Text Analysis of Classical Authors with ConTeXt

From Text to XML to PDF

Thomas A. Schmitz

In my classes, I often provide lists of vocabulary for the (Greek or Latin) authors that we are studying. I use ConTeXt not only to typeset these lists, but also as a tool for extracting them from pure text files. The ConTeXt distribution contains a number of helpers that make it more convenient to use than pure Lua.

Introduction

1. The Premise

The basic premise for my work is the simple fact that literary texts often use a vocabulary that students of a language find difficult to master. This is true in any language; for ancient Greek and Latin, the problem is exacerbated by the fact that students cannot acquire their vocabulary by immersion or fun activities like watching films or listening to music: they have to learn words the hard way. And there will always be specialized, rare words that students probably will not know (and shouldn't be required to learn because of their rarity). This means that the reading process of literary and poetical texts is slowed down: students have to look up too many words in the dictionary, and reading becomes an arduous task.

One way to help would be to have a list of rare words with their translations so students don't have to look each word up individually. Moreover, if such a list is produced the right way, it can help students prepare for their reading by identifying words that are relatively common by a given author. Producing such lists is my aim here, and I will be using ConTeXt to extract the words from a text file, `textfile.txt`. These files come from an online corpus. They are machine readable and thus have a predictable format, which makes them easy to process. In order to facilitate reading this article, we will be using a Latin author as an example, but most of my work is on Greek authors, and this will be relevant for some steps of the workflow. Here's a few lines of the poetry of the Latin author Catullus (first century BCE):

```
.85B.t.                {LXXXV}
.85B.1t       Odi et amo. quare id faciam, fortasse requiris.
.85B.2t           nescio, sed fieri sentio et excrucior.
.86B.t.                {LXXXVI}
.86B.1t       Quintia formosast multis; mihi candida, longa,
.86B.2t           rectast. haec ego sic singula confiteor;
```

```
.86B.3t       totum illud 'formosa' nego: nam nulla venustas,
.86B.4t           nulla in tam magnost corpore mica salis.
.86B.5t       Lesbia formosast, quae cum pulcherrima totast,
.86B.6t           tum omnibus una omnes surripuit Veneres.
```

2. Steps of Processing

Our task is to extract the “rare” words from this text file and present them in a manner that is useful to students. This involves two steps:

1. Reducing the Latin words to their standard grammatical form and filtering out words that are not relevant. Bringing this information into a format that is easy to process with ConTEXt but can also be used with other applications, should this be necessary. In our case, this format will be XML.
2. Typesetting the information in the XML file so that it becomes useful to students. This step will involve more than the mere converting from XML to PDF, as we will soon see.

Each of these steps brings its own set of challenges. Not all of them have clear and consistent solutions – natural language processing is sometimes messy, especially when it has to deal with highly inflected languages such as Latin and Greek. And we will also see that some aspects of the process cannot be automated; they rely on the understanding of a human reader competent in these languages.

Parsing, Extracting, and Filtering Words

3. The Structure of Our Table

The first challenge we have is thinking about conserving and converting the structural information that is already present in our text file. If you look at the second line, you will see the following items:

1. After an initial period (which is just the way the files divide parts of books and poems), we see the number of the poem in Catullus’ book. Here, it’s counted as “85B.” The letter “B” is an unfortunate artifact which may or not be meaningful; we have to conserve it just in case it is important.
2. After another period, we see the line number within the poem; again, followed by a letter, in this case, “t.” Again, this is an artifact of the way in which these text files were produced; here, we know that this letter is meaningless.
3. After the numbering, there is a tabstop and the Latin text of the line.

Lines 1 and 4 are merely headers for the following poems; we want to make sure that they are not processed.

Since ConTEXt includes not only LuaTEX, but also a number of useful additions (such as **lpeg**), we will use it for parsing our text file. We will produce a file `analyze.lua` and run it with the command `mtxrun --script analyze.lua textfile.txt`. In the first lines of this file, we make sure that the extensions provided by ConTEXt are indeed available and that Lua opens the necessary files for reading and writing:

```
require("char-ini")
require("l-lpeg")
in_file  = io.open(arg[1],'r')
out_file = io.open('parsed.xml', 'w')
```

Our script will read from the file that is given as an argument and write to a file named `parsed.xml`.

The next step is writing an input parser that will make sure only lines containing Latin text will be processed. We use **lpeg** to do this parsing. Since we know what the structure of these lines looks like, we can build an **lpeg** that will test whether every single line conforms to this structure and extract the relevant information. To save ourselves some typing, we introduce a number of handy abbreviations:

```
local P, R, S, Ct, C = lpeg.P, lpeg.R, lpeg.S, lpeg.Ct, lpeg.C
local period         = P"."
local digit          = R"09"
local poem_letter    = R"AZ"
local line_letter    = P"t"
local tabstop        = P"\t"
local end_of_line    = P"\n"
local rest_of_line   = (1 - end_of_line)
local entire_line    = period + C(digit^1 * poem_letter^0) *
period * C(digit^1) * line_letter * tabstop * C(rest_of_line^1)
```

I will not go into the details of the **lpeg** syntax here. Very briefly: you can build patterns from simple elements. We define the elements of which our line consists: periods, digits, letters, a tabstop, and then everything from this tabstop to the end of the line. If we add a `C` to our **lpeg** pattern, it will *capture* its match. The last line of the example thus defines what a line should look like and captures the relevant parts.

We can now read our input file `in_file` line-by-line and check whether each line conforms to our standard:

```
for line in in_file:lines() do
  if entire_line:match(line) then
    poem, line, all_words = entire_line:match(line)
  end
end
```

If the line conforms, the three matches that we have defined will be assigned to three variables, **poem**, **line**, and **all_words**. We will then assign our words to a Lua table that has the poems and lines as keys. So our table will (in a simpliflied form) look like this:

```
table = {
  ["85B"] = {
    ["1"] = { word1, word2, ... },
    ["2"] = { word1, word2, ... }
    }
  ["86B"] = {
    ["1"] = { word1, word2, ... }
  }
}
```

One thing I have learned over the years is a characteristic (you could also call it a shortcoming) of Lua: this table is what is known as an associative array and the poems and the line numbers are the keys that have values (which are in themselves again tables). When you want Lua to traverse (iterate through) such an associative array, you typically use this type of code:

```
for k, v in pairs (table) do
  process (k) process (v)
end
```

The problem here is that Lua will traverse the keys **k** in random order. Which means that in a poem, you may very well see the lines ordered in this way `1, 10, 2 ...` This is not what we want. One option would be to sort the poems and lines after we have built our tables. However, the reality of numbering poems and lines (and paragraphs and books) in ancient texts is messy. In our case, it is certainly possible to make Lua understand that poem 85 should come after 84 and before 86B. But lines (or, in prose texts, paragraphs) may contain Greek or Latin characters, commas, even other symbols. So I found myself coding exceptions to exceptions for the sorting, which were always liable to mess up my order (or make my Lua file fail). When I asked for help on the list, Hans cooked up a solution: he created a Lua structure called `table.orderedhash` that will allow you to loop through its keys in the order in which they were added. This is how we populate and loop through such an ordered table:

```
section_table = table.orderedhash()

for poem, poem_table in table.ordered (section_table) do
  process (poem, poem_table)
end
```

This way even bizarre numbering will not be a problem; the poems and lines are preserved in the order in which they appear in the text.

4. Splitting the Text into Words

Our next step will be to split the text (which we have captured in the string variable **all_words**) into words. First, we want to get rid of all the parts of the text that are *not* words, i.e., punctuation marks, special editorial symbols, parentheses, etc. For this, we can again use a helper that ConTEXt offers, an **lpeg.replacer**. It takes a Lua table as input and replaces the first item in every pair with its second item (which in our case will be the empty string, because we just want to remove them). Here's what the code would look like for a few punctuation marks:

```
local punctuations = lpeg.replacer {
  { "." , "" },
  { "?" , "" },
  { "," , "" },
  { "'" , "" },
}

function punctuation_replacer(s)
  return lpeg.match(punctuations, s)
end

all_words = punctuation_replacer(all_words)
```

Next, we split this text into words, which we capture in a Lua table. The helper function **lpeg.checkedsplit** is provided by ConTEXt; it has the advantage that it takes care of empty strings, which may occur if there is a whitespace before the end-of-line marker:

```
local capture_table = lpeg.checkedsplit(" ", all_words)
```

For every line in our `textfile.txt`, we now have the words in a Lua array **capture_table**. Now we need to do something with these words!

5. Reducing the Words to their Dictionary Forms

As I have mentioned, Latin is a highly inflected language. Words take extremely different forms, depending on their declension and conjugation. One huge problem is that the same form can be derived from a number of different words. As an example, look at the form *cane*. It can be derived from four different words:

ablative singular	*canis*	“dog”
imperative singular	*canere*	“sing”
imperative singular	*canēre*	“be gray”
vocative singular masculine	*canus*	“gray”

In some (rare) cases, even professional scholars may have doubts about what a particular word means (and of course, Latin writers may have played with these ambiguities). For a reader who is fluent in Latin, however, most cases will be clear, according to the semantic and syntactic context. Maybe one day in the not too distant future, computers will be able to take the context into consideration when determining the dictionary form of words; scholars in the digital humanities are working on a number of solutions. For the time being, however, all we have are simple lists that give one (or possibly several) dictionary form(s) for every word. Such lists can be found, e. g., at the Perseus Project or the Classical Language Toolkit. There is a gray area where our tool will never be perfect: if we look at the possibilities for *cane*, we can see that most occurrences will be derived either from *canere* or from *canis*. If we exclude *canus* and *canēre*, however, we may miss the one really exceptional passage. And if we require intervention everytime that a form is ambiguous, parsing would take extremely long. There is thus no ideal solution; in the end, a human reader will have to look at these lists and double-check.

I loaded the parsing lists that I got from the Classical Language toolkit into a Lua table `forms`. It has around 350 000 entries, which look like this:

```
forms = {
  ["cane"] =  "canis" ,
  ["canem"] =  "canis" ,
  ["canen"] =  "canis" ,
  ["canes"] =  "canis" ,
  ["canesque"] =  "canis" ,
}
```

This allows me to reduce any word to its dictionary form. In the case where a word is not found in this table, I want it to be visible so I append a special marker to this word. Here is what the code looks like:

```
for _, word in ipairs (capture_table) do
  if forms [word] then
    word = forms [word]
  else
    word = word .. "XXX"
  end
end
```

6. Filtering Words

Our next step is to filter these words: we want to exclude all proper names (e. g., in line 86B.1, the name *Quintia*). The folks at the Classical Language Toolkit have an (incomplete) list of Latin proper names against which I check my words:

```
if not names_table [word] then
  process (word)
end
```

And we want to exclude the most common words. In Germany, many students of Latin (at least at the university level) use the *Lateinischer Grund- und Aufbauwortschatz*, which contains around 2 700 of the most common Latin words. I have compiled these into a Lua list and filtered it again: words that are neither proper names nor in this list of common words will be entered into our Lua table. One little trick that I have learned along the way: the first idea would be to have these common words as a simple Lua array:

```
grundwortschatz = { "ab", "abesse", "abire" }
```

However, checking whether a given word is in such an array is tedious. Hence, we assign these words to a table by using them as keys and assigning them a default value:

```
grundwortschatz = {
  ["ab"] = true,
  ["abesse"] = true,
  ["abire"] = true
}
```

This way, it is easier and much faster to check if a word is in the list:

```
if not names_table [word] then
  if not grundwortschatz [word] then
    table.insert(table [poem] [line], word)
  end
end
```

After applying all these operations to our two poems, here’s what the Lua table containing the words looks like:

```
table = {
   ["85B"]={
    ["1"]={},
    ["2"]={ "excruciare" },
   },
   ["86B"]={
    ["1"]={ "formosus" },
    ["2"]={},
    ["3"]={ "formosus" },
    ["4"]={ "micare", "salum" },
    ["5"]={ "formosus" },
    ["6"]={ "surripere" },
   },
}
```

If you’re fluent in Latin, you will see that both derivations for l. 86.4 are, in fact, wrong: *mica* here is not imperative of the verb *micare* “glitter,” but rather nominative singular of the noun *mica* “crumb.” *salis* is not dative/ablative plural of *salum* “high sea," but rather genitive singular of *sal* "salt" (a word that is in the *Grundwortschatz* and would need to be excluded). This is something that has to be corrected in the proofreading stage of these lists. Moreover, you see that the lists for l. 85.1 and 86.2 are empty: these lines contain only proper names and common words. This will be important when we now write out these lists to a file.

7. Creating an XML File

In theory, we could now write these Lua tables to a file and start the typesetting process from there. However, I prefer to write them out to an XML file, for several reasons: XML is easier to read for a human than the Lua tables and, as a universal format, it can be repurposed for other uses than typesetting. Once we have our Lua tables in place, we can simply iterate through them and write everything to our `out_file`:

```
for poem, poem_table in table.ordered (section_table) do
  out_file:write('  <section number="')
  out_file:write(poem)
  out_file:write('">\n')
end
```

Within this loop, we process the single lines of every poem. Here, we are careful to include lines only if they contain a word list:

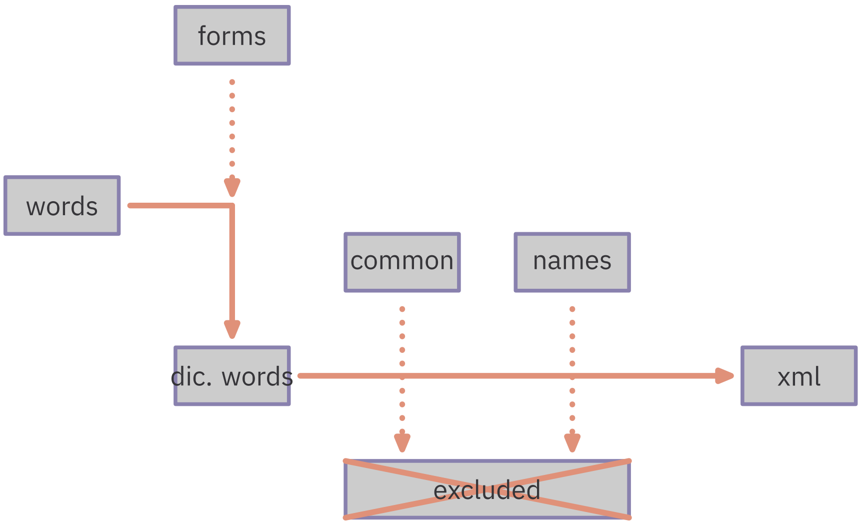
```
for line, line_table in table.ordered (poem_table) do
  if #line_table > 0 then
    out_file:write('      <subsection label="')
    out_file:write(line)
    out_file:write('">\n')
  end
end
```

Then, we write the words to the XML file by looping through the array **`line_table`**. In the end, this is what this file will look like (after applying the necessary corrections):

```
<author name="Catullus">
  <section label="85">
    <subsection label="2">
      <note type="voc">
       <word>excruciare</word>
      </note>
    </subsection>
  </section>
  <section label="86">
    <subsection label="1">
      <note type="voc">
       <word>formosus, a, um</word>
      </note>
    </subsection>
    <subsection label="3">
      <note type="voc">
        <word>formosus, a, um</word>
      </note>
    </subsection>
    <subsection label="4">
      <note type="voc">
        <word>mica, ae, f</word>
      </note>
    </subsection>
    <subsection label="5">
      <note type="voc">
        <word>formosus, a, um</word>
      </note>
    </subsection>
    <subsection label="6">
      <note type="voc">
        <word>surripere</word>
      </note>
    </subsection>
  </section>
</author>
```

8. Final Thoughts

In the preceding sections, I have shown how to use ConTEXt to parse and analyze a text file. We could summarize this workflow in a graphic:



While the convenience of the many **lpeg** helpers (defined in `l-lpeg.lua`) is certainly nice to have, I have neglected one area where ConTEXt really shines, compared to regular Lua: many string operations in Lua presuppose 8bit-wide characters and fail as soon as your text is utf8. This is usually not a problem for Latin, but of course it's a huge problem for Greek. Chapter 10.6 of the document `cld-mkiv.pdf` (which is part of the distribution) explains many functions and their use.

When I first began this work, I looked at several scripting languages: Perl, which was more or less created for this purpose, Python 3, which offers a number of powerful structures, and finally Lua. With ConTEXt's enhancement, Lua is by far the best fit when it comes to processing utf8 input. And it is a lot (really a lot) faster than the other languages.

I use the XML files created in this process not only to produce the vocabulary lists, but also to keep my personal notes on the classical texts that I read and study in classes; this is why it is necessary to differentiate the notes pertaining to vocabulary by giving them an attribute `voc`. This will be important in the typesetting process.

Analyzing and Typesetting

9. Formats

Now that we have the XML file, we can think about the output we want to produce. In its current stage, my typeset lists have three sections:

Nach Abschnitten **85, 2–91, 5**

86

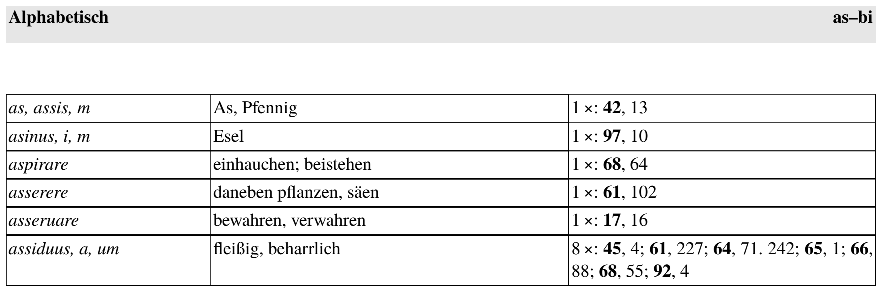
1	*formosus, a, um*	wohlgebildet, schön	3 ×
3	*formosus, a, um*	wohlgebildet, schön	3 ×
4	*mica, ae, f*	Krümchen, Bisschen	1 ×
5	*formosus, a, um*	wohlgebildet, schön	3 ×
6	*surripere*	heimlich wegnehmen, entwenden	3 ×

The first one simply lists the words and their German translations poem by poem, line by line; the last column contains the additional information how often the word occurs in the poetry of Catullus.

Mehr als einmal vorkommende Wörter **6 ×**

	irrumare	maulficken	**16**, 1. 14; **21**, 13; **28**, 10; **37**, 8; **74**, 5
	moecha, ae, f	Ehebrecherin	**42**, 3. 11. 12. 19. 20; **68**, 103
	tympanum, i, n	Kesselpauke; Teller, Scheibenrad	**63**, 8. 9. 21. 29. 32; **64**, 261
5 ×	*arca, ae, f*	(Geld-) Kasten; Gefängniszelle	**23**, 1; **24**, 5. 8. 10; **25**, 5
	basium, i, n	Kuss	**5**, 7. 13; **7**, 9; **16**, 12; **99**, 16
	charta, ae, f	Papyrusblatt, Papier	**1**, 6; **22**, 6; **36**, 1. 20; **68**, 46

The second section lists the words by frequency (and within each frequency range alphabetically). This is useful for students so they can memorize words that are particularly common in a given text.



Alphabetisch **as–bi**

as, assis, m	As, Pfennig	1 ×: **42**, 13
asinus, i, m	Esel	1 ×: **97**, 10
aspirare	einhauchen; beistehen	1 ×: **68**, 64
asserere	daneben pflanzen, säen	1 ×: **61**, 102
asseruare	bewahren, verwahren	1 ×: **17**, 16
assiduus, a, um	fleißig, beharrlich	8 ×: **45**, 4; **61**, 227; **64**, 71. 242; **65**, 1; **66**, 88; **68**, 55; **92**, 4

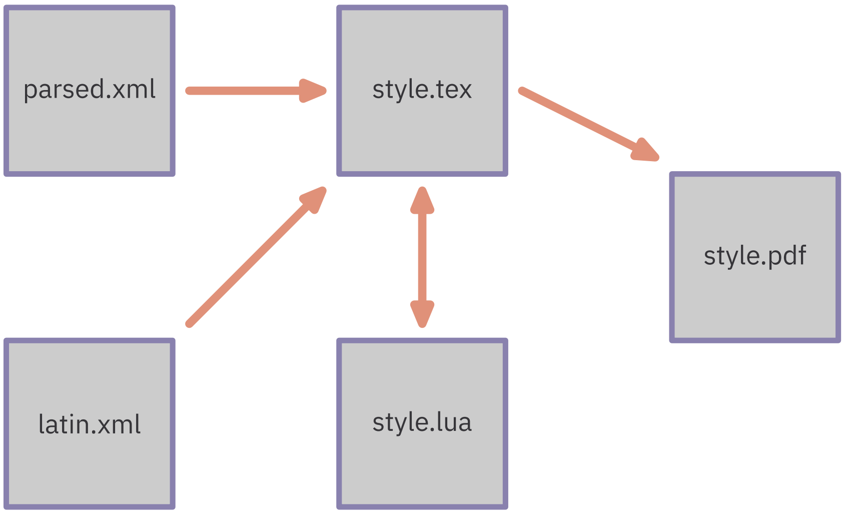
Finally, the last list gives the words and their translations in alphabetic order so students can look up words or do comparisons with other authors.

10. The Workflow

In order to process our file `parsed.xml`, we need a file containing the ConTEXt setups, which we call `style.tex`. And since we want to analyze and massage the information contained in XML with Lua, this style will be accompanied by a file `style.lua`. Finally, we have an XML file `latin.xml` that contains the German translation to Latin words. I have built this file over the years; it contains around 10 000 Latin words. Somewhere at the beginning of `style.tex`, we have to make sure that all these files are loaded:

```
\registerctxluafile{style}{1.001}
\xmlprocessfile{catullus}{parsed.xml}{}
\xmlloadonly{vocabulary}{latin.xml}{}
```

Schematically, this workflow could again be summarized in this graphic:



As we have seen, the root element of parsed.xml is named **<author>**. Our `style.tex` is a bit different from usual ConTEXt styles for processing XML. In general, we typeset the different elements of an XML file as they occur. Here, however, we want to capture, analyze, and transform the data in the XML file before we begin the typesetting proper. Hence, all the processing takes place within our instructions for this root element, and we typeset directly from Lua: we thus create a rule to process the root element and immediately pass control to Lua:

```
\startxmlsetups xml:list_setups
  \xmlsetsetup{#1}{*}{-}
  \xmlsetsetup{#1}{author}{xml:*}
\stopxmlsetups

\xmlregistersetup{xml:list_setups}

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

All the action thus takes place in `style.lua`.

11. Processing Notes in Lua

In this file, we prepare a number of Lua tables in which we will store and order the information:

```
local sections_table  = { }
local alphabet_table  = { }
local frequency_table = { }
```

Then, we move on to processing our root element. The most obvious way would be to simply traverse the different hierarchical levels (sections, subsections) of `parsed.xml`. However, as I mentioned, these subsections contain lots of elements that we do not want to process; we are only interested in **<note>** elements when they have an attribute `voc`. It thus makes sense to look at these elements only. Here's the Lua code to do this:

```
function xml.functions.author(t)
  for note in xml.collected(lxml.id(t), "/**/note[@type=='vok']")
do
    process(note)
  end
end
```

This filters out all the `note` elements with an attribute `voc`. We will then process this information to obtain a Lua table that is pretty similar to the one we have seen in the first part of this article; the same cautions apply; i.e., we need to have our poem and line numbers in a **table.orderedhash** so we can later traverse them in the order in which they appear in the file. I will not repeat information about these tables in this part of the article.

In order to process the note elements, we first have to extract the Latin word:

```
function process(t)
  local word = xml.text (t, "/word")
end
```

We can now begin to populate the tables we created earlier. One will receive the words as keys and the passages (poem and line numbers) as values so that we can later sort the words alphabetically:

```
if not alphabet_table[word] then
  alphabet_table [word] = { }
  alphabet_table [word] ["passages"] = { }
  table.insert (alphabet_table [word].passages, passage)
  alphabet_table [word] ["count"] = 1
else
  alphabet_table [word].count = alphabet_table [word].count + 1
  table.insert (alphabet_table [word].passages, passage)
end
```

This is how we build up the alphabetic table: for every word, we check whether it is already in the table. If it isn't, we create the entry, set its counter to 1, and add the passage where it occurs. If it's already in the table, we increment the counter and add our passage (I have not explained how we fill the variable **passage** as this is very much like populating the tables we built in the first part).

After building these two tables (**sections_table** and **alphabet_table**), we need to populate a third table, which will have the number of occurences as keys. For this, we traverse the **alphabet_table** and look at the **count** of each item; we only add words that occur more than once:

```
for word, word_tbl in pairs (alphabet_table) do
  if word_tbl.count > 1 then
    if not frequency_table [tostring(word_tbl.count)] then
      frequency_table [tostring(word_tbl.count)] = { }
    end
    table.insert (frequency_table[tostring(word_tbl.count)], word)
  end
end
```

12. Sorting Tables

Once all these operations have finished, we have access to our data in three Lua tables:

1. **`sections_table`** contains the words in the sequence in which they occur in the text and will be processed by poem and line;
2. **`alphabet_table`** has the words as keys and the passages where they occur as values; if we want to process it, we need to sort the keys alphabetically;
3. **`frequency_table`** has the number of occurences as keys and the words as values; here, we will need to sort both the keys (numerically, in descending order) and the values (alphabetically).

In theory, sorting table keys is not too difficult in Lua: you sort the keys, store the sorted keys in an intermediate array and retrieve them from there. Here is a little function that will sort keys numerically, in descending order:

```
function sort_descending (t)
  keys = { }

  for k, _ in pairs (t) do
    table.insert (keys, k)
    table.sort (keys, function (a, b) return tonumber(a) >
tonumber(b) end)
  end

  return keys
end
```

Here's how we use this function so we can loop through **`frequency_table`**:

```
sorted_frequencies = sort_descending (frequency_table)

for _, f in ipairs (sorted_frequencies) do
  (do someting with f and frequency_table [f])
end
```

The list of words that occur with frequency **`f`** will now be in the array **`frequency_table [f]`**. Here, we want to sort these tables alphabetically. This becomes problematic as soon as your table contains characters other than ASCII. For Latin, this is rarely the case but for Greek, everything is utf8. This is where the ConTEXt extensions to Lua come in handy: we want all words to be sorted as lowercase, unaccented characters. So we write a function:

```
function sort_alphabetically (t)
  keys = { }

  for k, _ in pairs (t) do
```

```
    table.insert (keys, k)
    table.sort (keys, function (a, b)
      return characters.lower(characters.shaped(a))
      < characters.lower(characters.shaped(b)) end)
  end

  return keys
end
```

With this function, we can retrieve our words in alphabetical order:

```
sorted_words = sort_alphabetically (frequency_table[f])

for _, word in ipairs (sorted_words) do
  context (word)
end
```

The functions **`characters.lower()`** and **`characters.shaped()`** are explained in the cld manual, chapter 11.2; they return a utf8 string in lower case and with diacritics removed. We can use the same function to sort the keys (words) in our **`alphabet_table`**.

13. Typesetting Huge Tables

Now that we have all the tables and sorting mechanisms in place, we can begin the typesetting itself. One phenomenon that became apparent during my tests with these big XML files is that the tables became extremely large, sometimes reaching hundreds of pages. The existing table mechanisms all had problems working with these huge tables. When I asked about it on the list, Hans created a new table mechanism, `framedtable`, which is simpler than the existing tables and can handle these huge files. This is what we will use here (even though the dataset for Catullus is fairly small and could be compiled with the natural tables environment, if we wanted). In order to save ourselves some typing, we create a couple of abbreviations:

```
local starttable = context.startframedtable
local stoptable  = context.stopframedtable
local startrow   = context.startframedrow
local stoprow    = context.stopframedrow
local startcell  = context.startframedcell
local stopcell   = context.stopframedcell
```

With these definitions, we can begin to loop through our Lua tables and typeset their content. We will not go through all of them, but just provide a few examples. Let us start with the alphabetic table; this is the easiest one to iterate through since every table row is for just one word. So our code will look like this:

```
starttable()
  for _, word in ipairs (sorted_alphabetic_words) do
    startrow()
       startcell()
         context (word)
       stopcell()
       startcell()
         context (vocabulary [word])
       stopcell()
       startcell()
         context (alphabet_table [word].count)
         context (" ×")
       stopcell()
     stopcolumn()
   end
stoptable()
```

Things get a little bit more interesting for the frequency table. In general, there will be several words for every given frequency **n**. We want to write this frequency only once, for the first word; for the following words, the first column of the table should be empty. Here's how we do this:

```
starttable()
  for _, frequency in ipairs (sorted_frequencies) do
    startrow()
       startcell()
         context.bold (frequency)
       stopcell()
       sorted_words = sort_alphabetically (frequency_table
[frequency])
       for i, word in ipairs (sorted_words) do
         if i > 1 then
           startrow()
             startcell()
             stopcell()
         end
         startcell()
            context (word)
         stopcell()
         startcell()
            context (vocabulary [word])
         stopcell()
       stopcolumn()
     end
   end
stoptable()
```

This way, when we process our root element, we first build our Lua tables, then we typeset them. In pseudo-code, this is what the entire process looks like:

```
function xml.functions.author(t)
  for note in xml.collected(lxml.id(t), "/**/note[@type=='vok']")
do
    populate (sections_table)
    populate (alphabet_table)
    populate (frequency_table)
    typeset (sections_table)
    typeset (alphabet_table)
    typeset (frequency_table)
  end
end
```

14. Using TEX

While the entire typesetting process is thus done in Lua, some parts of it are easier to code in TEX. To provide one example, we want to set up our page headers and tables. For this, we use named setups in our `style.tex` file:

```
\startsetups frequency_table
  \setupframedtablecolumn [1]
    [width=1.0cm,align=left,style=bold]
  \setupframedtablecolumn [2]
    [width=4.5cm,align={normal,verytolerant}]
  \setupframedtablecolumn [3]
    [width=6.5cm,align={normal,verytolerant}]
  \setupframedtablecolumn [4]
    [width=5.0cm,align={normal,verytolerant}]
  \dontcomplain
\stopsetups
```

In our Lua file, we can simply call these setups:

```
function xml.functions.author(t)
  for note in xml.collected(lxml.id(t), "/**/note[@type=='vok']")
do
    context.setups("frequency_table")
    typeset (frequency_table)
  end
end
```

The same is true for setting up the page headers or defining markings for these headers: it is easier to do the setups in ConTEXt and just use them in Lua.

This, then, is the full description of our workflow as outlined in the beginning: the same Lua style file will gather, analyze, and typeset the data from our XML file.

15. Final Thoughts

I am certain that my code is horribly inefficient; any decent programmer could probably point out ways how to optimize it. However, in my experiments, this is not too important. Some of my lists are very large and accordingly take a long time to compile; e. g., one, for the Greek author Plutarch, has around 20 000 entries and compiles to a PDF of 1250 pages. Typesetting this file takes a long time, up to 28 minutes on my computer (which is reasonably fast); the compile time is shorter for successive runs, when the `tuc` file is already present. However, ConTEXt spends most of this time typesetting these long tables. If we comment out the typesetting part of our Lua style and let it just process the table, the time compiling the file is reduced to around 6 seconds. This shows that optimizing the Lua code for processing the XML file would not be very useful; it would shave one or two seconds off a compile time of 14–30 minutes.

Conclusion

The use of ConTEXt I have shown in this article is very specialized. Nevertheless, the approach itself and the tools used have a much wider area of application. I hope they will be interesting and inspiring for other users from different domains.