**1.) Gettting stuff out of the XML files:**

I’ve only taught myself how to use the node.js SAX XML parser (I have to use a SAX parser rather than a DOM parser because the files are far too big to construct a whole DOM-tree out of them in memory; I tried it with supr.xml and it just slowly ate all my RAM without achieving anything), and I’m most familiar with using SQLite directly via its C/C++ interface, so I can’t go straight from XML -> SQLite without first extracting the data I need in a .csv format with node.js, and then running my C++ SQLite scripts on the resulting .csv file. (SQLite is not exactly the same as MySQL; as it says on their website it’s a “replacement for fopen()”, by which they mean it’s really just a more efficient way of accessing on-disk data: sqlite databases are just single files; you don’t need to have a database-server process running in the background like you do with MySQL. Doubtless this has implications for how it performs, but I’ve never noticed anything.)

My node.js xml-parser goes through all the .xml files in the directory, checks if the ‘language’ attribute of the <source> tag is equal to the specified language-tag (either ‘chu’ or ‘orv’), and then just extracts from each <token> tag the necessary information, as long as the token is marked as annotated[[1]](#footnote-0) and isn’t one of your empty ghost-tokens which you need to make your syntax-trees work. I’m not very pleased with this “presentation-before” and “presentation-after” strategy for specifying punctuation and spaces etc., but for now I have no choice but to just take them as they are into 2 separate columns of my own database.

The new lemma\_ids are generated during the extraction by just having a Map() object that uses the POS+OCS-lemma combined-string as its key, and a counter-number as its value, such that newly-encountered POS+lemma combos can be added to the Map() and are assigned a new lemma\_id, but for ones which have already been seen and added to the Map() we just retrieve their existing lemma\_id numbers. The C++ script used to get the data into an SQLite table re-maps my old original lemma\_ids onto these generated new lemma\_ids each time it runs, so changes in the amount of lemmas used in the base XML files shouldn’t affect anything as long as the database is re-generated.

I also keep track of both the single main title of each text (which I define as the one whose parent-tag is <source>) and the sub-titles (which I define as ones whose parent tag is <div>). The citation-part attribute used to specify Book-Verse-Chapter information for the gospel-texts is too annoying to deal with at the moment so the gospels will only be separated by chapter. The sentence numbers are taken as well because whatever display-solution I choose will need to be paginated based on number of full sentences per page (and obviously sentences are pretty much the base-units from your syntactical point of view anyway).

**2.) Getting stuff into an SQLite database**

Here I’m trying to start integrating your data with the LCS reconstructions and inflection-class-marking which are needed for the autoreconstructor: as input I take:

a .csv version of my lemma master-spreadsheet, with the original old lemma\_ids

the .csv file of every tagged chu word + data produced by node.js in the first step

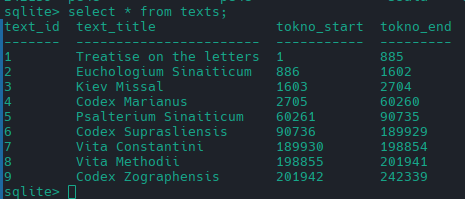
a .csv file of the chu lemmas extracted by node.js (not strictly necessary)

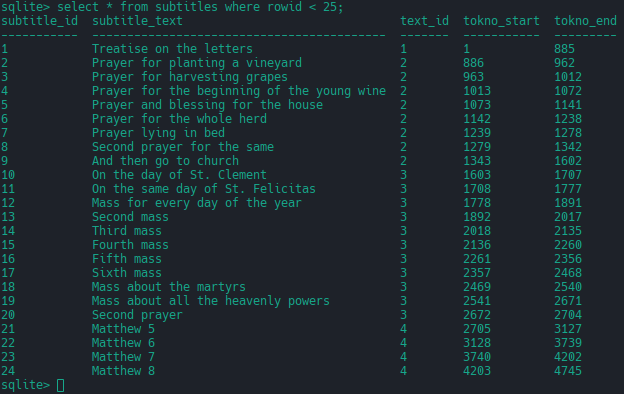
I read through the chu\_words.csv file produced by node.js and extract from each line the values to be inserted into each row of a table currently called “tagged\_corpus”:

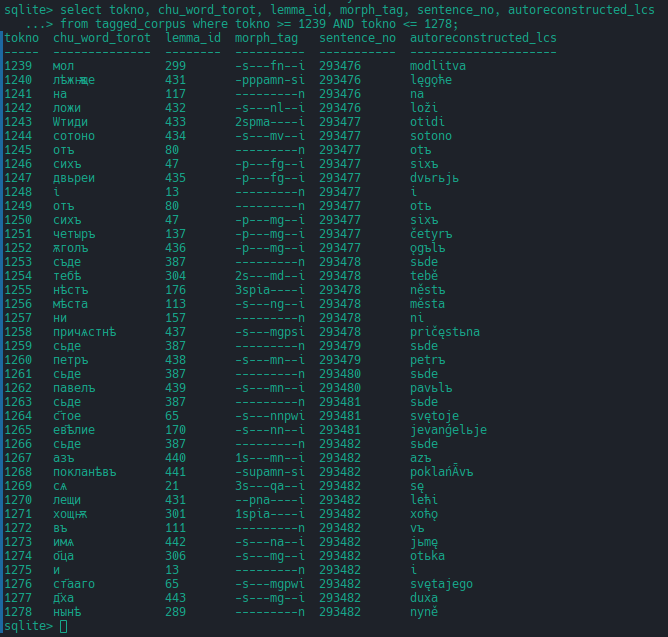
`INSERT INTO tagged\_corpus (chu\_word\_torot, chu\_word\_normalised, morph\_tag, lemma\_id, sentence\_no, presentation\_before, presentation\_after, autoreconstructed\_lcs) VALUES (?, ?, ?, ?, ?, ?, ?, ?)`

I don’t need the POS-tags because they are tied to lemmas stored in the separate lemmas-table and thus retrievable from the lemma\_id. I’ve also found a very quick and dirty way to integrate my autoreconstructor into this script, so the autoreconstructed\_lcs field is filled by taking the chu\_word\_torot string, the lemma\_id and the morph\_tag and passing it to a function which will just reconstruct single-words, rather than whole texts. If a particular word’s lemma isn’t included in the ~3400 which the Autoreconstructor can handle, then this function returns an empty string and I put NULL into the database using sqlite3\_bind\_null().

From the title and subtitle indexes extracted from the XML files I get offsets for the start and end of each text and of each subdivision within each text; these are stored in a “texts” and a “subtitles” table:







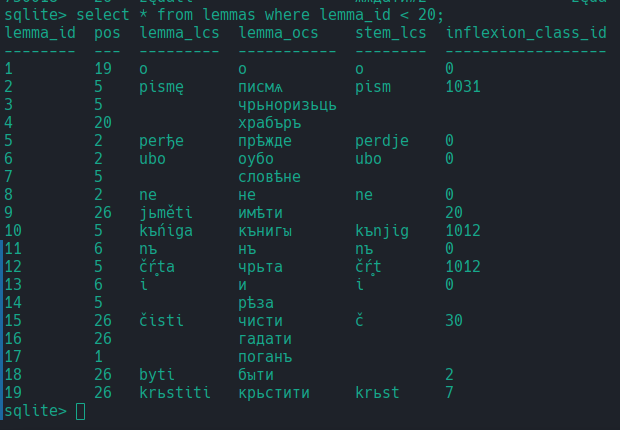
Obviously this is all completely basic and the absolute bare minimum for having any kind of text-corpus, but recreating it all from nothing but XML files (for me at least) is not an entirely trivial task.

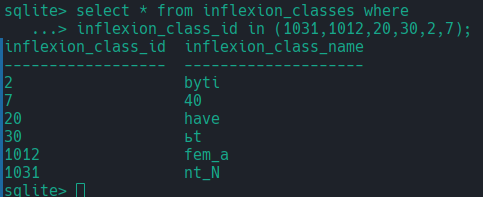
Building the lemmas-table is what requires my master lemmas-spreadsheet, because I want to add my inflection-class labels and lcs-stems to it so that full paradigms of lemmatised words in the texts can be generated from the web-interface (by getting the server to stick the endings onto the LCS stem of the lemma of the clicked-on word, and then run it through my LCS-to-OCS converter (or eventually dialect-specific converters, e.g. a Suprasliensis-dialect generator that has a tendency to harden \*ŕ, or a Marianus-generator that often vocalises or messes up the jers), and then send the results back to the browser).

The lemmas-file which I generate with node.js from the XML files (containing the new lemma\_ids) is converted into a std::unordered\_map object in memory (with the POS+lemma combo as the keys again), and then I look up each combined OCS lemma-form and POS-label from my master-spreadsheet in this map, and if it matches then I take the corresponding newly-generated lemma\_id, otherwise (and this only occurs with lemmas which you have got rid of since 2020, or with the additional lemmas I had to add to cope with your failure to distinguish certain verbs due to their infinitives being homoforms) I use the original lemma\_id from my spreadsheet.

The original very silly way I produce from the data in the master-spreadsheet the LCS stems which the Autoreconstructor adds endings to has been just wholesale copied over (it requires truncating non-ASCII strings so I had to use the ICU library; the C++ std::string is just a container for a null-terminated array of bytes). Non-reconstructed lemmas just get NULL as the value for their lemma\_lcs, stem\_lcs, and inflexion\_class\_id fields in the lemmas-table, but they will still have a POS and an OCS form, so I will be able to recreate what syntacticus does in producing a paradigm of attested forms for non-reconstructed lemmas, I just won’t be able to generate full LCS-based paradigms for them.

An inflexion\_classes table is produced to map the text-string inflection-classes used in my spreadsheet onto numbers (starting from 1 for verbs and 1000 for nouns), but this probably should be manually constructed so I can easily group together related inflectional paradigms by adjacent numerical ranges, so that someone can search not just for one narrow class like neuter-NT stems, but all neuters, or all consonantal-stems, or all class 1 verbs, etc. (Though I could of course just manually program these wider-classes into whatever piece of Javascript will be responsible for making the request, e.g. just have hard-coded arrays of numbers containing all the inflexion\_class\_ids that make up whichever broader category it is.)





**3.) Possibilities for displaying this in an interface**

Currently I am trying to repurpose the system used for my modern language-learning application as the basis for displaying these texts, because it already has a lot of useful graphical-assets for interfaces and tooltips etc., and the underlying functionality is very similar to what I need here (i.e. display the words of a text differently based on the data in adjacent fields of the database; allow different possibilities for interacting with each word based on how much annotation it has in the database, e.g. for fully lemmatised and autoreconstructed words I can generate the whole paradigm, or highlight other words on that page with the same inflection-class, or generate and display the “correct” form of a word which has been flagged as a morphological-innovation by the autoreconstructor).

Modern browsers are actually extremely powerful in what they allow you to do: you could for instance mark every word of the text for which an LCS reconstruction exists with a certain class in its HTML, e.g. <span class=”lcs\_exists”>word</span>, and simultaneously with the HTML you could get the server to send a JSON array of each lcs-reconstruction which exists on that page., e.g.

const lcs\_forms = [“slyšǞvъ”, “že”, “junošǞ”, “slovo”, “otide”, “skr̥bę”, …].

The length of the lcs\_forms array would equal the length of the array you’d get by doing `document.getElementsByClassName(“lcs\_exists”)`, so you could have a some on-page searchbox for finding the reflexes of LCS-phonemes just on that page, which would work by taking whatever the user inputs into the searchbox:

let search\_term = “šǞ”;

const matching\_indexes = new Array();

for(let i = 0; i < lcs\_forms.length; i++) {

if(lcs\_forms[i].includes(search\_term) matching\_indexes.push(i);

}

const reconstructed\_words = document.getElementsByClassName(“lcs\_exists”);

for(const index of matching\_indexes) {

reconstructed\_words[index].classList.add(“pulsate”);

}

And then the “pulsate” class would be listed in a CSS stylesheet with some CSS animations that would make every word found by the search-box burn orange every 1 or 2 or however many seconds, i.e.

.pulsate {

animation: burn\_orange 1s infinite;

}

@keyframes burn\_orange {

50% { color: orange; }

100% { color: inherit; }

}

Then to switch off the pulsating you could just do .classList.remove(“pulsate”) on the same list of elements.

This is just one very rough example of what kind of possibilities there are for displaying and interacting with the information contained in the database; the main point is that browsers can do a lot more than you might think with not very much effort, as long as one has complete control over the front-end and has not straight-jacketed oneself with some rigid framework.

**4. Stealing the OCS dictionary from the Czechs**

As I have already mentioned, it’s possible to programmatically make requests to the gorazd.org website where the digitgised version of the OCS dictionaries are hosted. The exact URL is "http://castor.gorazd.org:8080/gorazd/show\_record\_id;jsessionid=CC2E5311CE705F8E1F7780DD9095BF4F?value="+record\_id+"&xslFile=0&fields=&\_=1732327903110", where the bit I’ve highlighted is just an integer that you can increment on each new request to get the next entry. (Maybe the jsessionid shown there will have expired by now but you can very easily get a new version of this URL from the browser. As a test I stole about 2000 entries back-to-back and they never blocked me or registered me as suspcious.)

The smaller dictionary that just has words from canonical OCS in it is the only one we really need, and I think those records start from record\_id number 18080, but the dictionary-type is included in the response-JSON anyway so I just ignore everything where this field doesn’t == 2.

Extracting the main headword and the first English (or other modern-language) definition is very easy; what is harder is to get subsequent definitions, because there is not a guaranteed 1-1 correspondence between OCS-text and modern-language definitions, and you can’t just take all the English definitions and list them under the main headword, because often the English definition will only refer to some modified version of the headword, e.g. the entry for дѣло has a secondary English definition “idle” which is actually the definition of “без дѣла”. However nothing in the HTML allows me to distinguish between this secondary misleading English-definition and other secondary, but more proper definitions, such as “work, occupation, performance, action”, and not all of the secondary English definitions have a corresponding secondary OCS-entry, so it’s quite difficult even to correctly index each scraped English-definition with (if such exists) its corresponding OCS-entry.

I could of course just take the headwords and then take the entry as an undifferentiated mass of HTML (and steal their style-sheets to make it display in the same way), but that’s going to be too much data to get the whole dictionary into one JSON object, whereas if I could just take the English definitions I think it’d be small enough to load the entire thing into memory client-side and have look-ups be instant (an example of this is the Mitchell & Robinson Old English glossary which I stole from the .epub version of the book: even with a duplicated Old-English column with the diacritics etc. taken out to ease searching, the total size was only 464Kb, so I can just load the whole thing into the browser as a JSON object on page-load).

1. Really I should’ve taken unannotated sentences as well, and made it so the eventual database would just have NULL values for the lemma\_id and morph\_tag fields for unannotated words. That way I’d be able to display everything, but just alter the appearance / functionality of completely unannotated words (e.g. grey them out). It would also mean I could more easily play around with auto-tagging and auto-lemmatisation strategies using only the database, rather than relying on scattered .csv and text-files, which is what my current auto-tagging routines have to use. [↑](#footnote-ref-0)