ZENTRUM FÜR
INFORMATIONSMODELLIERUNG
AUSTRIAN CENTRE FOR
DIGITAL HUMANITIES

KARL-FRANZENS-UNIVERSITÄT GRAZ
UNIVERSITY OF GRAZ

Beyond TEI

Digital Editions with XPath & XSLT for the Web & in LaTeX

Sarah Lang

Harvard, April/May 2022

Overview

1. The workshop
2. XML transformations using XSLT

The workshop

Goals

1. get to know XPath & XSLT (and learn how to use it)
2. understand the role of XML/TEI, XPath and XSLT in Digital Editing
3. be able to use XSLT to generate HTML and LaTeX output from TEI
4. Two days isn't enough for you to master XSLT!

Schedule

Day 1, morning	XML, TEI and Digital Editing → repetition of the basics, making sure we're all on the same page, understanding why we're even learning XSLT.
Day 1, afternoon	Navigating XML documents using XPath, introduction to HTML (& Bootstrap) and LaTeX (& `reledmac`)
Day 2	Transforming XML documents into HTML & LaTeX output formats using XSLT

Single point of entry for all workshop-related materials: LaTeX Ninja blogpost & Github Repository (‘additional resources’ directory)

Introductions

Please introduce yourselves!

1. Name, pronouns, field/topic of study
2. Why did you come to this workshop?
3. Previous experience with Digital Humanities (DH) or editing?

Contact

@SarahALang_
@latex_ninja

sarahalang.com
latex-ninja.com

sarah.lang@uni-graz.at

Sarah Lang (she/they)

- originally from Germany, now in Graz (Austria)
- Studied Latin, French & History (teacher's education) in Graz & Montpellier (France), then Archaeology Bachelor, Master in Religious Studies & Philosophy
- got a DH certificate & started working at Zentrum für Informationsmodellierung (ZIM) / Centre for Information Modelling in Graz
  - Moral Weeklies/Spectators → *gams.uni-graz.at/mws*
  - Graz Repository of Ancient Fables (GRaF) → *gams.uni-graz.at/graf*
  - *PhD thesis:* Decoding alchemical *Decknamen* digitally. A Polysemantic Annotation and Machine Reasoning Algorithm for the Corpus of Iatrochymist Michael Maier (1568–1622)
- *Now:* teaching in Graz, Passau & Vienna; PostDoc in Graz. *Research interests:* history of science (alchemy), Neo-Latin, text mining and computer vision

XML transformations using XSLT

Finally: Beyond TEI!

Why are we doing this workshop? The motivation from our abstract:

- 👍 While it is an **ideal format for archiving digital data**, it is less than ideal for viewing and interacting with the edited text.
- 👍 The data transformation language XSLT allows editors to create multiple representations from their data encoded in XML, enabling the creation of both digital and print editions.

Goals for the next session

- 👍 Navigating XML using XPath
- ✖ What is XSLT?
- ✖ Setting up a transformation scenario
- ✖ Creating different representations from our data using XSLT
  - HTML (& Bootstrap)
  - LaTeX (& *reledmac*)

What is XSLT? i

XSL (*eXtensible Stylesheet Language*) is a programming language for transforming XML data. This allows for the storage of presentation-independent base data from which different representations can be generated (i.e. structure information for HTML).

This is called the *single source principle.*

XSLT (=XSL-Transformations) is often used as a synonym but actually is only one part like **XSL-FO** (*Formatting Objects* for describing print layouts in PDF; discontinued since 2012).

XSLT is made up of a set of processing rules (*templates*) according to which an XML document is traversed and processed. The rules are being applied whenever one matches the current node/content.

What is XSLT? ii

The parsing begins at the root node wich is why the first template is always this:

```
<xsl:template match="/">...</xsl:template>
```

Here, you can build your output document structure before the rest of the processing begins.

If no matching template is encountered, built-in rules are used (printing the text content of elements and their children. You will encounter this as content just being dumped into your output unformatted (usually, you only want the *<body>* printed and not all metadata of the *<teiHeader>* in unstructured form. Thus you need to explicitly suppress this built-in behaviour.

→ **In German:** IDE Kurzreferenzen ● **In English:** W3Schools tutorial

XSLT output formats

XSL(T): eXtensible Stylesheet Language (Transformations)

.XSL

An XSL stylesheet

is an XML document itself and describes rules for the transformation process of an input XML file (into one or multiple output formats).

Output formats are, for example, ...

- (x)HTML
- **XML** → e.g. Word, TEI and other XML standards, RDF, SVG
- **Text** → e.g. LaTeX (→ PDF), RTF, MARC.

Be careful! LaTeX is a lot like markup, too, but has different protected entities than XML and other conventions for 'escaping' them: LaTeX ampersand = `\&` versus XML = `&`
Use `xsl:output` and check for errors!

Getting started with XSL Transformations

Open an XML you want to transform & a new XSL stylesheet in Oxygen.

Notice that...

1. It's empty (the `<stylesheet>` has only attributes).
2. `<stylesheet>` has a namespace (`<xsl:stylesheet>`)

The auto-generated new `.xsl` document will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Tran
  xmlns:xs="http://www.w3.org/2001/XMLSchema
  exclude-result-prefixes="xs"
  version="2.0">

</xsl:stylesheet>
```

Namespace

Namespaces are defined in `xmlns` attributes:

- the link is the reference for the schema
- after `xmlns:` is the shorthand for the namespace (`xsl`)
- later we will address TEI as `t:TEI`
- this avoids mixups between the sometimes similar elements of HTML and TEI (`html:p` and `t:p`).
- what you call the namespace doesn't matter (`tei:` or `t:` both work) → shorter is better as long as it's not confusing

Configuring the transformation scenario

Click into the XML file & configure a transformation scenario (tool symbol):

1. Select ‘new’ (XSLT transformation).
2. **XSLT tab:**
   - 2.1 select XML file (probably already has the shorthand *`${currentFileURL}`*)
   - 2.2 select our new XSL file (needs to be saved)
   - 2.3 (you can do mass-transformations using the project options)
   - 2.4 select transformer: Saxon (any 9 version).
3. **XSL-FO tab:** Doesn’t concern us → to create PDFs in a manner alternative to LaTeX but not maintained since 2012
4. **Output tab:**
   - 4.1 ‘Save as’ → click green arrow, select *`${cfn}`* (‘current file name’). Then add *`-transform.xml`*, so our original file doesn’t get overwritten. (Whenever you transform, the transformation result file will be overwritten – avoid that by adding an ID / renaming it each time).
   - 4.2 Pick ‘Open in editor’ & show as XML (if XML or HTML if HTML). Also say ‘open as XML’ for LaTeX: ignore the complaints that ensue ☺
5. **Ok, then run it.** Now our – as of yet empty – new stylesheet (transformation) will be applied to the original file.

XSLT practice!

Set up your transformation scenario and run it.

What do you notice?

This is the standard behaviour of XSLT in the absence of matching processing rules (templates).

standard transformation scenarios i

With the last template, we indicated that we wanted the standard processing to be applied. This is just printing the text contents. → the resulting document isn't actually XML anymore! (it has no elements and no root)

Empty 'match root' template (<xsl:template match="/">)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <!-- What happens now? -->
    </xsl:template>

</xsl:stylesheet>
```

standard transformation scenarios ii

pushing the XML contents through the XSL tree using `apply-templates`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xsl:apply-templates/>
    </xsl:template>

</xsl:stylesheet>
```

standard transformation scenarios iii

‘Hard code’ structure into that root template

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xml>
            <xsl:apply-templates/>
        </xml>
    </xsl:template>

</xsl:stylesheet>
```

(i.e. type it in there)

Matching templates

Instead of laboriously hardcoding everything, we can also define templates which always become active whenever a certain element (that they match as indicated by the *@match* attribute) is encountered.

Read: Whenever you find *<p/>*, do XY.
We declare a 'namespace' (*xmlns*) called *t:* so our TEI data gets recognized as such:

```
xmlns:t="http://www.tei-c.org/ns/1.0"
```

When this line is present in the *<xsl:stylesheet>* root element, paths from our TEI file always need to have the prefix *t:*, e.g. *t:p*.

Read: *<p>* from the namespace with the shorthand *t:* as per the definition in the *<xsl:stylesheet>* root element.

processing/creating <p> elements in the output

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3
    xmlns:xs="http://www.w3.org/2001/XML
    xmlns:t="http://www.tei-c.org/ns/1.0
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xml>
            <xsl:apply-templates/>
        </xml>
    </xsl:template>

    <xsl:template match="t:p">
        <new>
            <xsl:apply-templates/>
        </new>
    </xsl:template>

</xsl:stylesheet>
```

XSLT processing paradigms i

All *p*, including their contents are preserved (thanks to *apply-templates*) but instead of *<p>* they are now called *<new>* (just for demonstration purposes).

This is referred to as the **push paradigm**.

You can see the **pull paradigm** where the *<head>* is added. We explicitly get the value of *<title>* to put in this element (careful about finding more titles than you wanted!

value-of select="" ≠ apply-templates

apply-templates will check for further processing rules deeper down the XSL hierarchy to match child elements of the current element.

value-of select="" only copies/prints the current element content (not the node!), further nested elements are lost!

value-of select="." gives you the current node's text content.

attribute={@rend} is a shorthand for attributes.

XSLT processing paradigms ii

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t="http://www.tei-c.org/ns/1.0"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xml>
            <head><xsl:value-of select="//t:title"/></head>
            <xsl:apply-templates/>
        </xml>
    </xsl:template>

    <xsl:template match="t:p">
        <new attribute={@rend}>
            <xsl:apply-templates/>
        </new>
    </xsl:template>

</xsl:stylesheet>
```

Deleting contents

XSLT automatically prints all element values (contents) but we can delete them. That way, we delete all elements, including their contents and child elements (!), from our output.
→ Don't accidentally write/save this on your original source data!
(Careful when setting up the transformation scenario to not name the output the same as the input XML.)

Delete consciously by creating an empty rule

```
<xsl:template match="t:hi">
    <!-- delete -->
</xsl:template>
```

This is mostly all you need to know.

Of course, there are more complicated functions for advanced usage. A few example are shown in the following.

XSLT example: poem to HTML page

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml" version="2.0">

    <xsl:template match="/">
        <html>
            <head>
                <title><xsl:value-of select="div/head" /></title>
            </head>
            <body><xsl:apply-templates select="div" /></body>
        </html>
    </xsl:template>

    <xsl:template match="div">
        <h1><xsl:value-of select="head" /></h1>
        <div><xsl:apply-templates select="lg" /></div>
    </xsl:template>

    <xsl:template match="lg">
        <p><xsl:apply-templates /></p>
    </xsl:template>

    <xsl:template match="l">
        <xsl:value-of select="." /><br />
    </xsl:template>
```

More functionalities i

variables

```
<xsl:variable name="substitute" select="content">
```

XSLT handles variable with a so-called *pass by value*, not *by reference*. Ergo only the value is passed that the element has at the moment the variable is created. If I define it too early, I might get the wrong value!

attributes

`<xsl:attribute>` is used to set attributes in the output document. It's not always necessary: Often you can use the shorthand *`type={@rend}`*.

More functionalities ii

plain text

<xsl:text> puts the text as it is, i.e. won't put in unwanted spaces (like XSL tends to do which can be a problem in LaTeX where space is significant. XSL will be very liberal with spacing, unless you explicitly suppress this using *preserve-space*.

sorting

There is *<xsl:sort>* for sorting values.

conditions

<xsl:choose> lets you choose a different way of processing according to a test using *<xsl:when>*. *<xsl:if>* only does something when a condition is met (i.e. if an element doesn't exist or has content XY). You should use this to check if optional elements exist to avoid errors!

More functionalities iii

Person list in HTML using `for-each`

```
<ul> <!-- unordered in HTML -->
<xsl:for-each select="t:persName">
    <li> <!-- list element -->
    <xsl:value-of select="." />
    </li>
</xsl:for-each>
</ul>
```

More functionalities iv

Setting attributes

```
<xsl:for-each select="t:persName">
    <span class={@rend}><xsl:value-of select="."></span>
</xsl:for-each>
<!-- OR -->

<span>
    <xsl:attribute name="id">
        <xsl:value-of select="@rend"><xsl:text>-person</xsl:text>
    <xsl:attribute>
    <xsl:attribute name="interpretation">
        <xsl:value-of select="concat(@rend,@ana)">
    <xsl:attribute>
    <xsl:apply-templates /> <!-- for the content -->
<span>
<!-- result e.g.
    <span id="monster-person" interpretation="monster-evil">
    Weird Sisters </span> -->
```

More functionalities v

For-each-group

```
<!-- could be sorted alphabetically.
     use sparingly, can produce mysterious errors. -->

<xsl:for-each-group select="bla"
  group-starting-with="dings[@rend="startbla]]"
    <xsl:apply-templates select="current-group()">
    <xsl:value-of select="current()/bla">
</xsl:for-each-group>
```

Merging multiple documents with (document( ))

```
<xsl:apply-templates
  select="document('Letter1_TEI.xml')/tei:TEI//tei:body/tei:div"/>
```

Loops: `for-each`

Example: Get each person (*//persName*) and generate a listing (*<ul>*)of the last names (*lastname*):

```
<ul>
<xsl:for-each select="//persName">
    <xsl:sort select="lastname" order="ascending" />
    <li> <xsl:value-of select="lastname"/> </li>
</xsl:for-each>
</ul>
```

Conditions I: if

xsl:if only runs if condition in *@test* evaluates as ‘true’:

```
<xsl:if test=" xpath-ausdruck "> ... </xsl:if>

<xsl:for-each select="//book">
    <xsl:if test=" author = 'Cicero' ">
        <li><xsl:value-of select="title"/></li>
    </xsl:if>
</xsl:for-each>
```

Conditions II: choose

You can also differentiate a number of cases:

```
<xsl:choose>
    <xsl:when test="some xpath"> ... </xsl:when>
    <xsl:otherwise> ... </xsl:otherwise>
</xsl:choose>
```

You can also sort (*xsl:sort*), copy (*xsl:copy - xsl:copy-of*) and use variables.

XSL paradigms: push

push paradigm

```
<xsl:apply-templates select="path"/>
```

Suitable for processing the text body (mostly).

Elements are processed whereever they are encountered, you don't need to know their exact structure or order. The input document's structure is preserved (or not if defined otherwise in the template rules). Useful if keeping a similar document structure is the intended goal.

```
<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>
```

push processing

Call Template / push method

```
<xsl:template match="/">
    <xsl:call-template name="etc">
</xsl:template>

<xsl:template name="etc">
    ... do sth ...
</xsl:template>
```

XSL paradigms: pull i

pull paradigm

```
<xsl:call-templates name="etc"/>
```

Useful mainly for extracting metadata from the *<teiHeader>* to display them somehow. Also allows you to go over the whole document and create a list of persons mentioned in it, for example at the beginning of the output, using *<xsl:for-each>*.

<xsl:for-each> and *<xsl:value-of select="">* are typical commands in the pull paradigm.

pull processing

Pick some specific nodes with full control. Useful if the output document's structure should differ from the input quite a bit or you just want to selectively keep certain elements (like just make a list of dates mentioned in a document without printing the text of the original document).

```
<xsl:value-of select=''pattern''/>
<xsl:apply-templates select=''pattern''/
<xsl:for-each select=''pattern''/>
```

The paradigms in practice

In practice, you will use a combination of both, maybe even for the same elements:

- **pull**, e.g.
  - get the metadata from the header to create a recommended citation
  - create a table of contents
- **push**, e.g.
  - get (select) and process the document body automatically
  - process chapter headings inside the document body

```
<xsl:apply templates
  select="//t:body" />

<xsl:apply-templates
  select="head" mode="toc"/>
```

`<xsl:template match="">`

process the element where it appears.

`<xsl:value-of select="">`

give me the element content as plain text/string (child elements and structure are lost).

`<xsl:apply-templates>`

traverses the rules and it applies the rules for anything found in a *`@match`*. You always have to push the *`apply-templates`*) in each single template rule.

Facts

How to create your stylesheet

top down, starting with the root of the source document. Regelwerk nach Top-down-Prinzip, ausgehend vom Wurzelelement des Quelldokuments.

root element `xsl:stylesheet`

XSLT is XML istself which is why it has an all-encompassing root element where lots of parameters can be declared (namespaces, XSL processor, etc.)

XSLT practice!

Start with setting up your transformation scenario (see slides, cheatsheet and video).

Do the XSLT (prep) exercises on the exercise sheet if you haven't already done so (2.1). Do exercises 2.2 and 2.3.

Set up the transformation to run `mini-bootstrap.xsl` and/or `mini-latex.xsl` (using the `dracula.xml`) (see materials/resources folder)

Try to run it on your own data (won't transfer properly), then adapt for your own data.

XSLT to HTML practice!

First, try to configure a transformation scenario and get this to work at all (ideally on the `dracula.xml` first). Then try the following:

Easy: Write simple template rules for a few elements (where do they belong in the hierarchy?), copy how it's done from the existing rules. Read the info on the worksheet!

Harder: Try to adapt the 'Show persons' toggle and template match for TEI `placeName` to 'Show places'.

If you're feeling adventurous (HTML):

First, try to configure a transformation scenario and get this to work at all (ideally on the `dracula.xml` first). Then try the following:

Have a look at the `mini-bootstrap-popover.xsl`. Try to understand what's going on.

The root template (`match="/"`) builds an HTML document and loads the Bootstrap framework. You can mostly ignore this for now. Start from the bottom where the `t:p` template is.

The template is quite a bit longer than the other mini example but not too complicated. It implements the popover wippet by Roman Bleier which allows users to show expanded/abbreviated versions for `<expan>` and `<abbr>`.

Try to add the infobox wippet (read the comments in the code) or look up other Bootstrap elements.

Can you make the template work on your own data?

XSLT to LaTeX practice!

First, try to configure a transformation scenario and get this to work at all (ideally on the `dracula.xml` first). Then try the following:

Easy: Write simple template rules for a few elements (where do they belong in the hierarchy?), copy how it's done from the existing rules.

Harder: Take a look at the next slide…

If you're feeling adventurous (LaTeX):

First, try to configure a transformation scenario and get this to work at all (ideally on the `dracula.xml` first). Then try the following:

Have a look at the `mini-latex-reledmac.xsl`. Try to understand what's going on & configure a transformation scenario to use the template on your own data.

The template is mostly a simplified version of the TEI Critical Apparatus Toolbox template but still complicated. Some templates are very long – leave them alone!

Remember: This is your first day using XSLT. The template might be too difficult for now. If so, use it for practicing after the workshop.