recomendation with fonts is to use them sparingly. Too many fonts distract and decrease redability. Use each font—or even typeface—with a specific purpose. The natural way in ConTEXt is not to apply fonts or typefaces directly, but to use them to set up the layout in elements. This makes the task easier when you change your mind or your requirements are different. Remember that good typography and font usage always require `style`.[34]

E *Paragraph Layout*

ConTEXt builds paragraphs as *Markdown* does. You need a blank line to have different paragraphs.

```
This
builds
a
single
paragraph.

This is another one.
```

ConTEXt adds a blank space at the end of the line. To remove the space inserted between lines, add a percent sign right after the last character in the line.

```
sin%
gle
para%
graph
```

As explained in the previous section, `\\` inserts a new line within the same paragraph. `\par` inserts a new paragraph—like a blank line would do.

Interlinear space is specified with `\setupinterlinespace`. Default value is `line=2.8ex`. `ex` is a relative length unit. Its value is the width of the lowercase *x* glyph in the document's default font. A way to specify another a different value for interlinear spacing would be:

```
\setupinterlinespace
    [line=3ex]
```

Indentation is set with `\setupindenting`. The basic setup would be:

```
\setupindenting
    [medium, always]
```

Other values than `medium` are `small`, `big` or a fixed dimension—such as `1cm`, `1em` or `1ex`.

If your document needs indented paragraphs after headings—which shouldn't be the case in English or German—, you need the following setup command:

```
\setupheads
    [indentnext=yes]
```

Spacing between paragraphs is controlled by `\setupwhitespace`. Values and units are the same as the ones for `\setupindenting`.

```
\setupwhitespace
    [big]
```

Paper size can be specified with `\setuppapersize`. `A4` is the default option. To specify another paper size, such as letter size:

```
\setuppapersize
    [letter]
```

Other values would be `legal`, `tabloid`, `folio`, `executive` and sizes from the DIN/UNE A, B and C series.

Default orientation is `portrait`. Landscape orientation is specified with:

```
\setuppapersize
    [A3, landscape]
```

Page layout is specified by `\setuplayout`. A basic setup would be:

```
\setuplayout
    [backspace=2.5cm,
     width=middle,
     topspace=2cm,
     bottomspace=1.25cm,
     height=middle,
     header=0cm,
     footer=1.5cm]
```

`header` and `footer` specify dimensions for headers and footers. `backspace` sets the left margin, or the inside margin in double–sided pages. `topspace` sets the top margin. Adding `width=middle` and `height=middle` equals horizontal and vertical margins. `cutspace` and `bottomspace` specify the right and bottom margins.

Double–sided pages can be specified with the following command—which isn't the default:

```
\setupagenumbering
    [alternative=doublesided]
```

Page numbers can be removed with another option from the previous command:[35]

```
\setupagenumbering
    [location=,]
```

Possible values for `location` are `header`, `footer`, `inmargin`, `left`, `right`, `middle`. `left` and `right` are to be understood as inner and outer margins in double–sided pages.

G *Widow and Orphan Lines*

Widow, orphan and broken lines may be avoided by ConTEXt. They aren't avoided by default.

The following code removes window, orphan and broken lines:

```
\startsetups[*lessstrict]
    \setup[reset]
    \widowpenalty=10000
    \clubpenalty=10000
    \brokenpenalty=10000
\stopsetups

\setuplayout
    [setups=*lessstrict]
```

In case you don't mind to have broken words at the end of a page, you may need to set `\brokenpenalty=1` in the previous sample.

H Section Headings

Section headings in ConTEXt are: `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\subsubsubsection` and `\subsubsubsubsection`. These are seven section levels, including parts. For this explanation, ”section” is any text division that begins with a title.

Section headings are invoked such as in:

```
\chapter{My Chapter Title}
```

Font and sizes can be specified with the option `style`, such as explained before. Some examples would be:

```
\setuphead
    [section, subsection]
    [style=\em]

\setuphead
    [chapter]
    [style=\word\sca]

\setuphead
    [subsubsection]
    [style=\bix]
```

The first command configures both section and subsection titles to use italics at the default size. The second line setups chapter titles with all letters in small caps at 1.2 times the standard font size in the document. The third line specifies subsection titles in bold italics at 0.8 times the standard font size.

Section numbers may be specified with `\definestructureconversionset`, such as in:

```
\definestructureconversionset
    [sectionnumbers]
    [0, R, Characters, n, g]
    [n]

\setupheads
    [sectionconversionset=sectionnumbers]
```

In the previous sample, parts won't have a number, chapters would have uppercase Roman numbers, sections would be numbered with uppercase Latin letters, subsections with digits and subsubsections with lowercase Greek letters. After defining a numbering scheme, it should be applied to the headings (as done in the last line from the previous sample).[36]

There is also another option that controls section numbering. `sectionsegments` allows to set up which section levels are present in each heading level. Two samples show how this option works:

```
\setuphead
    [section]
    [sectionsegments=section]

\setuphead
    [chapter]
    [sectionsegments=part:chapter]
```

The first command specifies that section titles would have only the number of the section. The second line configures the chapter number to be compound by part and chapter numbers.

I Makeups

Makeups are useful to include content that always fits in a single page. This is important, because the remaining content won't be split into the

next page. This is useful for copyrights, dedication, epigraphs, colophon and similar pages. It would be risky—if not simply unwise—to wrap the table of contents in a makeup. Because contents might not fit in a single page.

By default, makeup contents are vertically centered. They don't contain any page number. Makeups are created by defining them, such as with any other element in ConTEXt:

```
\definemakeup
    [copyright]
    [top=,
     style={\tfx\setupinterlinespace},
     page=no,
     doublesided=no]
```

With double–sided pages, makeups are placed in an odd page with an empty page after it. `page=no` disables the makeup placement in an odd page and `doublesided=no` removes the blank page after the makeup. Assigning an empty value to the `top` key, places the makeup contents at the top. Contents will be placed at the bottom, if we assign an empty value to `bottom`.

Because of its special content, makeups may have special layouts. They need to have the same identifier in the first pair of braces from `\definemakeup` and `\setuplayout`:[37]

```
\setuplayout
    [epigraph]
    [backspace=.33\paperwidth,
     cutspace=\cutspace]
```

This special layout for the makeup allows to have different margins, remove header and footer. In the previous sample, right margin is kept

from the default layout and left marging is set to a third of the paper width.[38]

J *Optional Compilation*

Examples of optional compilation are versions from the same document:

- in different paper sizes,
- with different page layouts,
- containing and excluding footnotes,
- with footnotes or endnotes.

Of course, these are only some suggestions. Your document may demand other versions from the same text.

These different compilation options are called modes in ConTEXt. They are defined as in the following example:

```
\startmode[letter]
    \setuppapersize
        [letter]
\stopmode
```

There is an option to define code when not in mode:

```
\startnotmode[letter]
    \setuplayout
        [header=0cm]
\stopnotmode
```

Note that starting and closing commands should match. I think it isn't uncommon to close `\startnotmode` with the shorter `\stopmode` command.

Multiple modes sharing the same code or text may be specified separated by commas:

```
\startmode[letter, A4]
    \setupbodyfont
        [pagella]
\stopmode
```

10 Languages

This chapter explains two basic features for language handling in ConTEXt. The first feature is the font fallback mechanism. The second feature describes a method for automatic hyphenation in different scripts without language markup.

These features are independent from each other. And both of them are also independent from language markup in the source text. The reason for this independence is that both capabilities are related to scripts, not to languages. Scripts—or alphabets, if you prefer—may not be considered as different from languages. But scripts and languages do differ.

A Font Fallbacks

Font fallback—so it is called in ConTEXt—is the mechanism to provide an automatic font substitution for a glyph range when this isn't fully covered by the specified font. An example will ease the understanding of the previous statement. *GaramondNo8* can be used to typeset a document written in English. But if these document contains Russian expressions—it may be a paper on Tolstoi—, either a fallback typeface is provided for the Russian characters—such as *Theano Didot*—, or passages in Russian will only have empty glyphs. In short, Russian text will be unreadable in the PDF document generated by ConTEXt.

Typeface fallbacks make sense only with scripts. Of course, nothing prevents anyone from specifying typefaces for each language. But this would be a suboptimal approach, at best. Because it is the glyph[39] what has to be replaced. Glyphs belong to scripts. And languages use scripts.

You may need a typeface—or only the upright font—for a document that contains ancient Greek. In ConTEXt, one can deploy a command that adds a typeface switch after the language switch. But this is highly inefficient. The typeface or font doesn't depend on the language: you would use the same typeface, if you had to add text passages in modern Greek also.[40]

Automatic typeface replacement is useful to replace glyphs, independent from languages in which they are deployed. A similar scenario would be required for a German text including text passages in Ukranianian and Belarusian. Defining typefaces for languages in those cases makes so much sense as specifying typefaces for documents mixing English and German, or Latin and Italian.

The commands to define a typeface fallback in ConTEXt:

```
\definefallbackfamily
    [mainface]
    [rm]
    [GFS Didot]
    [preset=range:greek]

\definefontfamily
    [mainface]
    [rm]
    [TeX Gyre Pagella]

\definefontfamily
    [mainface]
    [mm]
    [TeX Gyre Pagella Math]

\setupbodyfont[mainface]
```

The previous sample uses *TEX Gyre Pagella* replaced with *GFS Didot* for the Greek glyphs. The replacement typeface would be used for both polytonic and monotonic Greek glyphs.[41]

The following questions should be taken in consideration.

1. "Font family" is another name for typeface, although in the command `\definefontfamily` a typeface family is actually meant.

   Not only a font group is defined, but a typeface family which may contain roman or serif, sans–serif, monospaced, math, calligraphy and handwriting typefaces. Of course, you don't need to define all typefaces, only the ones required in your document.

2. Typeface replacements—or fallbacks—have to be defined before the font family is defined. Otherwise, they won't work.

3. Both font and fallback family definitions contain the same pair of brackets.

   – The first pair sets up the typeface family name as invoked by both `\setupbodyfont` and `\switchtobodyfont`.

   – The second pair of brackets specifies the font family. Possible values are: `rm`, `ss`, `tt`, `hw`, `cg` and `mm`. Their synonyms are respectively: `serif`, `sans`, `mono`, `handwriting`, `calligraphy` and `math`.

     Just in case you wonder, the `mm` typeface is needed in unordered lists. The math font family has to be defined when other `\definefontfamily` has been used. Otherwise, ConTEXt will crash with an unnumbered list.

– The third pair invokes the typeface name. This is the name displayed by the operating system in any font dialogue in a graphical application.

– The fourth pair of brackets sets up glyph ranges and other options such as font scaling.

  A sample for other glyph ranges and scaling would be:

```
\definefallbackfamily
    [mainface]
    [rm]
    [FreeSerif]
    [preset=range:cyrillic,
    scale=1.025]
```

4. Fallbacks can be defined for each font family in the same typeface family. They have to be defined for replacements to take place with the particular typeface.

B *Automatic Hyphenation in Different Scripts*

First of all, I must warn that this isn't automatic language detection. It may have a similar functionality under some circumstances. But these circumstances are essential to make hyphenation work right.

Automatic hyphenation may work automatically by loading all hyphpenation patterns in the main document language:

```
\setuplanguage
    [en]
    [patterns={en, agr, ru}]
```

This will only work if the document main language is US English. Remember that this is the default in ConTEXt. With other languages, the following commands should be used:

```
\setuplanguage
    [es]
    [patterns={es, agr, ru}]

\mainlanguage
    [es]
```

The following questions should be observed:

1. Automatic hyphenation with the above setups will only work, if there is no language markup for Greek and Russian.

2. Hyphenation patters may be loaded with an exception: only one language per script. Also the source document should contain only one unmarked language per script. Otherwise, automatic hyphenation may be wrong in any of the languages using the script.

3. A side–effect of this method is that the minimal number of letters before and after the hyphen are the ones from the document main language. The minimal number of letters before and after the hyphen in the given language may differ.[42]

This method of automatic hyphenation without any language markup works when you only have one language per script. If you mix Russian and Ukraninan and with any of the setups provided before, you have to markup Ukranian. Otherwise both Russian and Ukranian will be hyphenated following the Russian hyphenation rules. It may be wrong in some—or in many—hyphenation points. This would be similar to

hyphenate Dutch, French, German, Italian or Spanish texts without specifying the document main language. Because the default language is US English, hyphenation points in other languages may be wrong.

11 Text Divisions

As announced in the part dedicated to `pandoc`, book matters don't make sense in XHTML outputs. They make sense in ConTEXt, since it can handle them properly.

ConTEXt enables the definition of book matters beyond the ones available: `frontmatter`, `bodymatter`, `appendices` and `backmatter`. By default, title numbering is removed in `frontmatter` and `backmatter`. And it is set to uppercase Latin numbering in `appendices`, but only for chapters—it is removed in other headings.

Section blocks—as ConTEXt calls them—can be used with the following pair of commands:

```
\startfrontmatter
    \chapter{Introduction}
\stopfrontmatter

\startbodymatter
    \chapter{The Subject-Matter}
\stopbodymatter

\startappendices
    \chapter{Appendix}
\stopappendices

\startbackmatter
    \chapter{Notes}
\stopbackmatter
```

In order to remove page numbering from all the matters before the foreword or the introduction, I had to create a new section block:

```
\definesectionblock
    [whatcomesfirst]
    [firstmatter]

\setupsectionblock
    [firstmatter]
    [page=no, after=\page]

\startsectionblockenvironment[whatcomesfirst]
    \setuppagenumbering
        [location=]
\stopsectionblockenvironment
```

The markup in text should be started and stopped with `\start`_`t``firstmatter` and `\stopfirstmatter`. At least, these tags should contain the table of contents, such as in:

```
\startfirstmatter
    \completecontent
\stopfirstmatter
```

12 Parsing XHTML Sources

Environments are the method to compile XML sources with ConTEXt. The complete documentation is to be found in *Dealing with XML in ConTEXt MkIV*.[43]

This final chapter aims to explain how ConTEXt deals with XHTML sources generated by `pandoc`. I must confess that the document cited above is beyond my understanding. As I wrote before, I'm only an average computer user, not a programer. So the following pages should be an introduction to the environment contained in the file `pandoc-xhtml.tex`.

A Basic Environment

The most basic environment for HTML would read:

```
\startxmlsetups xml:pandoc
    \xmlsetsetup{\xmldocument}
    {html}
    {xml:*}
\stopxmlsetups

\xmlregistersetup{xml:pandoc}

\startxmlsetups xml:html
    \xmlflush{#1}
\stopxmlsetups
```

This most basic environment would typeset everything from the HTML file. But no formatting at all would be considered. With this environment, the whole HTML file would result in a single paragraph.

There are three steps required by any environment:

1. Select which XML elements you want to recognize from the document and match them to ConTEXt elements.

2. Register the element list.

3. Specify how ConTEXt will handle each element.

In short, you need to choose what you want from the XML source and how ConTEXt will deal with it.

B Element Selection

You need first to select which elements you want in your ConTEXt document from the XML source document.

The basic element definition for an HTML document with only paragraphs would be the following:

```
\startxmlsetups xml:pandoc
    \xmlsetsetup {\xmldocument}
        {html|body|p}
        {xml:*}
\stopxmlsetups
```

The elements in the HTML document would be `<html>`, `<body>` and `<p>`. The previous snippet defines a list. This list must be also registered. List registration is explained in the next section. The list contains the selection of XML elements and their matching element setups.

The previous `\xmlsetsetup` command defines the basic element matching: for each XML element, a corresponding element setup will be assigned. The element setups would be `xml:html`, `xml:body` and

`xml:p`. Those element setups must be configured later or ConTEXt will ignore them.

Another way of handling XHTML elements is removing them from the selection. Since it only contains CSS code, the `<style>` element should be removed to avoid having its contents in the final PDF document:

```
\xmlsetsetup{\xmldocument}
    {style}
    {}
```

`\xmlsetsetup` is the command that matches XML elements with their ConTEXt counterparts. This command has three arguments:[44]

- The first argument is where to read the element from. `\xmldocument` should be the value here, since we are reading from the XML sources.

- The second argument is the XML element you want to select. In the previous sample, we specify `{style}`.

  This second argument allows element selection by pattern matching. There are more complex alternatives. But the most basic pattern is the element name, as shown before.

- The third argument is the ConTEXt setup that deals with the XML element.

  It has to be defined later. Otherwise, the XML element won't be included in the final PDF document.

Attribute Selection

You may select XML elements filtering them by their attributes.

Of course, you may want to select XML elements by an attribute, regardless their value. The following command would do it:

```
\xmlsetsetup {\xmldocument}
    {[@label]}
    {xml:logo}
```

This command would select any element that has a `label` attribute. This may not very helpful when working with HTML sources.

What comes more handy is to select XML elements by the value of their attributes. The most basic syntax would be:

```
\xmlsetsetup {\xmldocument}
    {[@attribute='value']}
    {xml:specialname}
```

There are three questions to be considered here:

- `@attribute` may be any attribute, such as `id` or `class`.
- `'value'` must be the exact and complete name—or string, if you prefer—as contained in the attribute.

  If your value contains a hyphen, you should type it as `\letter_percent-`.
- `xml:specialname` is the name that allows you to setup the selected XML element. It doesn't have to be unique, because you may define many ways to achieve the same result. But it should be distinctive enough for the required configuration.

An example for attribute selection by its value would be:

```
\xmlsetsetup {\xmldocument}
```

```
{[@class='tex\letterpercent-logo']}
{xml:logo}
```

Both XML elements parsed to `xml:logo` would generate TEX logos from the XML sources `<span class="tex-logo">TeX</span>` or `<span class="tex-logo">ConTeXt</span>`.

Just an important note: `\letterpercent` is required by ConTEXt to be able to find the hyphen in the attribute value, as already explained.

Complex Pattern Matching

The previous element selection is based on single elements or attribute vaules. In most cases, this is not enough. The document may require a more complex element selection.

The simple selection for elements by name, attributes by their names or its values was defined in the previous subsections as:

```
element

[@attribute]

[@attribute='value']
```

Of course, all three options are contents for the second argument of `\xmlsetsetup`.

Mixing Elements and Attributes

You may need to select an element by its name and a specific attribute value. The pattern would be:

```
element[@attribute='value']
```

This pattern could be deployed in:

```
\xmlsetsetup{\xmldocument}
    {h1[@class='part']}
    {xml:part}
```

The previous element selection sample would match the following HTML element:

```
<h1 class="part">Basic Typography</h1>
```

The XHTML code would be generated from the following *Markdown* code:

```
# Basic Typography {.part}
```

Defining Pattern Paths

But the previous setup won't work with `pandoc`. First, because `pandoc` adds other class names.[45] And ConTEXt requires the exact value match in `[@attribute='value']`. The previous expression should be read "attribute equals to value". If there were more values, it wouldn't be equal.

To match an attribute which contains the `part` value, which may also be among other values, the command should read:

```
\xmlsetsetup{\xmldocument}
    {h1[contains(@class,'part')]}
    {xml:part}
```

There is another reason because it won't work with HTML output generated by `pandoc`. Since `--section-divs` are required to be able to format whole text sections instead of only titles, title attributes would be in a parent division. In this case, the selection command should read:

```
\xmlsetsetup{\xmldocument}
    {[contains(@class,'part')]/h1}
    {xml:part}
```

The slash character—/—is the way to specify a child element in the path. In the above sample, we require that the `<h1>` element has a parent element with an class attribute that contains the value `part`.

C *List Registration*

All `\xmlsetsetup` commands define a list by being contained inside the following command:

```
\startxmlsetups xml:listnameyoumaywanttochoose
    ...
\stopxmlsetups
```

`xml:listnameyoumaywanttochoose` is the name that needs to be registered with the `\xmlregistersetup` command, such as in:

```
\xmlregistersetup{xml:listnameyoumaywanttochoose}
```

The deployment of `\xmlregistersetup` requires the following considerations:

- The registration of an actual list is mandatory to compile XML sources. If registration doesn't take place, ConTEXt will compile the environment source itself.

- The element list should have the `xml:` preffix as part of its name. Even if it might work without this preffix, it is wise to add it to aviod stupid name clashes.

- Of course, you may want to replace `xml:listnameyoumaywant`<u>`t`</u>`ochoose` with a descriptive name. Just in case it might help, I have used `xml:pandoc` for the element list.

- Althought it isn't mandatory, it is a good practice to perform the list registration *after* the element list you want to register. It improves both logical order an code readability.

D *Element Setup*

The element setup is the way to specify how ConTEXt has to handle each particular XML element.

Basic Inclusion

The most basic setup is to include the contents from element in the final PDF document. For the `<body>` element, the command would be:

```
\startxmlsetups xml:body
    \xmlflush{#1}
\stopxmlsetups
```

`\xmlflush{#1}` is the command to include the contents from the previously selected XML element.

`\startxmlsetups ... \stopxmlsetups` is the command pair that enables element setup. Since they have been already used for element selection, there are two differences that must be taken into account:

- Although you may place any element setup anywhere in you environment—except inside another element setup or the element list—, please write them *after* the list registration. This is simply to avoid a code mess.

I advise you to write setups from element ancestors to descendants. So it would be harder to miss something in the element tree.

In plain English: if `<body>` contains `<p>` and you fail to both register and setup `<body>`, you won't be able to access `<p>`. To avoid this, write setups first for `<body>` and then for `<p>`.

- The name from the element setup must match a name in the element selection list. This is the name contained in third argument—the third pair of braces—from `\xmlsetsetup`. As explained before, there will be no setup when there is no match in the element list.

- `\startxmlsetups xml:name` and `\stopxmlsetups` contain the actual ConTEXt setup for each XML selected element.

Element setup is achieved mixing commands that provide data from the XML file with other commands that set up layout for these data.

Reading Attributes

In some cases, it is important to read the value of attributes. You can have them with the following command:

```
\xmlatt{#1}{attribute}
```

The previous command will print the value of the element attribute named `attribute`. This could be applied to languages to set the document main language. The complete setup should read:

```
\startxmlsetups xml:html
```

```
    \mainlanguage[\xmlatt{#1}{lang}]
    \xmlflush{#1}
\stopxmlsetups
```

The previous setup does the following tasks:

1. It takes the root element information, since `<html>` is the root element in a XHTML file.

2. It setups the document main language using the command `\mainlanguage`. `lang` is an attribute specified for the `<html>` element by `pandoc`.

3. Since the root element contains all other elements, it is essential to include the descendant elements of `<html>`. Otherwise, the resulting PDF document would be empty.

Attribute values also help us to set text passages in other languages than the main language in the document. The required setup would be:

```
\startxmlsetups xml:lang
    \begingroup
        \language[\xmlatt{#1}{lang}]
        \xmlsetup{#1}{xml:\xmltag{#1}}
    \endgroup
\stopxmlsetups
```

The previous setup reads the language information with the same command used to configure the document main language. Since, language commands are switches, they must be enclosed. And you need to add the content itself. `\xmlsetup{#1}{xml:\xmltag{#1}}` is a more explicit version of `\xmlflush{#1}`.

Conclusion

These pages constitute a brief introductory explanation on how to use ConTEXt with `pandoc`. They describe a way to generate high–quality PDF documents.

As you can see from this document, environments in ConTEXt are powerful tools. They are much more powerful than described or deployed here. Since ConTEXt is a typesetting system, it can also generate complex textual structures. And this is beyond the aim of `pandoc`. I think environments can deal with any document `pandoc` may represent.

This document is also an example. Since the source is provided, you may use it to adapt it to your needs. If anything doesn't work as you expected, open an issue at `https://github.com/ousia/from-pandoc-to-context/issues/new`.

I want to thank once again Hans Hagen for this powerful tools environments are. Of course, my humble implementation with `pandoc` may contain errors. These are all mine. I'm not a programer and I may have not fully understood even his advices.

Epilogue

This document describes a method to typeset XHTML sources from pandoc with ConTEXt. The document itself has been generated using the method described in these pages. And this method seems to work fine.

In my personal opinion, going from pandoc to ConTEXt is the easiest and most powerful way to generate high–quality PDF documents from light–weight markup sources. If the description contained here has caught your eye, you can try and judge by yourself.

Environments are used to typeset critical editions from TEI XML sources.[46] Since critical editions are rather complex text structures, I think that this shows that ConTEXt may be powerful enough to typeset most—if not all—documents generated from *Markdown* sources.

This document is a brief introduction to environments in ConTEXt. I want to describe a more general method of electronic editing with free–software tools. Since pandoc is already well documented, ConTEXt environments are the other key for this method. With a real example, I want to describe the whole picture of digital edition with professional standards.

See you in *A Single Source*. Coming to any device near you!

A Adding a Copyright Page

pandoc has a rights field in its YAML metadata.[47]

The way to add a copyright page in the metadata would be:

```
---
rights: |

    © Year The Author. All rights reserved.

    This book doesn't constitute legal advice.
...
```

The sample above assumes that your copyright page contains more than a paragraph. Because of that you need to add the horizontal bar character—|—after the metadata field name. All paragraphs that belong to the rights field should be indented with four spaces.

CSS code for that page may be:

```
.rights {
    line-height: 120%;
    text-align: justify;
    font-size: 95%;
    text-indent: 0%;
    page-break-before: always;
    padding-top: 2%;
    }
}

.rights p {
    text-indent: 0%;
```

```
padding-top: 0%;
}
```

`page-break-before: always` is a mandatory attribute, if you require a separate copyright page. Vertical margin after a page break doesn't work—or at least, it doesn't work with *Adobe Reader Mobile*. This is the reason why `padding-top` should be specified.

Notes

1 I know there are tools to generate other formats from LaTeX sources. I'm afraid they are too complex for the average user. And these are somehow extending TeX beyond its original capabilities.

2 Since KF8, the native format for the *Amazon Kindle* is a very similar format to ePub, it might be easy to implement a native writer in `pandoc` to KF8. Right now, conversion to *Kindle* can be only achieved using `ebook-convert` from *calibre*.

3 This is the reason why `pandoc` is written always with its first letter lowercase. `Pandoc` and `pandoc` would be two different programs.

4 Available from `https://www.gnu.org/software/groff/manual/groff.pdf`.

5 Just in case you wonder, software isn't sold, only licensed. When you purchase software, you are properly purchasing a license to use the software.

6 "Text processing" isn't to be confused with "word processing".

7 In this particular case, `pandoc` provides a method to do it, as discussed in *A. Adding a Copyright Page*.

8 This option is enabled by default in ePub generation.

9 Nothing prevents from having both `{.hidden #colophon}` in a title. But don't read it as if it were a "hidden colophon".

10 This seems to be also the case with ePub version 3.

11 Using classes instead of identifiers doesn't prevent that first chapters in section blocks would have their own unique identifier each.

12 See `https://github.com/jgm/pandoc/issues/1614`.

13 I'm afraid that hyphenation rules are so complex in English, that I can't say it for sure.

14 At least German and Spanish—two languages I used to know—allow only syllabic hyphenation. Even their denominations *Silbentrennung* and *partición silábica* refer to the syllabic word–partition. Besides from the fact that syllabes are different in different languages, syllabic hyphenation isn't the case in English.

15 I don't own any *Apple* device. So I don't know what happens when no language is specified in an ePub file.

16 "Typeface" and "font" are not the same. *Palatino* and *Times* are two different typefaces. *Garamond* bold and italics are two fonts, belonging to the same typeface.

17 Available at `http://www.adobe.com/devnet/pdf/pdf_reference_archive.html`.

18 `http://www.w3.org/TR/CSS2/selector.html#pseudo-elements`.

19 Available at `http://wiki.contextgarden.net/ConTeXt_Standalone`.

20 In that case, ConTEXt would expect `filename..tex` instead of `filename.tex`.

21 Compiling the same version of the source file of this document, the speedup is over 20 % on my computer.

22 My computer is a latop, which is almost ten years old. I remove auxiliary files when compiling XML files with ConTEXt. Because if `pandoc` already generates an extra HTML file from another source, I think there are too many extra files. But this is only my personal opinion.

23 `http://geany.org/`. It has versions for both *Linux* and *Windows*.

24 There are a couple of things that you have to adjust in your computer:

- XML template name and path. It is wise to keep only a copy of your template on your computer. You can create as many soft links as you

need. If you need to replace something in the template, you will have to do it only once.

- *ConTEXt Suite* path. You can totally remove this command, if you have the paths of the ConTEXt version you want to use already specified in your system.

25 `http://notepad-plus-plus.org/`. It's available for *Windows* only.

26 It is a very common mistake to type `\startext` instead.

27 U.K. English, German new orthography from 1996, German old orthograhpy from 1901, French, Dutch, Spanish, Swedish, Greek, Portuguese, ancient Greek, Latin, Bulgarian, Czech, Hungarian, Romanian, Slovak, Slovenian, Japanese, Ukranian, Vietnamese.

28 Nothing prevents us from using full commands for languages instead of shortcuts.

29 A more complex syntax than braces is the following:

```
\begingroup\de\em Wort\endgroup\ is a German word.
```

It is clearer, since it is harder to be ignored in text. But you also have more characters to type.

30 Of course, `\definestartstop` may help here. But it is beyond what I intend here.

31 At your own risk, the slanted font can be enabled with the `\sl` command. The font must be available.

32 The command to mark emphasis with italics reads:

```
\setupbodyfontenvironment
    [default]
    [em=italic]
```

33 It makes no sense to use a document typesetting system when one doesn't care about basic typography.

34 The option named `style` is used to specify the font in all ConTEXt elements that need it.

35 Just in case it isn't clear enough: the option needs to be empty.

36 Of course, you may use another identifier than `sectionnumbers` for `\definestructureconversionset`. All you have to do is to write the same identifier as value to the `sectionconversionset` option.

37 Identifier names from the previous and following code snippets don't match: `copyright` and `epigraph`. It's intended.

38 Of course, `backspace=10cm` is also a valid value.

39 A glyph is a visual representation of a character in a given font. *ß* is a lowercase letter—used in German—mapped as a computer character. Some fonts may define that glyph. But the same typeface may contain a given glyph in some fonts and not in other fonts. All fonts belonging to the same typeface.

40 Just in case it isn't clear: the Latin alphabet is used for virtually all western European languages. This is a single script deployed in many languages.

41 The polytonic system was part of the standard orthography for ancient and modern Greek up to 1982. It was replaced by the monotonic system.

42 A minimal sample shows the issue:

```
\setuplanguage
    [en]
    [patterns={en, agr}]

\setuplanguage
    [de]
    [patterns={de, agr}]

\setuplanguage
```

```
    [es]
    [patterns={es, agr}]

\mainlanguage
    [de]

\setupwhitespace
    [big]

\definefontfamily
    [mainface]
    [rm]
    [FreeSerif]

\setupbodyfont
    [mainface]

\starttext
\hsize\zeropoint
legible χαλεπα {\agr χαλεπα}

\en legible χαλεπα {\agr χαλεπα}

\es legible χαλεπα {\agr χαλεπα}
\stoptext
```

Hyphenation points for χαλεπα are *χα-λε-πα*—as shown in the example.

- German doesn't hyphenate the unmarked Greek word. German requires a minimum of three letters before and after the hyphenation.

- English doesn't hyphenate the first syllabe in the untagged Greek word. English requires at three letters after the hyphenation.

- Spanish fully hyphenates the unmarked Greek word. It only requires a minimum of two letters before and after the hyphenation.

I know that χαλεπα should read χαλεπά instead. But otherwise, it won't be displayed with *TEX Gyre Pagella*.

43 Available from `http://pragma-ade.com/general/manuals/xml-mkiv.pdf`.

44 Arguments in ConTEXt are enclosed in braces. `\xmlsetup` has three arguments, each one inside each pair of braces.

45 For first–level headings, it adds the class values `section` and `level1`.

46 See Thomas A. Schmitz, *Kritische Editionen mit TEI XML und ConTEXt*, in `http://www.dante.de/events/Archiv/dante2011/programm/vortraege/folien-ts.pdf`.

47 This YAML metadata field is a workaround to have the copyright information before the table of contents.

Since this document doesn't really require this `rights` metadata field, it doesn't use it.

This book was generated with pandoc
(http://pandoc.org/) and typeset with
ConTEXt (http://contextgarden.net/).

TEX Gyre Pagella, *GFS Didot* and
Cousine were the selected typefaces.