Editing the Medieval Text

Course Handbook

P. S. Langeslag

revision of December 3, 2024

Contents

Software

For the purpose of this course, I recommend you set up Visual Studio Code ('VS Code' or 'Code') and install the XML extension by Red Hat. You are free to substitute your coding editor of choice if you prefer, whether or not it handles XSLT transformation, but all instructions and demonstrations in the context of this course will assume you are working with VS Code.

Installation and Setup

Look for Visual Studio Code in your operating system's package manager ('app store') or user repository, or else find it at https://code.visualstudio.com.

Once you have it installed, launch Code and open the extensions manager (`Ctrl+Shift+X`) to install the `XML` extension by Red Hat. You may additionally want to install `tei-publisher-vscode` by e-editiones. That last extension is really intended as a bridge between Code and the standalone TEI Publisher app, but it offers a transformation preview even if you don't have TEI Publisher installed, so it turns Code into more of a full-fledged XML editor.

Default Keybindings

VS Code provides a great many keybindings. So many, in fact, that it has to rely on sequences of key presses ('chords'). Here you can typically keep your modifier key (i.e. `Ctrl` or `Command`) down while you press a sequence of letters, styled in documentation as e.g. `Ctrl+K Ctrl+O`, or sometimes simply `Ctrl+K O`. The present manual refers to PC shortcuts; their macOS counterparts typically substitute `Command` for `Ctrl`. See further Custom Keybindings as well as Entities below.

Project Management

VS Code relies on the concept of the workspace, which in most cases simply maps the contents of a single folder (though it can handle multiple) to your graphical interface and keeps a metadata record. One way to create a new workspace is simply to create a folder for your project, then open it in Code (`Ctrl+K O`). In our case, however, we will take a public repository as our starting point, so there will be no need to identify a location just yet. Just keep in mind that whenever you initialize a workspace, you will be prompted to register whether you trust the authors of the files in your folder. Not trusting imported workspaces can be a useful security precaution, but it disables some extensions, including `tei-publisher-vscode`, so keep that in mind as you determine your trust levels. This course doesn't involve any files that should have you worried.

Accessing Course Files

If you do not already have Git installed and set up, the easiest way to get at the course files is to visit https://github.com/langeslag/editing, click the green button that reads < > Code, and select Download ZIP. Extract the archive to a new folder in a location of your choice. Finally, in VS Code select File > Open Folder… and locate your new folder.

If you have Git already installed and configured, you can access the course repository directly from within VS Code. First close any other project folders you may already have open (`Ctrl+K F`), then navigate to the Source Control window (`Ctrl+Shift+G`). When

prompted for a repository URL, copy in `https://github.com/langeslag/editing`. Once the remote folder has been cloned, you will be prompted to open the new workspace and given a choice of whether to trust its contents.

File Templates

VS Code lacks a native approach to file templates. However, there are many extensions available. If you like, you can install the `File Templates` extension by Bruno Paz. It relies on the presence of templates in your `User/FileTemplates` folder, so you could copy our template into this directory. This unlocks a `New File from template` function in the Command Palette. However, this doesn't save you much work compared to manually duplicating `template.xml`; just make sure not to make any changes to the template file itself if you go the latter route. Whichever approach you choose, you should now be able to load your new file and pick up at XML File Structure and the TEI Header below.

Custom Keybindings

If you regularly transcribe documents that contain special characters, you may want to register keybindings for them. (Refer to this section only for common unicode characters that display well in fixed-width fonts; for other use cases see Entities below.) As noted above, VS Code has a lot of keybindings already registered, so you will either have to look for unused combinations or override keybindings for functions you don't expect to use. On a US keyboard, `Alt+ ` ` ('backtick') is a conveniently available initial 'base note' for a chord, which you can combine with letter keys for the characters you need.

Custom keybindings are stored in a JSON file. JSON is a widely used data storage format. You'll get used to it by using it; for the time being you just need to be able to reproduce structures like those printed below.

To access the keybindings file, open the Command Palette (`Ctrl+Shift+P`, or `F1`) and start typing `Preferences: Open Keyboard Shortcuts (JSON)`, and select that option once it comes up. This opens `keybindings.json` in the editor. For the time being, the file is almost empty:

```
// Place your key bindings in this file to override the defaults
[]
```

What you'll want to do now is enter the desired keybindings in between the square brackets as follows:

```
[
    {
        "key": "alt+` w",
        "command": "type",
        "args": {
            "text": "p"
        }
    },
    {
        "key": "alt+` s",
        "command": "type",
        "args": {
```

```
            "text": "ſ"
        }
    },
    {
        "key": "alt+shift+` s",
        "command": "type",
        "args": {
            "text": "&slongdes;"
        }
    },
]
```

The keybindings should work as soon as you save the file. The validator will tell you if you've got anything wrong, e.g. if you've forgotten a bracket, comma, or quotation mark. For more on VS Code keybindings, see the documentation.

Please note: macOS has a lot of keybindings hardcoded into the operating system that will override anything you might have defined yourself. This makes the process of identifying suitable keyboard shortcuts considerably harder on macOS. The same is true to a lesser extent of most non-English keyboard layouts across all operating systems.

XML File Structure and the TEI Header

Upon opening the template, you should be looking at the contents of a 46-line XML file. You will find that the entire file is comprised of a series of **nodes** that are instances of specific **elements**, i.e. metadata containers, which are expressed in pairs of **tags** set apart by angular brackets, such as **<title>**, **<editor>**, or **<text>**. All valid elements for our particular framework are defined elsewhere, in the schema, and described in the TEI Guidelines, but some of the ones you should expect to work with actively are covered below, with reference to our specific use cases and the assumptions inherent in the transformation stylesheets we'll use to process our transcriptions.

The logical structure of XML has it that every instance of an element must be *closed*: thus to write a prose paragraph you start with the ** ** opening tag, followed by the paragraph content, whose end is signalled by the ** ** closing tag; the forward slash indicates that the element is being closed. However, an opening tag can also be closed immediately without the need for a separate closing tag. This is typically done where no content is required: when we use **<pb/>** for 'page beginning', for instance, the node is '**empty**' and has no text content but serves only as metadata indicating that a new manuscript side starts here (it is a **milestone element**), and so there is only a single tag, with the forward slash that closes it occuring immediately before the closing bracket of the sole or opening tag. In these cases, predefined kinds of information may still be contained in the form of **attributes**: **<pb** `n="18v"`**/>** marks the start of the verso of the 18th folio. In either form, the node must be closed with the help of a forward slash.

XML documents accommodate **nesting**: the content of one element, say **<library>** (in a hypothetical situation not covered by our schema), can include further nodes using such elements as **<bookCase>**, which in turn may contain lower structural levels such as **<book>**. It is, however, an unforgiving structure: if you close the containing element (the bookcase) before closing the contained element (the book), your document is invalid and will yield no output when you attempt to transform it into another format, such as PDF or HTML.

Likewise, the schema regulates how elements may nest; a sensible schema would prohibit for bookcases to fit inside bookcases, or for bookcases to fit inside books. Our schema forbids for the lower-level element `<s>` (sentence) to contain a higher-level element `<p>` (paragraph), for instance. In the process of drafting XML, you will invariably end up spending time investigating why your code will not validate, or why the output of your transformation is not as expected, but your editor's validator and sometimes the feedback received from the transformation processor will provide the necessary guidance at least with the first of these challenges.

Mention was made above of **attributes**, which are optional or mandatory features contained within the opening (or sole) tag of an element and set off from the element name by a space. (Attributes are identified in the documentation using a preceding at-symbol, e.g. `@type`. While the `@` is in fact used in languages used to *navigate* and *process* XML, it is printed in XML documentation only to avoid confusion with elements; the `@` is not to be used in XML code!) Each attribute in turn has a **value**, which is supplied in (conventionally double, but always straight) quotes and set off from the attribute name by an equals sign. Any further attribute is again set off by a space:

```
<head xml:lang="la" rend="color(red)">DOMINICA PALMARUM</head>
```

Although technically each attribute must have exactly one value, we can use a custom **delimiter** to allow us to encode more than one piece of information, which we may separate out when processing the file by relying on that same delimiter. We will use a single space as a delimiter between attribute substrings. Thus if the rubric from the above example is two lines tall in addition to being in red ink, its `@rend` attribute could read `rend="color(red) size(2)"`. Once we have settled on the space as a delimiter, we should ban the use of space characters in attribute values where the material to their left and right is not intended as two separate values; thus we cannot use `@reason="faded ink"` if we are also relying on space to delimit multiple substring values for `@reason`.

XML element and attribute names are **case sensitive**. This means you have to correctly and consistently reproduce the typographical case of the class you mean to reference. In the above example, the element `<bookCase>` uses a convention known as **camelcase**, using a capital only for the initial of each non-initial word element in the name. When working with a schema that registers an element `<bookCase>` but not `<bookcase>`, the latter form will not validate.

You will have noticed in the template that the lines are indented at different distances, like so:

```
<library id="Bodley">
  <bookCase name="Junius">
    <book n="11"/>
  </bookCase>
</library>
```

This is an optional but practical way of visualizing the nested structure of the elements: for every new child element, it is best to start a new line and indent it one additional unit. Coding editors will indent your lines for you as you work, and many have explicit formatting commands as well. VS Code has the keybindings `Shift+Alt+F` (Windows), `Shift+Option+F` (macOS) and `Ctrl+Shift+I` (Linux). Or if you forget, open the Command Palette with `Ctrl+Shift+P` and start typing `Indent using tabs`.

The header, which is what we call everything between `<teiHeader>` and `</teiHeader>`

(ll. 9–31 of the template), contains metadata about the file, yourself, and the text and manuscripts covered in your transcription. In our template, action is required where a line starts with a complete XML tag but ends with instructions contained within `<!-- -->` tags, as these instructions specify what information to enter between the opening and closing tags. These special brackets signal a **comment**: anything contained within them is ignored by the processor and intended only for the benefit of those editing the raw markup. Code editors will usually render comments in a distinct colour so they are easily told apart.

Understanding Tools and Files

You will be most immediately engaging with a single filetype: the XML file. This is where you will transcribe your texts, and it is what you will be submitting to the transformation processor. However, our full file infrastructure is a little more complex. There is, first of all, the **schema**, which is a document declaring what XML elements are valid and what attributes they may, or must, have. For our convenience, we will be using a remote (i.e. online) schema covering all the elements defined by TEI. The header of your XML file makes reference to this schema (l. 7). This serves two purposes: it informs anyone accessing the file what standard we are using, but it also serves as the benchmark for the **validation** process carried out by your editor to test your syntax against the schema.

In addition to the schema, we need a **(transformation) stylesheet**, i.e. a document declaring what an HTML (or DOCX, ODT, LaTeX) file generated from your XML document should look like: it specifies not only traditional design features such as the base typeface and text colour, but also what formatting action any given XML element triggers, for instance that a rubric should be rendered in a certain font size, weight, and colour, that foreign text should appear in italics, and whether something encoded as appearing in red ink in the manuscript should in fact be reproduced in red. XML files may be styled in CSS, in which case the XML file is loaded directly in a browser but styled using information contained in the CSS stylesheet, and no new file is written to disk. However, for more complex cases we may need to generate new files in a format intended for reading or further processing through a process called **transformation** using an **XSLT transformation processor**. We will experiment with the CSS and XSLT solutions in turn, but because we will be transforming to PDF via LaTeX, which requires additional software, we will rely on a custom transformation script set up on our course server rather than work with editor extensions or standalone local setups. Thus after a first few experiments with CSS, you will upload your XML files to https://langeslag.uni-goettingen.de/editing and, if all is well, receive a PDF file for download in return. Both the schema and the stylesheet can be stored locally, but for simplicity's sake we are referencing remote copies instead.

Populating the TEI Header

Returning to your new document, you will see that its larger part is taken up by the TEI header (ll. 9–31). The logical structure of the TEI header is explained in the course video on the topic; for our practical work, we'll limit ourselves to entering the necessary details into this basic header template. So go ahead and enter a title into the **`<title>`** field (i.e. in between the **`<title>`** and **`</title>`** tags); enter your name into the **`<editor>`** field; and populate the **`<listWit>`** field with information on your manuscripts: one **`<witness>`** element per witness, with the desired siglum as a value of `@xml:id` and the full call number as the element content. To add a second witness, simply copy out the whole block from

<witness> up to and including </witness> and paste it underneath the existing block. Make sure to change at least the value of @xml:id for the new witness or your document will not validate. You can enter random content if you are just trying this out at this stage. Just keep in mind that the information is to be entered in between the opening and closing tags of each element, not after the latter, and not into the comment.

Starting Your Transcription

Next, scroll down below the template header. Our template accounts for different scenarios: it expects you will be transcribing prose, verse, or both. The transformation stylesheets we will be using build on those of the Digital Latin Library, which work with strictly defined text divisions encoded using the element . Our template provides one for prose and one for verse; thus if you expect to encode verse only, you'll want to delete the three lines comprising the prose division (<div type="textpart" …), and vice versa if you won't be needing the verse environment. Then place the cursor after <p n="1"> or <l n="1">, respectively. This is the starting point of your transcription; only if you have a rubric (i.e. a manuscript heading) to input, add a <head> element in the preceding line and enter it there.

NOTE
TEI does not require that we number our paragraphs, but the LaTeX package we will use to typeset the textual apparatus relies on it, and therefore so do the DLL stylesheets we have adapted for our purposes. Thus we will contain all prose within numbered paragraphs, and verse in numbered lines, just to be able to export a critical edition to PDF.

Prose and Verse

It may be difficult to impose paragraph breaks onto a text written by someone else in a language you don't fully understand. For our purposes, it is acceptable to keep a prose transcription within a single paragraph, or alternatively to break it into multiple paragraphs arbitrarily. Just remember to number each new paragraph incrementally to meet the DLL's requirements.

Sequences of verse are not expected to occur within a <p> element. Instead, in addition to enclosing each *line* of verse within an <l> element, we wrap the full verse *sequence* in <lg> for 'line group'. Though not strictly required, it is best practice to number verse lines explicitly (<l n="1">). The <lg> (line group) element likewise is not mandatory, but it is recommended whenever two or more lines of verse form a sequence. Finally, although the DLL handbook mandates that elements be given attributes like @type, @n, and @xml:id, your document will transform fine with bare elements, as long as the top-level division reads . Remember that attributes are not repeated on closing tags, so indentation is useful for telling nested elements apart:

```
<div type="edition">
  <div>
    <p n="1">The <title level="m">Aeneid</title> opens as follows:</p>
  </div>

  <div>
```

```
    <lg>
      <l n="1">Arma virumque cano, Troiae qui primus ab orus</l>
      <l n="2">italiam fato profugus Laviniaque venit</l>
      <l n="3">litora, . . .</l>
    </lg>
  </div>
</div>
```

For your first attempt at a transcription, start with something simple and verify whether it (validates and) transforms correctly. Create a few numbered paragraphs contained in a `<div>`, copy in some random prose and see if it will transform (see Styling and Transformation below); then add an `<lg>` environment inside another `<div>` with a few lines of verse and try again. Once you've ensured everything works as expected, you can start encoding your homework assignments.

Validation

While you'll have to get used to understanding what you see in the raw XML, if you are using an XML editor there are a couple of tools at your disposal to help make sure all is well. The first of these is validation, which holds your code against the schema containing the rule set (in our case for the TEI brand of XML) and points out any mismatches. Every advanced XML editor comes with a validation service, and our template links to an online schema file (l. 8). You'll generally rely on *automatic* validation, which draws Word-like squiggly lines under invalid code as you edit your document. For instance, if in your new document you create a new line within the `<p n="1">` element and type `<head>` (to encode a manuscript rubric), it should only take a second or two before underlining appears. In VS Code, upon seeing the squiggly line appear you can retrieve the full text of the error either by opening the Problems panel (`Ctrl+Shift+M`) or with the debugger by hitting `F8`. Rubrics are not allowed inside paragraph elements; thus any text-initial rubrics should precede the first `<p>` in the text body. VS Code shows the location of any errors in red in the scrollbar on the right. Live validation thus helps you maintain valid code as you type. If any issues are still outstanding when you are done transcribing, you may choose to identify and resolve any outstanding issues by initiating the formal validation process or navigating through VS Code's debugger, or you can simply click on or navigate to each of the red positions in the scrollbar and solve them using the cues provided by automatic validation.

Entities

Keybindings inserting special characters directly as glyphs, as recommended above, are only advised for standard unicode glyphs that are ideally contained in the character sets of leading fixed-width typefaces as well. The reason we want them to be standard unicode, as opposed to 'Private Use Area' characters, is that otherwise the wrong choice of font won't render them at all, nor will it be clear what glyph they are meant to represent. The reason we want them represented in popular fixed-width fonts is that code editors conventionally use fixed-width fonts, and if these can't display your special characters they will come out as **tofus** (□) and you won't be able to identify their intended value or tell them apart on sight.

In all other cases, you will need to encode your special characters as **entities**. This means

you declare and describe them at the top of your XML file, assign them a user-readable identifier, and also provide the corresponding (hexadecimal) identifier of its slot within the unicode character set. You will see that our template `template.xml` has a few such declarations right at the top, contained within the doctype declaration:

```
<!DOCTYPE TEI [
<!-- TIRONIAN SIGN ET -->       <!ENTITY et "&#x204A;">
<!-- COMBINING MACRON -->       <!ENTITY combmacr "&#x0304;">
<!-- LATIN SMALL LETTER LONG S DESCENDING -->       <!ENTITY slongdes "&#xF127;">
]>
```

For each of three special characters, these lines document the three aspects mentioned above in the same order. You can add further characters by copying in the relevant lines from the Menota entities list. Now to use these special characters in your document, you need to take the second piece of information (in the above example, that's `et`, `combmacr`, or `slongdes` respectively) and enter it delimited by an ampersand on the left and a semicolon on the right, e.g. `&et;`. In fact, as soon as you type an ampersand in an XML coding editor such as VS Code with the XML extension, a dropdown menu appears with all the available options, i.e. all the entities declared within the document. If your validator should ever complain about undeclared entities, just make sure you have copied in the correct entity declarations and don't mistype them anywhere, or use any entities that haven't been declared.

Just two final notes on the general use of entities. First, whereas we use them here to gain access to special characters, their broader purpose is to be able to map longer or more complicated sequences to short references, so you could e.g. conveniently produce the string 'North Atlantic Treaty Organization' by typing simply `&NATO;`. Second, entities, like XML element and attribute names, are case sensitive: if you've declared an entity as `NATO`, you cannot invoke it with `&nato;` but must type `&NATO;`.

Styling and Transformation

To see the result of your work, you will either need to reference a CSS stylesheet before opening your XML file in a browser, or you can transform your document into a format meant for reading or further processing. Let's take these in turn.

CSS (Cascading Style Sheets)

By default, if you open your XML file in a browser (opening it as a local file as you would in any other app, e.g. with the `Ctrl+O` keybinding), it is likely to warn you that it has no style information associated with it. Odds are it will output its contents nonetheless, but in document-tree format, i.e. much the same as in your code editor. If we now declare a stylesheet, it will look for style information you have provided in your CSS file. For anything not styled there it will fall back on the browser's defaults (this is the 'cascade' referred to in the format's name). This means that if you declare a stylesheet that doesn't exist, your browser will simply print the content of your XML file, including the full header, in a default style.

Stylesheet declarations go near the top of your document, ahead of `<TEI>`. So we can append a line like the following e.g. after line 7 of the template:

```
<?xml-stylesheet type="text/css" href="tei.css"?>
```

Now if you reload your browser window, it will print the entirety of the file, including its header, with the browser's default style settings. Note, however, that it prints the entire content as a single line, with no particular formatting. This is because XML uses custom elements, which the browser defaults can't possibly cover, and it is unlikely to have specifications even for a comparatively widespread standard like TEI. So we'll need to create the referenced stylesheet `tei.css` (e.g. `Ctrl+N S tei.css` in VS Code) and populate it with style instructions.

Though it has become remarkably powerful over the years, CSS is not an advanced scripting language. We can only tell it how to display each type of element, or not to display it. So let's start by filtering out some of the header information using the instruction `display: none`, starting each major unit of text on a new line using `display: block`, and styling the title with the properties `font-size` and `font-weight`. Copy the following code into your `tei.css`:

```
title {
    font-size: 400%;
    font-weight: bold;
}

publicationStmt, sourceDesc {
    display: none;
}

editor, text, div, p {
    display: block;
}

text, div[type=textpart], div[type=poem] {
    margin: 1em 0;
}
```

CSS is not designed to add content (as opposed to style) that is not in the document. We can get around this in minor ways by using pseudoclasses, for instance by adding this instruction:

```
editor:before {
    content: 'edited by ';
}
```

But the bottom line to remember is that CSS is there to style content that is already there, and at most to disable some types of content from being displayed. Anything beyond that requires XSLT.

For more on CSS, I recommend the online tutorials by W3Schools and especially web.dev.

XSLT (Extensible Stylesheet Language Transformations)

Whereas CSS merely provides style information for documents and does not reshape them in any major way, XSLT can take your XML document and apply a more complex set of transformation rules to each element it encounters, or even mine it selectively for the information it wants and output that in an order and structure it prescribes. As such, its capabilities place it somewhere in between a stylesheet and an interpreted programming

language, but by official definition it too is a stylesheet language. XLST processing outputs a new file to disk, which may be anything from plaintext to DOCX or (indirectly) PDF, but perhaps the most common output formats are HTML and XHTML.

Whereas the commercially available editor Oxygen ships with TEI stylesheets for transformation to XHTML, PDF, DOCX, and ODT, VS Code does not offer transformation out of the box. The extension `tei-publisher-vscode` enables transformation with a narrow range of stylesheets, though they won't quite work as intended unless you have TEI Publisher (and `eXist-db`) installed. You may nevertheless wish to transform to HTML or XHTML as you work in order to ensure your work renders as intended. In VS Code with `tei-publisher-vscode`, `Ctrl+Shift+A` will give you a predetermined list of stylesheets to choose from (after retrieving it online); if you select one, it will render a transformation preview in a new pane. Please note that the stylesheets provided by TEI Publisher do not account for many of our elements, so the results will not be as intended e.g. if you encode abbreviations or textual variants. For your critical editions, you will instead work with an online PDF transformation utility set up specifically for our coursework. Head to https://langeslag.uni-goettingen.de/editing and submit your document to the transformation tool. Once you submit, you will be presented with at least a few lines of feedback on the process and the validity of the file. Ignore warnings, but closely study any errors. If transformation is successful, you will see the PDF output if you click the PDF button. If not, you will receive a 'file not found' error and you will have to identify and resolve the problem within your XML file so you can try again.

Metadata

You will transcribe manuscript text simply by entering it as content of the **<body>** element. The kinds and level of detail of the metadata you will be encoding will differ from one homework assignment to the next. Because our main objective will be to produce normalized editions with a textual apparatus, the elements that form the critical apparatus are the most important; but this handbook also documents a handful of basic elements to do with appearance, abbreviations, scribal interventions, and editorial emendations.

The Textual Apparatus

Our template comes with a TEI header preconfigured to produce a textual apparatus encoded using the parallel-segmentation strategy of linking the apparatus to the transcription itself. This means that you will transcribe your text as if you were inputting plaintext until you run into a word that differs between two witnesses, at which point you will produce an entry for the critical apparatus by opening an **<app>** environment, within which you will enter all the variant readings in one **<rdg>** element per reading. There may be multiple manuscripts sharing the same reading; these are all encoded in one and the same instance of **<rdg>**. For the reading you want to use in the edited text, the element **<lem>** is often used instead of **<rdg>**, leaving **<rdg>** for *variant* readings only, and our DLL-derived stylesheets in fact require this approach.

In order that you may refer to witnesses at all, you will have to populate the **<listWit>** environment in the TEI header with one entry for each manuscript witness collated. Each witness requires a siglum; a programmatic identifier (often identical to the siglum); and the manuscript's full call number. In our case we'll furthermore want to add a folio reference.

The siglum can be assigned at your discretion, though for some texts there is an established

set of sigla. The manuscript call number consists of its location (i.e. city or town), the library that houses it, and the full shelf mark. We'll encode this information in child elements to each <witness> element along the following lines:

```
<witness xml:id="Xd">
  <abbr type="siglum">X<hi rend="superscript">d</hi></abbr>
  <name>Cambridge, Corpus Christi College, MS 201</name>
  <locus from="85" to="87"/>
</witness>
```

As you can see, the siglum in this case differs from the XML identifier inasmuch as it is formatted, the second character being encoded as superscript so it will appear as X^{d}. (Instead of the <hi> node you can alternatively use a superscript unicode glyph.) <locus> is here used as an empty element with attributes @from and @to indicating the first and last folio or page of the text here edited. Our stylesheets have been set up to recognize from the values of @from and @to whether it concerns pages or folios, and whether the text is contained on a single page or spread across several.

Whether to include the explicit specifier 'MS' for 'manuscript' in the call number depends: it may be omitted in many cases, but it is sometimes required to make clear that it concerns a handwritten manuscript rather than a printing. Furthermore, some libraries use custom identifiers either to specify the material (e.g. 'perg.' for parchment) or to reflect a language tradition other than English, using either the Latin abbreviation or that used in the library's official language, so that some manuscripts are referred to as 'cod.', 'Hs.,' or similar. You'll simply want to reproduce what you find in the facsimile used, and perhaps do a little online investigating.

Within the text body, a typical apparatus entry will look somewhat like the following:

```
This bit is regular text; it is the same across all witnesses.
But now we come to a word that <app>
        <lem wit="#A #C #D #E">differs</lem>
        <rdg wit="#B">defers</rdg>
        <rdg wit="#F">defunds</rdg>
    </app> between the witnesses.
```

When a word is omitted from one or more witnesses, use an empty <rdg/> element to indicate this:

```
a <app>
    <lem wit="#X">green</lem>
    <rdg wit="#Y"/>
  </app> bush
```

If a word is not present in the *preferred* reading, you cannot encode an empty <lem> element, as this would leave the apparatus with no way of informing readers *where* other witnesses add a word. The solution in these cases is to include a neighbouring word in the reading rather than use an empty element:

```
<app>
    <lem wit="#V #H #U #b">Wart</lem>
    <rdg wit="#S">Doe wart</rdg>
</app>
```

With this problem as with others, you may want to try different solutions and see how they transform.

Please note the following:

- TEI gives editors a choice between encoding each reading using **<rdg>**, so that the document does not explicitly endorse any one reading as preferable, or explicitly indicating which reading is to be used in the edited text by encoding it with **<lem>** (for *lemma*, here meaning the sequence as it should appear in the edited text), as in the example. However, the DLL stylesheets require a **<lem>** entry, so that's the approach you'll want to take in order to be able to transform using our transformation tool.
- We refer back to the witnesses identified in the TEI header using the programmatic identifiers with which they are there associated, with one important difference: in the apparatus we prepend a hash or pound symbol **#** to the identifier to indicate it refers to an identifier defined elsewhere in the document.
- Multiple witnesses sharing the same reading are entered as space-separated values of the attribute `@wit`.
- For ease of reading the source code, you'll want to print each reading on its own line of code.

Also keep in mind the following principles of critical editing:

- It is conventional to keep lemmata short, to a single word where possible.
- Decide how detailed you need the apparatus to be before you start recording it. It often suffices to record variant readings only if they differ in meaning or substantial points of grammar, and to ignore mere orthographical variants. It is also possible to record all variation but distinguish between significant and minor variation using the **<rdgGrp>** element as explained in the TEI Guidelines.

To ensure you've understood this section, populate **<listWit>** in the header with a handful of fictional witnesses, then add an instance of fictional variation to your draft. Submit it for PDF transformation and ensure everything looks as intended.

Notes and Witness Detail

A quick glance at any textual apparatus makes clear that there is more to the apparatus than sigla and readings: there is also frequent need to offer further detail, such as when an earlier reading has been modified by a historical corrector. TEI offers the element **<witDetail>** for this sort of information. This element has the mandatory attribute `@wit` to associate it with the witness whose reading is being annotated, and if the transformation processor is to associate it with the correct reading it furthermore needs an attribute `@target` pointing to an `@xml:id` associated with the appropriate **<lem>** or **<rdg>** element. Any readings or other manuscript content quoted in your note had best be wrapped in **<mentioned>**, so decisions about their presentation (e.g. italics) can be made in the stylesheet. A simple annotation may then look as follows:

```
arcu <app>
  <lem wit="#A #B" xml:id="scelerisque2.4">scelerisque</lem>
    <witDetail wit="#B" target="#scelerisque2.4">corr. from
     <mentioned>scelerumque</mentioned></witDetail>.
</app> sit amet
```

If you wish to add a note to the apparatus for a given entry, but not to offer details on any one witness, you can enclose a <note> element within the <app> environment, i.e. as a sibling to the <lem> and <rdg> elements. While TEI allows this element to be encoded anywhere between readings and recommends the use of @target to clarify what reading it goes with, the current state of our DLL-derived stylesheets will only render your note if it appears at the end of the <app> entry without @target.

Emendation

TEI's core module has a way of flagging and emending scribal errors by enclosing them in a <choice> environment, with the attested form as the content of <sic> and your emendation as the content of <corr>:

```
sceolde <choice>
 <sic>beran</sic>
 <corr>beon</corr>
</choice> ærest gode gehalgod
```

This information is not, however, automatically incorporated into the critical apparatus. In a critical edition, therefore, the superior solution is to offer the correct reading as <lem> but associate no witnesses with it if no manuscript has it correctly. The incorrect reading may then be encoded as <rdg>:

```
sceolde <app>
  <lem>beon</lem>
  <rdg wit="#A">beran</rdg>
</app> ærest gode gehalgod
```

Abbreviations

If we choose to encode abbreviations, we will make use of the <choice> element to encode two parallel readings: once as it occurs in the manuscript and once resolved into its implicit full form. The elements to be used in parallel are <abbr> and <expan>. Within the latter of these, it is best to indicate which letters you are supplying for the expansion by wrapping them in <ex>: you then have the option of printing supplied content in italics, for instance. For completion's sake, we can tag the abbreviation marker <am>, though only where it can be isolated from the letters themselves. An abbreviation *sc̄s* for 'sanctus' may then be encoded as follows, with the entity code &combmacr; representing the macron over *c*:

```
<choice>
  <abbr>sc<am>&combmacr;</am>s</abbr>
  <expan>s<ex>an</ex>c<ex>tu</ex>s</expan>
</choice>
```

In more straightforward cases, where the abbreviation marker and the resolvable material are immediately adjacent, TEI allows us to simplify by bypassing the outer pair of <abbr> and <expan> and placing the inner pair of <am> and <ex> immediately within <choice>:

```
mannu<choice>
        <am>&combmacr;</am>
        <ex>m</ex>
     </choice>
```

Punctuation

Medieval punctuation differs greatly from modern punctuation. Depending on your manuscript, you may encounter mostly simple puncti (·) or a greater variety of marks. Because we will mostly be producing normalized or semidiplomatic editions, you may ignore these and insert modern punctuation where you see fit, or just leave out punctuation altogether. If you do have a use case for medieval punctuation, you will often find that the unicode character set does not account for them. Fortunately, the Medieval Unicode Font Initiative has defined a larger set, and we can use their custom entity codes as documented here. The three most common marks are additionally on our list of special characters.