Producing different forms of output from XML via ConTEXt

Thomas A. Schmitz

This paper will describe the workflow that I have been using for about fifteen years now at Bonn University. I am a professor of ancient Greek and give a lecture course of about 90 minutes (i.e., 2 × 45 minutes) every week during the term. Over the course of these fifteen years, I have changed and adapted my workflow and the tools I use: after experimenting with LATEX, I used ConTEXt first to produce my lecture notes, then to produce the slides as well. After a while, it became clear that it would be beneficial to produce all material from one and only one source file. It turned out that XML was the best input format for this: it is easier to reuse for other purposes, and it allows output in different formats because of the way it is processed in ConTEXt.

Since a picture says more than a thousand words, the easiest way of demonstrating what "different forms of output" means is showing a few examples.

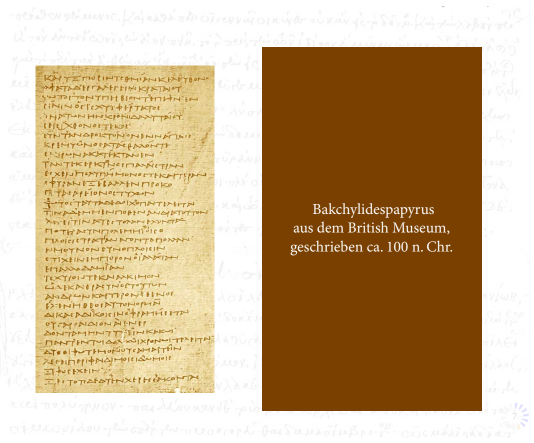
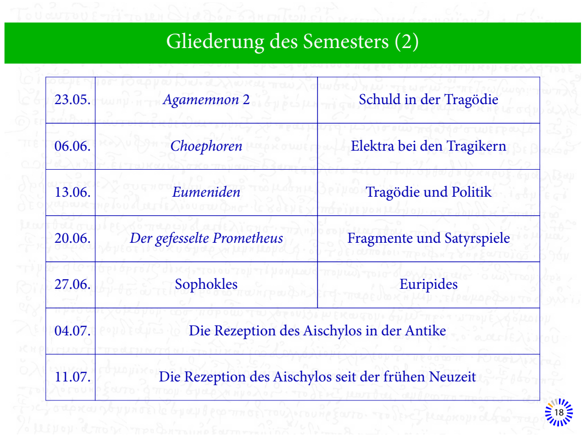

Bakchylidespapyrus
aus dem British Museum,
geschrieben ca. 100 n. Chr.

Figure 1: Slide with picture

The most important format is of course the slides that I show in my lectures. They should help students understand and master the subject; they contain different forms of material, such as pictures (figure 1 shows a slide with a picture of a papyrus fragment).

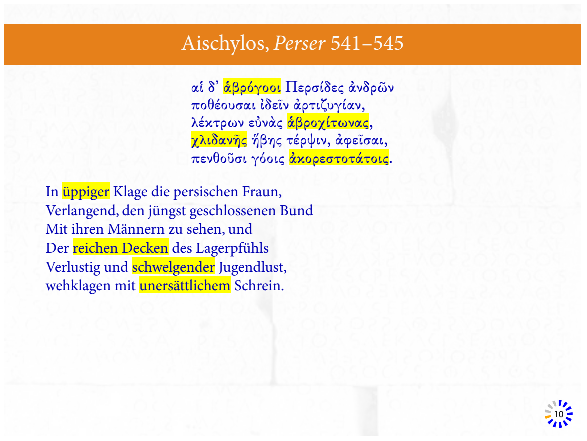

Gliederung des Semesters (2)

23.05.	*Agamemnon 2*	Schuld in der Tragödie
06.06.	*Choephoren*	Elektra bei den Tragikern
13.06.	*Eumeniden*	Tragödie und Politik
20.06.	*Der gefesselte Prometheus*	Fragmente und Satyrspiele
27.06.	Sophokles	Euripides
04.07.	Die Rezeption des Aischylos in der Antike	
11.07.	Die Rezeption des Aischylos seit der frühen Neuzeit	

18

Figure 2: Slide with table

The slide in figure 2 contains a table, another frequent format; but most of my slides contain texts and translations, like the one shown in figure 3; I teach a philological discipline, after all.


Aischylos, *Perser* 541–545

αἱ δ' ἁβρόγοοι Περσίδες ἀνδρῶν
ποθέουσαι ἰδεῖν ἀρτιζυγίαν,
λέκτρων εὐνὰς ἁβροχίτωνας,
χλιδανῆς ἥβης τέρψιν, ἀφεῖσαι,
πενθοῦσι γόοις ἀκορεστοτάτοις.

In üppiger Klage die persischen Fraun,
Verlangend, den jüngst geschlossenen Bund
Mit ihren Männern zu sehen, und
Der reichen Decken des Lagerpfühls
Verlustig und schwelgender Jugendlust,
wehklagen mit unersättlichem Schrein.

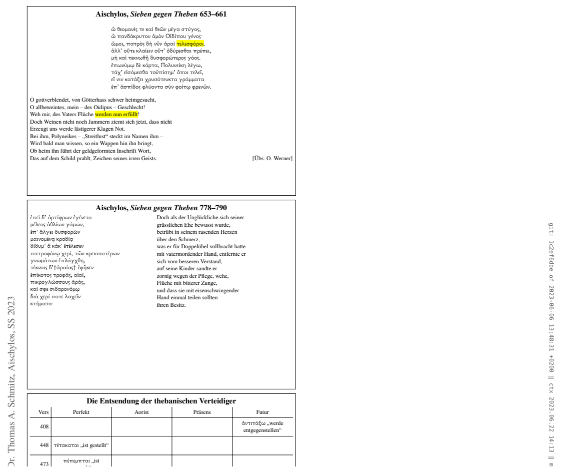
10

Figure 3: Slide with text

These slides have been produced with the ConTEXt module `simpleslides` that Aditya Mahajan and myself wrote many years ago. It offers a number of visually appealing styles for such presentations, and of course, it is free and open software and can be downloaded from the ConTEXt garden.

I had always made these slides available to students, originally by uploading them to my departmental website. Then, some years ago, students asked me to provide them in a format that would be easier for them to print and reuse.

doi.org/10.47397/tb/44-2/tb137schmitz-xmlcontext



Aischylos, *Sieben gegen Theben* 653–661

ὦ θεομανές τε καὶ θεῶν μέγα στύγος,
ὦ πανδάκρυτον ἁμὸν Οἰδίπου γένος·
ὤμοι, πατρὸς δὴ νῦν ἀραὶ τελεσφόροι.
ἀλλ’ οὔτε κλαίειν οὔτ’ ὀδύρεσθαι πρέπει,
μὴ καὶ τεκνωθῇ δυσφορώτερος γόος.
ἐπωνύμῳ δὲ κάρτα, Πολυνείκη λέγω,
τάχ’ εἰσόμεσθα τοὐπίσημ’ ὅποι τελεῖ,
εἴ νιν κατάξει χρυσότευκτα γράμματα
ἐπ’ ἀσπίδος φλύοντα σὺν φοίτῳ φρενῶν.

O gottverblendet, von Götterhass schwer heimgesucht,
O allbeweintes, mein – des Oidipus – Geschlecht!
Weh mir, des Vaters Flüche werden nun erfüllt!
Doch Weinen nicht noch Jammern ziemt sich jetzt, dass nicht
Erzeugt uns werde lästigerer Klagen Not.
Bei ihm, Polyneikes – „Streitlust“ steckt im Namen ihm –
Wird bald man wissen, so ein Wappen hin ihn bringt,
Ob heim ihn führt der geldgeformten Inschrift Wort,
Das auf dem Schild prahlt, Zeichen seines irren Geists. [Übs. O. Werner]

Aischylos, *Sieben gegen Theben* 778–790

ἐπεὶ δ’ ἀρτίφρων ἐγένετο	Doch als der Unglückliche sich seiner
μέλεος ἀθλίων γάμων,	grässlichen Ehe bewusst wurde,
ἐπ’ ἄλγει δυσφορῶν	betrübt in seinem rasenden Herzen
μαινομένᾳ κραδίᾳ	über den Schmerz,
δίδυμ’ ἅ κάκ’ ἐτέλεσεν	was er für Doppelübel vollbracht hatte
πατροφόνῳ χερί, τῶν κρεισσοτέρων	mit vatermordender Hand, entfernte er
γνωμάτων ἐπλάγχθη,	sich vom besseren Verstand,
τέκνοις δ’†ἀραίας† ἐφῆκεν	auf seine Kinder sandte er
ἐπίκοτος τροφᾶς, αἰαῖ,	zornig wegen der Pflege, wehe,
πικρογλώσσους ἀράς,	Flüche mit bitterer Zunge,
καί σφε σιδαρονόμῳ	und dass sie mit eisenschwingender
διὰ χερί ποτε λαχεῖν	Hand einmal teilen sollten
κτήματα·	ihren Besitz.

Die Entsendung der thebanischen Verteidiger

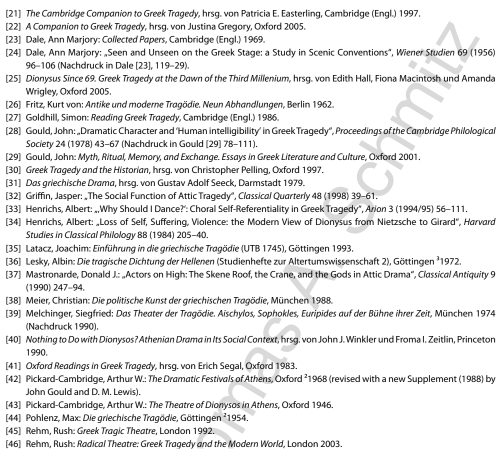
Vers	Perfekt	Aorist	Präsens	Futur
408				ἀντιτάξω „werde entgegenstellen“
448	τέτακαται „ist gestellt“			
473	πέπεμπται „ist			

Dr. Thomas A. Schmitz, Aischylos, SS 2023

git: 1c2ef6dbe of 2023-06-06 13:48:31 +0200 ‖ ctx 2023.06.22 14:13 ‖ m

Figure 4: Printout for students

The format shown in figure 4 is a design that the students themselves suggested: on an A4 sheet, the left side has a very simplified and scaled down view of the slides, minus all the decorations, backgrounds, and fancy effects, but still keeping all the important elements such as images, tables, and text; the right column is blank so students can take notes next to the slides and thus be sure what their notes refer to.

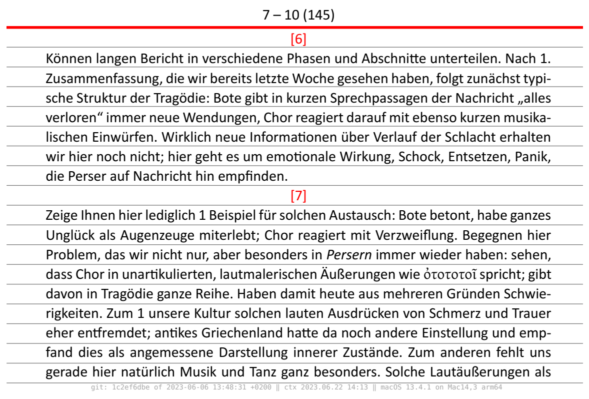


[21] *The Cambridge Companion to Greek Tragedy*, hrsg. von Patricia E. Easterling, Cambridge (Engl.) 1997.
[22] *A Companion to Greek Tragedy*, hrsg. von Justina Gregory, Oxford 2005.
[23] Dale, Ann Marjory: *Collected Papers*, Cambridge (Engl.) 1969.
[24] Dale, Ann Marjory: „Seen and Unseen on the Greek Stage: a Study in Scenic Conventions“, *Wiener Studien* 69 (1956) 96–106 (Nachdruck in Dale [23], 119–29).
[25] *Dionysus Since 69. Greek Tragedy at the Dawn of the Third Millenium*, hrsg. von Edith Hall, Fiona Macintosh und Amanda Wrigley, Oxford 2005.
[26] Fritz, Kurt von: *Antike und moderne Tragödie. Neun Abhandlungen*, Berlin 1962.
[27] Goldhill, Simon: *Reading Greek Tragedy*, Cambridge (Engl.) 1986.
[28] Gould, John: „Dramatic Character and 'Human intelligibility' in Greek Tragedy“, *Proceedings of the Cambridge Philological Society* 24 (1978) 43–67 (Nachdruck in Gould [29] 78–111).
[29] Gould, John: *Myth, Ritual, Memory, and Exchange. Essays in Greek Literature and Culture*, Oxford 2001.
[30] *Greek Tragedy and the Historian*, hrsg. von Christopher Pelling, Oxford 1997.
[31] *Das griechische Drama*, hrsg. von Gustav Adolf Seeck, Darmstadt 1979.
[32] Griffin, Jasper: „The Social Function of Attic Tragedy“, *Classical Quarterly* 48 (1998) 39–61.
[33] Henrichs, Albert: „'Why Should I Dance?': Choral Self-Referentiality in Greek Tragedy“, *Arion* 3 (1994/95) 56–111.
[34] Henrichs, Albert: „Loss of Self, Suffering, Violence: the Modern View of Dionysus from Nietzsche to Girard“, *Harvard Studies in Classical Philology* 88 (1984) 205–40.
[35] Latacz, Joachim: *Einführung in die griechische Tragödie* (UTB 1745), Göttingen 1993.
[36] Lesky, Albin: *Die tragische Dichtung der Hellenen* (Studienhefte zur Altertumswissenschaft 2), Göttingen ^{3}1972.
[37] Mastronarde, Donald J.: „Actors on High: The Skene Roof, the Crane, and the Gods in Attic Drama“, *Classical Antiquity* 9 (1990) 247–94.
[38] Meier, Christian: *Die politische Kunst der griechischen Tragödie*, München 1988.
[39] Melchinger, Siegfried: *Das Theater der Tragödie. Aischylos, Sophokles, Euripides auf der Bühne ihrer Zeit*, München 1974 (Nachdruck 1990).
[40] *Nothing to Do with Dionysos? Athenian Drama in Its Social Context*, hrsg. von John J. Winkler und Froma I. Zeitlin, Princeton 1990.
[41] *Oxford Readings in Greek Tragedy*, hrsg. von Erich Segal, Oxford 1983.
[42] Pickard-Cambridge, Arthur W.: *The Dramatic Festivals of Athens*, Oxford ^{2}1968 (revised with a new Supplement (1988) by John Gould and D. M. Lewis).
[43] Pickard-Cambridge, Arthur W.: *The Theatre of Dionysos in Athens*, Oxford 1946.
[44] Pohlenz, Max: *Die griechische Tragödie*, Göttingen ^{2}1954.
[45] Rehm, Rush: *Greek Tragic Theatre*, London 1992.
[46] Rehm, Rush: *Radical Theatre: Greek Tragedy and the Modern World*, London 2003.

Figure 5: Bibliography

Finally, I produce handouts from my source file. For regular lecture courses, these contain just the bibliography; for invited lectures or conference talks, they may also contain passages from original texts, translations, or excerpts from scholarly literature. As you can see in figure 5, I am very fond of numbered bibliographical styles because they make it so much easier and more efficient to refer to single items on the list.

What we have seen so far is, if you like, the client-facing side of my business, but of course the material I produce for myself, i.e., the notes for my lectures, is as important as these documents.



7 – 10 (145)

[6]

Können langen Bericht in verschiedene Phasen und Abschnitte unterteilen. Nach 1. Zusammenfassung, die wir bereits letzte Woche gesehen haben, folgt zunächst typische Struktur der Tragödie: Bote gibt in kurzen Sprechpassagen der Nachricht „alles verloren“ immer neue Wendungen, Chor reagiert darauf mit ebenso kurzen musikalischen Einwürfen. Wirklich neue Informationen über Verlauf der Schlacht erhalten wir hier noch nicht; hier geht es um emotionale Wirkung, Schock, Entsetzen, Panik, die Perser auf Nachricht hin empfinden.

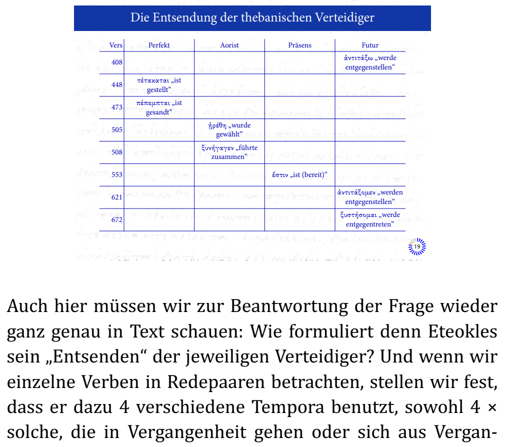
[7]

Zeige Ihnen hier lediglich 1 Beispiel für solchen Austausch: Bote betont, habe ganzes Unglück als Augenzeuge miterlebt; Chor reagiert mit Verzweiflung. Begegnen hier Problem, das wir nicht nur, aber besonders in *Persern* immer wieder haben: sehen, dass Chor in unartikulierten, lautmalerischen Äußerungen wie ὀτοτοτοῖ spricht; gibt davon in Tragödie ganze Reihe. Haben damit heute aus mehreren Gründen Schwierigkeiten. Zum 1 unsere Kultur solchen lauten Ausdrücken von Schmerz und Trauer eher entfremdet; antikes Griechenland hatte da noch andere Einstellung und empfand dies als angemessene Darstellung innerer Zustände. Zum anderen fehlt uns gerade hier natürlich Musik und Tanz ganz besonders. Solche Lautäußerungen als

git: 1c2ef6dbe of 2023-06-06 13:48:31 +0200 ‖ ctx 2023.06.22 14:13 ‖ macOS 13.4.1 on Mac14,3 arm64

Figure 6: Manuscript as “index cards”

Figure 6 shows a no-nonsense, very basic format for my manuscript: the locations where I have to change slides are simply indicated by red numbers (grayscaled for print); the slides themselves are not rendered. This is wonderfully readable on small devices such as phones or tablets, and it has the benefit that the amount of text on these “index cards” is pretty standardized, so I know exactly how many of these pages it takes to fill a 45-minute presentation; this helps me preparing my lectures.



Die Entsendung der thebanischen Verteidiger

Vers	Perfekt	Aorist	Präsens	Futur
408				ἀντιτάξω „werde entgegenstellen“
448	τέτακαται „ist gestellt“			
473	πέπεμπται „ist gesandt“			
505		ᾑρέθη „wurde gewählt“		
508		ξυνήγαγεν „führte zusammen“		
553			ἔστιν „ist (bereit)“	
621				ἀντιτάξομεν „werden entgegenstellen“
672				ξυστήσομαι „werde entgegentreten“

19

Auch hier müssen wir zur Beantwortung der Frage wieder ganz genau in Text schauen: Wie formuliert denn Eteokles sein „Entsenden“ der jeweiligen Verteidiger? Und wenn wir einzelne Verben in Redepaaren betrachten, stellen wir fest, dass er dazu 4 verschiedene Tempora benutzt, sowohl 4 × solche, die in Vergangenheit gehen oder sich aus Vergan-

Figure 7: Manuscript with view of slides

In general, however, I prefer to see the slides on my tablet when I give the lecture, in exactly the form that the students see them on the big screen behind me so I have text and translation or other elements before me when I talk about them; hence I produce the format shown in figure 7 that has the slide on top of the page and then the text of the lecture below.

All these different formats and outputs are produced from one common source file written in XML, `lecture.xml`. This file is compiled with ConTEXt. I will provide just a few basic pieces of information about the way ConTEXt processes XML; for readers who want to learn about this topic in depth, there is the document *Dealing with XML in ConTEXt MkIV* in the ConTEXt distribution that contains all the details. Like most things in ConTEXt, processing XML is done via setups, which are usually collected in a special "environment" file.

```
\startxmlsetups xml:presentation_setups
   \xmlsetsetup {#1} {*} {-}
   \xmlsetsetup {#1} {lecture|
                    presentation} {xml:*}
\stopxmlsetups
\xmlregistersetup {xml:presentation_setups}
```

The code above shows the beginning of such an environment file. Setups are collected in `start/stop` pairs; in this case, they are the setups for the entire document. Line 2 is important: it tells ConTEXt to drop everything and not typeset any XML element unless it is explicitly mentioned in the following list and has its proper setup; if this line is not present, ConTEXt will use some heuristics to retrieve and typeset at least the text from all elements. And then follows this list of elements, hierarchical from the root element down to all the other subelements.

```
\startxmlsetups xml:presentation
  \xmlflush {#1}
  \par
\stopxmlsetups

\startxmlsetups xml:presentation
  \color [red] {\xmlflush {#1}}
  \page
\stopxmlsetups

\startxmlsetups xml:presentation
  \startsection [title=\xmlatt {#1} {title}]
    \xmlflush {#1}
  \stopsection
\stopxmlsetups
```

After collecting the elements that we want processed, we create setups for every single one of them; they all live in the `xml:` namespace. Just to provide an idea of what these setups look like, we see above some examples of how an element with the name `presentation` could be processed. The first setup is the most basic one: the command `\xmlflush` on l.2 takes the content of the current element (in our case the element `presentation`; the `#1` stands for the content of the current XML node) and puts its content into the TEX stream, where text will be typeset, subelements will be processed according to the rules defined in their own setups, etc. After every element, ConTEXt will insert a paragraph break. Of course, you can apply all the usual ConTEXt commands to the content of the element, as the second example shows: here, we flush the content inside a command that will apply the color red to all text, and we insert a page break at the end of the element. The third setup assumes that the element `presentation` has an attribute `title`, which can be retrieved and typeset with the command `\xmlatt` and the name of the attribute; here, we use this attribute as the title of a ConTEXt section in our document, then flush its content inside this `start/stop` pair.

```
\startxmlsetups xml:framed
  \xmlfunction {#1} {framed}
\stopxmlsetups
```

It is also possible to process the content of our XML in Lua, as the code above shows. In this case, we hand the content of the element `framed` over to a Lua function `xml.functions.framed`, which we define like this:

```
function xml.functions.framed (t)
  if tex.modes ["print"] then
    scale_factor = 0.5
  else
    scale_factor = 1
  end

  context.framed (
    { width = scale_factor *
       tex.dimen ["textwidth"] },
        function () lxml.flush (t) end )
end
```

Within this Lua function, we have access both to the power of the Lua programming language and to all ConTEXt commands, and this makes some tasks somewhat easier. The example here is quite basic to provide an idea of what can be done: first, we define a numerical value `scale_factor` that depends on the ConTEXt mode: if processing takes place in mode `print` (this is the mode that produces the handout where slides are typeset with a reduced width), this value is set to `0.5`, in all other modes it is set to `1`. We then use this value in calculating the width of our frame: we multiply it with the dimension `textwidth`. The result will be that this frame will be spread across the entire page in other modes and across half of the page in mode `print`. Within this frame, we then flush the content of our element. The last line of our code shows a difficulty: some ConTEXt commands need to be finished before typesetting can begin; this is done with the somewhat clunky `function ... end` structure in this line. If you want to learn more about these details, you can have a look at the manual *ConTEXt Lua documents*, which is also part of the ConTEXt distribution.

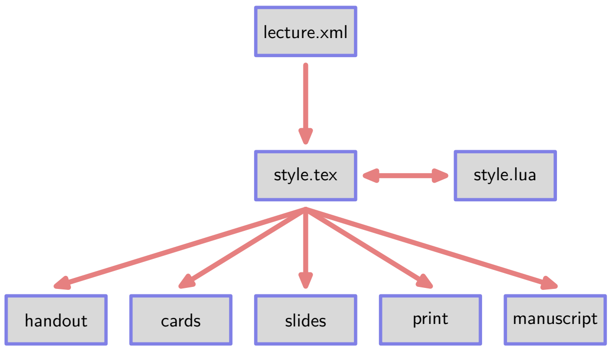


Figure 8: Schema of the workflow

This, then, is a short summary of the way in which ConTEXt processes XML. Knowing these general rules, we now understand the workflow that I have created for my lecture; you find a schematic representation in figure 8: in order to compile the `xml` file `lecture.xml`, we need a ConTEXt style file `style.tex`, which is connected with a Lua file `style.lua`, where Lua commands are defined; both files control the typesetting process. I will describe some of the salient features that allow me to derive different forms of output from the same XML file.

```
<lecture language="en" style="BigNumber">
  <presentation date="23_05_16">
    <title>The <emph>Persae</emph></title>
    <content>
      <p>Text ... </p>
      <slide> ... </slide>
    </content>
  </presentation>
  <presentation date="23_05_23">
    <title>The <emph>Supplices</emph></title>
    <content>
      <p>More Text ... </p>
      <slide> ... </slide>
    </content>
  </presentation>
  <presentation date="23_05_30">
    <title>The <emph>Agamemnon</emph></title>
    <content>
      <p>And still more Text ... </p>
      <slide> ... </slide>
    </content>
  </presentation>
</lecture>
```

We begin by taking a brief look at the main structure of my `lecture.xml` file, which is summarized in the code sample above. The text for the entire lecture course of one semester is contained in this XML file, which will grow to around 30,000 lines over the course of the teaching term. The structure is quite simple: the root element `lecture` consists of a number of individual `presentations`; each one of them has a an attribute `date` and subelements `title` and `content`; this content consists of the text (i.e., my lecture notes) and the slides that will be shown on the screen.

When we compile this document, it is obviously efficient to have all individual presentations in one document: this way, it is easy to create cross-references, a bibliography with consistent numbering, and to move content around. However, when we want to compile the slides for an individual lecture, we do not want all the slides for all presentations included in the resulting PDF file (which would become huge in size because of included images, etc.), but just the ones for the current lecture. This is why every individual lecture has a `date` tag. We will use this attribute to filter individual lectures by making use of a feature of ConTEXt that is called "modes". Modes provide a means for conditional typesetting; they allow us to include or exclude parts of our source into the typesetting process.

```
\doifmode {presentation} {
 \doifmode {\xmlatt {#1} {date}} {
   \setupTitle [title = {\xmltext {#1} {title}}]
   \xmltext {#1} {content} \page
 }
}
```

The code above shows how this works. We first define a block that will only be processed when ConTEXt is in mode `presentation`; this is what the first `\doifmode` line does. Inside this block, we nest a second mode, which is set from the `date` attribute of our `presentation` element: ConTEXt will execute the following code only when its mode is equal to this `date` attribute. It will skip over all other `presentation` elements and only typeset the one that corresponds to its mode (i.e., its date) if and only if it is in `presentation` mode; if it is in any other mode (typesetting the manuscript or the bibliography), it will process all `presentation` elements.

We pass these modes to ConTEXt when we call it from the command line. So in our case, the call would be `context --environment=lecture-style --mode=presentation,23_05_23`, and ConTEXt will process only the `presentation` element with the `date` attribute of 23_05_23.

```
\startxmlsetups xml:slide
  \startmode [presentation]
    \xmlfunction {#1} {presentation_slide}
  \stopmode
  \startmode [cards]
    \incrementcounter [slide_number]
    \midaligned {\color [red]
      {[{\rawcounter [slide_number]}]}}
  \stopmode
  \startmode [combined]
    \xmlfunction {#1} {combined_slide}
  \stopxmlsetups
\stopxmlsetups
```

Here is another examples where the modes mechanism allows conditional typesetting and filtering of content. For the element `slide`, we define three different setups, dependent on the mode we are using. When we are in mode `presentation`, ConTEXt will pass the entire content of this element on to the Lua function `presentation_slide`, where we take care of the different subelements, such as including pictures, typesetting tables and text, etc.

However, when we are in mode `cards` and want to produce our index cards, something else happens: ConTEXt will simply increase a special counter `slide_number` for slides by 1 and typeset the result on its own line; we have already seen the result in figure 6. This way, every place where we need to advance our slides is clearly marked in our manuscript; we will not run the risk of forgetting that a new slide was supposed to come up.

When we are in `combined` mode, the element content is passed to a Lua function `combined_slide` that is defined in our `style.lua` file. We have seen the general outlines of how such Lua functions work; here we find another example of things that may be easier to handle in Lua than in TEX.

```
local i = 1
local ctx = context

function xml.functions.combined_slide (t)
  i = i + 1
  local current = xml.attribute
    (t, "../../", "tag", "")
  local textwidth = tex.dimen.textwidth
  ctx.page ()
  ctx.framed ( { width=number.todimen(textwidth)
    frame="off", align="middle",
    height="10cm" }
         function ()
         ctx.externalfigure({ "presentations/"
            .. current .. ".pdf"},
         { page=i, width="13cm" } )
         end )
       ctx.blank { "line" }
end
```

ConTEXt converts every XML element into a Lua table, and we pass this table `t` as an object to the Lua function `combined_slide`. We retrieve the date of the current presentation in the variable `current`. Since we know that the element `presentation` is “grandparent” of the element `slide`, we have to move two levels up to find this attribute; this is what the expression `xml.attribute (t, "../../", "tag", "")` does. This information will be used when we construct the object of the ConTEXt command `\externalfigure`: the slides for every single presentation are stored as PDF files in a subdirectory `presentations/`, and they are named with the value of their `date` attribute, so a presentation shown on July 15 would be named `23_07_15.pdf`. When we concatenate the strings and variables `"presentations/" .. current .. ".pdf"` in Lua, we make sure that ConTEXt will use this file as an external figure. Finally, we increase the counter `i` each time this macro is called; this counter is used to retrieve the single pages of our PDF file, so they will be shown one by one, in ascending order. Before the picture of the slide, ConTEXt inserts a page break. Again, we have seen the result of this code above in figure 7: a small image of every slide will be on top of the page or pages containing the notes to the slide.

This, then, is the general mechanism to produce different forms of output from a single XML file: we use ConTEXt modes to apply different setups to different XML elements; depending on which mode we compile with, the slides from a certain presentation or the text for the notes or the bibliography will be typeset; instead of typesetting the slides, we may increase a counter or include them as images. I hope that the general principle is clear now. I want to conclude this brief overview with two special cases that have been useful over years.

As I have shown, XML is a versatile and self-testing input format: if the XML code is valid, it should compile in ConTEXt; you do not have to worry about closing groups and nesting brackets. Nevertheless, it necessitates some work: for every detail you want to typeset in your manuscript or on your slides, you have to come up with XML code to represent it and with a setup to translate it into something ConTEXt can process.

This is worthwhile for code that you use over and over again, but it makes it more difficult to write small adjustments. And certain tasks are much easier to code in TEX rather than in XML. One example is MetaPost graphics. In theory, it would be possible to write XML code that would then be translated to MetaPost code, typeset by ConTEXt and displayed on your slides. But this would be exceedingly painful and demand lots of work. It would be preferable to have some way of simply writing MetaPost code (or even arbitrary TEX code) and have it executed when ConTEXt processes our XML file. ConTEXt has a macro `\processTEXbuffer` that does just this: it executes included TEX code and typesets the result. We use this mechanism by defining an XML element `processbuffer` and writing a setup for it; the code below shows how to do this.

```
\startxmlsetups xml:processbuffer
  \processTEXbuffer [\xmlflush {#1}]
\stopxmlsetups

\doiffileelse {presentation_buffers.tex}
       {\input presentation_buffers}
       {\relax}
```

Moreover, I have found it more efficient to collect all the different TEX buffers for my lectures in one TEX file `presentation_buffers.tex`. Within this file, I have the TEX code for my various buffers, e.g., the MetaPost code that produces figure 8 (I include a short extract only).

```
\startbuffer [workflow]
\startmode [presentation,combined]
\setupMPvariables [workflow] [a=0.5in, b=0.2in]
\stopmode
\startmode [print]
\setupMPvariables [workflow] [a=0.25in, b=0.1in]
\stopmode

\startuniqueMPgraphic{workflow}
numeric a, b ; a = \MPvar {a} ; b = \MPvar {b} ;
path p[] , q[] ;

p[1] := fullsquare xyscaled (a,b) ;
fill p[1] withcolor \MPcolor {lightgray} ;
label (textext ("slides"), center p[1]) ;
\stopuniqueMPgraphic

\uniqueMPgraphic {workflow}
\stopbuffer
```

This way, our XML file can simply call the buffers in this file inside the element `processbuffer`:

```
<slide>
  <slidecontent>
    <processbuffer>workflow</processbuffer>
  </slidecontent>
</slide>
```

Finally, typing these long ConTEXt calls with the proper `--environment` and the proper `--mode` on the command line was pretty cumbersome, so I delegated this part to a Makefile, where I specify the date of the presentation I want to typeset in a variable `day`; a short extract of this Makefile would look like this:

```
combined: lecture.xml
       context --environment=style \
         --mode=presentation,$(day) lecture.xml
       cp lecture.pdf ./presentations/$(day).pdf
       #
       context --environment=style \
         --mode=combined,$(day) lecture.xml
```

When I call this Makefile with the call `make day=23_05_17 combined`, ConTEXt will first compile in mode `presentation`, then copy the current presentation into the proper directory under the correct name, then run once again in mode `combined` and include the single slides into the lecture notes. This is fairly simple, but it saves a lot of typing over the course of a semester.

I hope this paper has raised some interest and shown the advantages of producing all our output for a lecture from one single XML file.

◇ Thomas A. Schmitz
Institut für Klassische und
Romanische Philologie
Universität Bonn
Rabinstraße 8
53111 Bonn
Germany
`thomas dot schmitz (at) uni-bonn dot de`