XPath and XSLT exercise sheet

Sarah Lang

May 2022

Use your own (or the provided Dracula) XML/TEI data for the following exercise.
The results are usually more interesting when using a slightly bigger base of data with lots of annotations (not just a toy XML example). Maybe consider using the Dracula data, at least for parts of the exercise. The suggested solutions will only match those (because the solutions obviously depend on the hierarchy of your data and thus, the provided solutions might not apply to you).

1. Open your data in the Oxygen XML editor (or any XML editor of your choice which supports XPath (read more here)).
2. Locate and use the XPath field for the XPath exercise.[1]
3. Create an `.xsl` stylesheet and set up a transformation scenario for it using the information on the slides (and/or cheatsheet) where there should be a step-by-step explanation.[2]
4. Don't continue past the XPath exercises until you've made this work correctly.

[1] In case there isn't an XPath field – though it should be there by default – go to 'window' → 'configure tools'. At the very bottom it should say 'XPath'. This field needs to be checked.

[2] The following video (in German) contains the setup of an XSL transformation: Einführung in XSLT, Teil 1: Transformationsszenario konfigurieren, Showcase Basisstylesheets. Maybe you still find the visuals helpful for setting up your document. I will add subtitles at some point in the future.

1 XPath Exercises

1.1 Query formal criteria

1. Count chapters or text sections (`<div>`).
2. Print headings (`<head>`).
3. Use `substring(), substring-before(), substring-after()` to print chapter headings (or anything else) based on the criterium that it contains a certain string of text.
4. Use `distinct-values()` to print all personal names (`<persName>`).
5. Print existing name variants (come up with a creative solution, depending on what's in your data) if applicable, maybe also using `distinct-values()`.
6. Try out local and global *scopes* (absolute and relative paths). What are advantages and disadvantages of each approach? In which case would you prefer one over the other?
7. Try out `matches()` and `contains()`: Mind you, `matches()` knows how to do regular expressions (regex) while `contains()` doesn't![3]
8. Give me the heading of the 4th part.
9. Personal name not containing 'peter'.
10. All persons and places appearing together in the same part (to be printed with just one XPath command).

1.2 Create statistics and query metadata (get creative, multiple solutions possible!)

1. How many people or places are there in your document?
2. Which people or places appear where in your document?
3. Who wrote the text?
4. Which information (metadata) can be queried from the document head (`<teiHeader>`).
5. Which people appear together in parts? Who 'never meets'?
6. Which people appear together with which places (if applicable)?
7. How many elements of textual criticism are there in your document?
8. How many pages does your document cover?

[3] Mabye try to regex-annotate all dates in your document. Read more in this blogpost: Automating XML annotation: Get more done using RegEx Search&Replace and `xsl:analyze-string` (or on the cheatsheet). There are also these two Youtube videos in German showing examples in Oxygen: TEI-Annotation in Oxygen XML: Von den Basics zur Automatisierung mit regulären Ausdrücken and TEI-Annotieren mit regulären Ausdrücken (Theorie-Einheit Regex).

2 XSLT

2.1 Preparing the transformation

1. Which attributes and elements appear? (research on the web or on the cheatsheet how to do this)
2. Which attribute values exist in your document? (And thus need to be covered in a data transformation later?)
3. Write things down or make list which ones you would like to cover in the transformation and how they should each be visualized (in laypeople's terms for now or you can already research Bootstrap options for a web view).
4. Find out how those elements are named in HTML (web-search or look on W3Schools) and how they can be displayed. (Mind you, HTML isn't organized in terms of semantic meaning of elements like TEI but only in terms of presentation).
5. Which additional information or features would be interesting for a user of my digital edition? Statistics? A metadata view of the `<sourceDesc>` or `<msDesc>`? The ability to highlight something?
6. **After the LaTeX presentation:** Answer these questions for LaTeX too. Which commands will you need? Find out how to do use `reledmac` here in this `reledmac` template on Overleaf. Think about which elements in your TEI need to become what structures in `reledmac`.[4]

2.2 Come up with XPath queries for the template rules

1. Write down the elements (and attributes) to be included: For which ones do I need the push or pull paradigm? (i.e. where in my stylesheet does the query need to go, in the 'head' or the 'body'?). Do I want to transform elements as they are encountered ('push them through the pipeline' or do I want to decide where they need to go and 'pull them out'?)
2. Do I need absolute or relative paths? (has something to do with the paradigm used)
3. Formulate what you are looking for in natural language (i.e. English): "I want all the mentions of person X which appear in the header."
4. Start translating those into XPath queries: Path, condition, function, etc.
5. Put the result into the XPath field of your editor. Does it work? If no, read the error message and think about how to set it right.

[4] There is a simple `reledmac` transformation stylesheet to help you out on Github → 'additional resources' directory.

2.3 Input the queries into your XSL document.

1. Once the query works, think about where to put it in your XSLT document. Don't forget that the success of this depends on whether you've used the right path (absolute or relative) and the correct name spaces. In the XSL, we need to call all TEI elements using the namespace we assigned, in our example `t:`, so each path needs to have that appended, like so:

   `//t:teiHeader/t:titleStmt/t:title`. When testing the XPath in the XPath field, you need to do it without that namespace or it won't work! (`//teiHeader/titleStmt/title`).

2. If you're not sure whether things are working correctly in the XSLT, start by writing 'test/debug messages', i.e. just writing 'test' as a result. That way you can check whether it appears and if yes where. (Is that where it was supposed to go? Was anyting found at all?).

3. Click 'run transformation' after each addition to make sure your stylesheet still works (you might have to save first). Mistakes are easier to locate if you find them early on. Search and localize errors ('bugs') in your code after every step to maintain control.

4. Once the queries work, start building up the HTML functionality.

5. *Bonus:* Once the basics work, come up with ideas for extra features. Look at digital editions you like for inspiration.

2.4 About our XSL example document (`mini-bootstrap.xsl`)

- **Document structure:** In the `mini-bootstrap.xsl`, the whole upper part (where we're building the HTML `<head>`) is just me loading the Bootstrap framework and some basics to speed up the setup.

  You should start at the bottom of the document where it says something like `<xsl:template match="t:p">`, i.e. a template for processing a TEI paragraph.

- **Where to place your own/new template rules:** Be careful when placing your new template rules – you need to respect the tree hierarchy of the source document. So if you want to add `<div>` into the document which already contains the `<p>`, this needs to go above it (outer part of the russian doll). If you want to add something 'smaller' like `<hi>` it needs to go afterwards. If elements are siblings, their hierarchy doesn't matter.

- **Also, never forget to continue passing on** `<xsl:apply-templates/>` because if you don't, the data will be printed as you stated but not be passed on to the next hierarchy/round of processing. Only leave it out for elements where you are sure that they don't contain child elements. Or always write it and XSLT will figure out for itself whether there's anything else left to be handeled. To check whether you might have forgotten child elements to take into account, try `//*[parent::div]` or `//*[parent::note]` (give me all elements which have a `<div>` or `<note>` as a parent element).

  Shift around your templates if you realise you didn't take some of those 'dependencies' into account or add `<xsl:apply-templates/>` as needed.

- **Tutorial video (in German):** For adapting the provided mini XSL files, you might want to watch the following video (in German): Einführung in XSLT, Teil 3: Eigene Templateregeln erstellen/anpassen.

3 Solutions

Solution ideas can be found on the slides as well as in the following section.
If you're totally lost, start by trying to find out what the questions are trying to get at (what could one want to query here?), then formulate in English, then try it out on your example data.
The solutions will work on the Dracula data but probably mostly not on your own data which has a different structure. Try with the Dracula data first if you're lost.

`substring()` won't work in the Oxygen view because this doesn't support string operations – but it will work in XSLT. If the command is correct, you can tell by the fact that you're not getting an error message. Also, you can only use string operations on exactly one string (not a number, i.e. set of strings). You can reformulate your XPaths to accommodate this.
Example:

- `contains(//t:origPlace/@ref, 'pleiades')`: Tests results in error if there is more than one `<origPlace>` element in a document because many string operations only allow a char-literal or one element, not a set of nodes as input. The error can appear even when there actually is only one element where the node test is true (only one `@ref='pleiades'`).
- Solution: `t:origPlace/@ref[contains(., 'pleiades')]`.

A path query returns a set of results/nodes. Predicates (i.e. conditions) are stated in [brackets]. Here, I can also put a function which is supposed to deliver an answer (i.e. a number or yes/no). Functions can also be put around the whole expression but then, you usually get just an answer (like yes/no), not a set of nodes as a result (which you probably want for further processing).
Think of it like this:

1. "Give me all `<persName>`s where it's true that it has an `@ref` which contains the value 'Dracula." → Answer: Here are your nodes.

   ```
   //persName[@ref[contains(., 'Dracula')]]
   ```

2. If you did this, you would only get the content of the `@ref` attributes (you probably want the content of the element it belongs to):

   ```
   //persName/@ref[contains(., 'Dracula')]
   ```

3. "Is there a `<persName>` which has `@ref='#Dracula'`?" → Answer: Yes. No nodes delivered. In this case, XPath will refuse to process because there is more than one node:

   ```
   contains(//persName/@ref, 'Dracula')
   ```

   Reformulate like so (string query is inside the brackets):

   ```
   //persName[contains(@ref, 'Dracula')]
   ```

To make sense of the slight difference in these commands, just try to reformulate them (or formulate them at first) in natural language. Then translate into XPath. Usually, there is more than one possible solution which will work for you.
For example (from the `mini-bootstrap.xsl`), you can reformulate

```
//t:placeName[ancestor::t:teiHeader][@xml:id = $normalized-place]
```

("Give me the place name that matches two conditions: its ancestor is the `<teiHeader>`, i.e. the `<placeName>` is part of the `<teiHeader>` and its `@xml:id` matches the one saved in the 'normalized-place' variable.)
into the computationally more efficient (and more common) version:

```
//t:teiHeader//t:placeName[@xml:id = $normalized-place]
```

However, if the above (more verbose) version is easier to understand for you, just use the first one. Once you graduate to working with big Digital Humanities projects, make sure to always use the latter.

Testing local and global scopes: Set your cursor into a `<div>` and then only put `p` as an XPath query: Now only nodes in the local context will be found (if there were any `<p>` elements as children elements in that `<div>`. `//` will always give you results from the global scope.

`not(//persname[@ref="#Mina"])` returns 'false' because XPath asks: Is there a `persname[@ref="#Mina"]` in the whole document? The answer is 'yes/true' (if that's the case).
Afterwards, it negates that answer with `not()`, i.e. it's not true that there is no `persname[@ref="#Mina"]` in the whole document.

4 Solution of the `dracula.xml` (depends on the data, of course)

4.1 Formal structure

```
1. //div[@type="chapter"]   -- count(//head)
2. //head -- //div[@type='chapter']/head
3. substring-after(//div[type=....],'Chapter')
4. distinct-values(//placeName/@xml:id)   OR TRY distinct-values(//placeName)
        --> what's the difference?
5. distinct-values(//persName[@ref="#Mina"]
7. //p[contains(.,'Christmas')]  OR TRY //p[matches(.,[A-Z][a-z])]
8. //div[@type="chapter"][@n="4"]/head  OR TRY  //div[@type="chapter"][4]/head
9. //persName[@ref != "#Mina"]  OR TRY \\persname[not(@ref='#Mina')]
10. //div[@type='chapter'][1]/persName | //div[@type='chapter'][1]/placeName
ODER //person | //place
```

4.2 Statistics

```
1. //div[@type="letter"]//placeName
2. Variants of the above with placeName and persName
3. //author OR TRY //titleStmt/author
4. e.g. //teiHeader/fileDesc/titleStmt/....
5. and 6. combined queries from above
//persName | //placeName (maybe add more precise path)
```

We could construct the union of Mina and Dracula with a complex query like so:
`//div[descendant::persName[@ref='#Dracula']]//persName[@ref ='#Mina']`, ergo "Give me all the `<div>`s who have `<persName>` descendants (i.e. not necessarily just child elements but all sorts of descendants) with the `@ref` attribute with value '#Dracula'. And from those `<div>`s, give me the Mina `<persName>`s.
Or just do:

```
//p[persName[@ref="#Mina"] and placeName[@ref="#London"]]
```

4.3 Preparing XSLT

See info box on the cheatsheet (has all the solutions).

4.4 Some more general info

1. If titles with a `<head>` element in the title are queried, only part of the title is shown but this should be just a display problem in Oxygen. Should work in the XSLT.

2. If you want to use OR, you need to repeat the whole XPath, not just `/div[@n="1"]//(placeName|persName)`. Correct solution:

   ```
   //div[@n="1"]//placeName | //div[@n="1"]//persName
   ```

3. Broadly speaking, you need a global scope whenever your cursor isn't currently in the area you're looking at (*local scope*, i.e. you're currnetly in a `<div>` element and what you're looking for is a child of this `<div>`). This is mostly the case for the header. In this case, it's probably safer to go for an absolute path, also because there could be ambiguous elements (like `<author>` elements in a `<bibl>` but you want the author from the `<titleStmt>` or the `<sourceDesc>` specifically). Such mixups can only be avoided by using absolute paths for those elements where you're just looking for one concrete thing, not trying to process all elements of the type `<hi>`, for instance. This is usually done using `<xsl:value-of select=".">` rather than an `<xsl:template match=".....">` in the 'XSL body' (and templates which aren't inside `<xsl:template match="/">`, i.e. match root).