Matthew DeHass

6/24/2023

**Treebank Search Tool Codebook**

**NOTE: THIS CODEBOOK IS SAVED TO THE GITHUB REPOSITORY, AND I ALSO KEEP A COPY OF THIS AS A PDF THERE WHICH SHOULD BE UPDATED EVERY NOW AND THEN.**

This is my tentative “codebook” for the Treebank Search Tool (hereafter **TST**), an XQuery module and script written in BaseX[[1]](#footnote-1) for searching the LDT v2.1 and PROIEL v2.0 (**double check**) treebanks at minimum.[[2]](#footnote-2) The goal of the project is to understand how a feature of syntax varies between genre and register. I put “codebook” in quotes because this will be much more detailed than a finalized codebook should be, it is really a journal where I plan out certain processes in advance. First, I list the steps I need to take to make a fully functional querying system. Next, I will have a “journal” part where I work through the problems. Third, there will be some examples of what the code should look like and how the existing functions should be used. Fourth, there will be a guide for how comments should be structured. Fifth, there will be a list of goals which need to be implemented backwards over the code. This means there are a few steps to this process:

1. Find out how each process in querying the treebanks can be generalized to work with at least LDT and PROIEL; also figure out how it can be generalized to lemmatized texts. Note that you do not necessarily need a **conversion function** (one which converts LDT search terms)
2. Find words based on their pos, relation (i.e. the relation type), or lemma, and also produce a negative search for each of these things (i.e., return anything that is not punctuation, or not a noun)
3. Be able to take these results and find other words with a certain relationship (that is, child, parent, sibling, etc.) which are filtered based on the same criteria in 2). This search should return all the members of a relationship, so I can decide whether I want to look further at relationships of the heads or the dependents. The ability to exclude certain kinds of relationships should be baked into the 2) search algorithm.
4. Be able to nest the searches. The results of a search like 3) can go back as one part of another search like 3)
5. Have a function to find sentences which contain positive results to multiple different queries like that in 3) at the same time. If I want to find sentences where *hortari* is used with an *ut*-clause and an ablative absolute is also present, for whatever reason I would want to do something like that, it should be possible. This is not for **exploratory** work, but **confirmatory** work.
6. For **exploratory** work, may be good to have a function which only returns the children of the PRED root (therefore, a general function for returning children, which can be used for this purpose)
7. There should be a function to print the raw results or put them in a CSV file. The CSV should contain
8. All of these need to be able to export to CSV. A few notes on the formatting:
   1. **Every** CSV output should have the sentence id, source document relative path, citation to location in the text as exact as possible, title, author, and URN. **There should be a function for reading and retrieving sentences from this list**, **and a function for exporting in treebank format.** These should be in the far left columns; since they are not necessary for text analysis, I want to easily be able to chop off the left 6 columns and put the data into some kind of software.
   2. The rest of the CSV output should vary by use case. If I am using the simple search in 2), it is as simple as all the info in the word’s attributes (14 columns, since I want there to be one for each postag character). If a complex search (**this part is under construction!!!!**), it should have the forms of the two related words on one line, the relation and relationship of each, and also the word ID’s, so they can be found with more info.
   3. CSV example header:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| path | urn | title | author | sentence | term-a | a-subdoc | term-b | b-subdoc | b-relation-to-a | (add more later) |

<csv>

<record>

<path></path>

<urn></urn>

<title></title>

<author></author>

<sentence></sentence>

<term-a></term-a>

<a-subdoc></a-subdoc>

<term-b></term-b>

<b-subdoc></b-subdoc>

<b-relation-to-a></b-relation-to-a>

</record>

</csv>

1. In line with 8), a function for exporting a series of word nodes in plain-text, as a treebank document, and as a CSV should be available.
2. Also have a function that can find the highest node in a sentence of a certain type (to cover situations where the PRED is not the highest because two main clauses are coordinated).

Also keep in mind that all code examples will be colored in “White, Background 1, Darker 50%” (a color at the far bottom left of the theme colors in my palette), such as the following:

declare %public function deh:sentence-lengths($docs as node()\*) as item()\*

{

for $doc in $docs

for $sentence in $doc//sentence

let $words := $sentence/word[deh:check-punct(fn:string(@form)) ne true()]

return <count sentenceid='{$sentence/fn:string(@id)}'>{fn:count($words)}</count>

};

**USAGE:**

Intro:

The TST is primarily used in the **AGLDT\_Search\_Test.xq** file (hereafter called **Tree XQ**). The file has two modules it requires: **FunctX** (cite later), and my module, **agldt\_search.xqm**. The names are old and misleading, of course: PROIEL will soon be compatible as well. The following information in this intro section covers some of the basics of the Tree XQ file that you shouldn’t need to worry about day-to-day, but is important to the functioning of the file.

The Tree XQ file is kept in the *ma\_thesis\_23-24* GitHub repository; within that same directory, you can find the **agldt\_search.xqm** file as well, and it is referenced at the top of Tree XQ in the following way:

import module namespace deh = "https://www.youtube.com/channel/UCjpnvbQy\_togZemPnQ\_Gg9A" at "./agldt\_search.xqm";

We import the module with the namespace deh:, which is the first three letters of my last name and likely a bad convention, but it works. I have no plans for a separate module currently, but that may change in the future. I pick a random URL I know for the URI, although I do not know if it is necessary; I did that early in my XQuery career, so I will hopefully change it before long.

As mentioned above, you also need the **FunctX** module, which is declared in the following way:

import module namespace functx = "http://www.functx.com" at "http://www.xqueryfunctions.com/xq/functx-1.0.1-doc.xq";

I pull it straight of the web, but sometimes it is giving me issues, so I have a local version downloaded that I can switch to just in case. This won’t help people on other systems (although the general advice will), but it is at C:/Program Files (x86)/BaseX/src/functx\_lib.xqm. The next lines in the Tree XQ file are comments giving the LDT postags and their values and the relations with brief explanations of each; the relations are useful, but keep in mind that the **TAGSET.xml** file is kept in the same directory, and always accessible; it does not give explanations, but does “translate” the single letters or short tags into their full-word equivalents, for the LDT at least. The same can be done with PROIEL; I exported them to the **PROIEL-TAGSET.xml** file, but they are always available at the top of every PROIEL xml file in the <annotations/> element, which is directly dependent on the file root (</proiel>). Next is a commented-out set of postags which I put in a map manually, before I was drawing that information automatically.

You also need to have a collection of treebanks somewhere to reference. The easiest way I have found to do so is to make a collection of the appropriate directory. The code looks like this:

declare variable $ldt2.1-treebanks := fn:collection("./treebank\_data/v2.1/Latin/texts");

declare variable $ldt2.1-with-caes-jerome := fn:collection("./treebank\_data/v2.1/Latin");

declare variable $all-ldt := ($ldt2.1-treebanks, fn:collection("./harrington\_trees/CITE\_TREEBANK\_XML/perseus/lattb"));

declare variable $proiel := (fn:collection("./PROIEL-DATA/syntacticus-treebank-data/proiel"));

It pulls every XML file in the directory, including subdirectories, into the collection, so I don’t have to worry about specifying all the subdirectories in the Harrington trees project. As you can see, you are allowed to declare “global” variables before the FLWOR expression proper here, but keep in mind you can do no such thing in a Module file.

Querying the Corpus:

There are several ways to query the corpus. The lowest-tech way is to simply search the XML files themselves, which is helpful in some scenarios. If you are unfamiliar with querying this corpus with my tools, skip this paragraph and go straight to the explanations of deh:query() and deh:search(). Words in LDT are marked by the <word/> token (and Harrington trees too; if anything I mention does not apply to Harrington trees, I will say so explicitly, but otherwise assume all the same things apply). The are subordinate to the <sentence/> element, which is in turn subordinate on the <body/> element (at least in v2.1, which I am using). In order to get to the <word/> elements, you can do $ldt2.1-treebanks/treebank/body/sentence/word, or just $ldt2.1-treebanks//word, as I usually do. After that, you can add any predicates you want; $ldt2.1-treebanks//word[fn:string(@relation)][[3]](#footnote-3), or substitute @lemma, @form, @postag, @id, or @head, if you want the word’s relation to its head, lemma, form as it appears in the text (i.e. if the form is redeuntibus, the lemma is redeo), postag is the set of characters which tells us about part of speech, id is a numeric identifier, and the id of the word’s head, respectively. Most of these apply to PROIEL, although @head is @head-id. See the Guide to the Annotation Schemes below in this section. The point is, you can use any of the basic XQuery functions listed on functx.com (that’s where I go) under the “Strings” section to manipulate the results the way you want. Best to use a function like that if you are not already confident something is spelled a certain way, since there can be idiosyncrasies. If you want only sum, for example, you would put the following:

$ldt2.1-treebanks/treebank/body/sentence/word[fn:string(@lemma) = "sum1"]

Also keep in mind how most lemmas in LDT end in a number, even if there is only one possibility. For PROIEL, it would be:

$proiel/proiel/source/div/sentence/token[fn:string(@lemma) = "sum"]

Note the differences. The structure is different, with <div/> elements dividing up the text by analogy to TEI practice, and the basic unit is the <token/>, not word. Otherwise, many similarities apply.

The main point of entry is the deh:query() or deh:search() functions. These work with both treebanks, although deh:query() still has some issues with that. Both must be used in complement to achieve a robust result, but let us start with the simpler one, deh:search(). This function takes four arguments. The first is a sequence of strings, which correspond to the parts of the LDT’s <word>/@postag attribute and PROIEL’s <token>/@morphology. Each string can belong either to PROIEL or LDT; if they are different, one will be ignored when searching the other. Also, if you want to search for multiple different possible parts of speech, you can do so with PROIEL, but that functionality is not currently extended to other search terms in this sequence. The list of the possibilities for LDT are below:

adjective

conjunction

adverb

exclamation

numeral

noun

pronoun

preposition

punctuation

verb

irregular

first person

second person

third person

plural

singular

future

imperfect

plusquamperfect

present

perfect

future perfect

gerund

gerundive

indicative

imperative

infinitive

participle

subjunctive

active

deponens

passive

feminine

masculine

neuter

accusative

ablative

dative

genitive

locative

nominative

vocative

comparative

positive

superlative

The following are the parameters for PROIEL:

first person

second person

third person

uncertain person

dual

plural

singular

uncertain number

aorist

future

imperfect

pluperfect

present

perfect

resultative

future perfect

past

uncertain tense

gerund

indicative or subjunctive

indicative or imperative

gerundive

subjunctive or imperative

indicative

imperative

infinitive

optative

participle

subjunctive

finite

supine

uncertain mood

finiteness unspecified

active

middle or passive

middle

passive

unspecified

feminine

masculine

neuter

masculine or neuter

masculine or feminine

masculine, feminine or neuter

feminine or neuter

uncertain gender

accusative

ablative

genitive or dative

dative

accusative or dative

genitive

instrumental

locative

nominative

oblique

vocative

uncertain case

no case

comparative

positive

superlative

uncertain degree

no degree

strong

weak or strong

weak

inflecting

non-inflecting

Not all of these seem to be productive in the treebank: this requires review.

Note that most parts of speech are not listed in the same way for PROIEL. They have their own codes, which are listed below. The reason I don’t use the full names but the codes is because the codes are specific and idiosyncratic enough that I want to force myself to review the annotation scheme, which is again listed at the top of every PROIEL file. I have removed the ones which do not apply to the Latin trees.

A- Adjective

Df Adverb

Ma Cardinal Numeral

Nb Common Noun

C- Conjunction (coordinating)

Pd Demonstrative Pronoun

F- Foreign word

Px Indefinite Pronoun

I- Interjection

Du Interrogative Adverb

Pi Interrogative Pronoun

Mo Ordinal Numeral

Pp Personal Pronoun

Pk Personal reflexive pronoun (i.e. *se*)

Ps Possessive pronoun

Pt possessive reflexive pronoun

R- Preposition

Ne Proper Noun

Pc Reciprocal Pronoun (*alterutrum*)

Dq Relative Adverb

Pr Relative pronoun

G- Subjunction (subordinating conjunction)

V- Verb

The second argument of the function is the relation: it is just a string. This is mainly handled by the deh:relation-match() function, which passes a 1 and nothing higher or lower if you have a successful match, which is how the higher function decides to return the token. You can pass a sequence of these if you want to get every token that matches the WHOLE required list of postags and whose relation at least partially matches the chosen parameter (so, a search for “PRED” will still return “PRED\_CO”). This returns true if the @relation matches ANY of your search terms, so you do not have to worry about a list which includes PROIEL relations screwing with your LDT relations.

The third argument of the function is the lemma: this works the same as the @relation, but it checks the string with fn:matches(), which supports RegEx syntax; I usually search “^sum(1|)$” when I want to search both treebanks at once, because of the aforementioned difference between the two, how LDT suffixes a number onto most lemmas but PROIEL does not. Note that this RegEx expression assumes I am looking for *sum1*; this is because I knew this in advance.

The fourth argument is either a treebank document, sequence of treebank documents, or a sequence of tokens (that is, <word/>’s or <token/>’s). The deh:tokens-from-unk() function makes sure any of these can work.

In place of any of these arguments you can pass empty strings or sequences; currently, empty sequences are safer, because I screwed up in making sure you can do both. If you are only using deh:search(),

Sentence marking for search output adds

Also note that, if you want to just get a sequence of all the words, you should do it in the TST as $words := ($all-trees//word, $all-trees//token), because you should not forget that the LDT series uses <word/> elements for each token, and PROIEL uses <token/>.

Annotating Results:

Returning results from deh:query() or using deh:mark-node adds a lot of information to the words, allowing you to query the results in turn for further patterns. deh:mark-node is supposed to retrieve the following information, if it can get all of it:

@deh-urn: The document’s base-uri, which may or may not have the urn:cts

@deh-subdoc The subdoc of the given word

@deh-title The given title of the work, which is retrieved from the web if necessary

@deh-author The given author, also retrieved from the web if needed

~~@deh-docpath Likely deprecated, this replaces the spaces in the docpath, but there should be no spaces anymore…~~

@deh-sen-id Id of the parent sentence

@base-uri As it says, the base uri of the document

This, after I am done modifying it 7/29/23, should rely on deh:mark-node primarily, which retrieves the info for PROIEL or LDT. deh:mark-node() currently only annotates about 4 words/second, which is ridiculous, but that’s alright, I’m working on it.

Use deh:query() when you want the function to handle bundling the results.

This function also takes a “comment” as its final argument, which can be empty, but, if not, allows you to add an annotation to the root of the results, or the top of the CSV file.

Guide to the Annotation Schemes:

Explanation of Minor AGLDT\_Test\_Search.xqm Functions:

**Journal:**

**Issue 1: LDT and PROIEL Compatibility:**

The tough decision is whether to leave each treebank in their own format or transfer both over to an independent format, so one function works for everything. Let us first look at the conversion options.

**MODULE FORMATTING NOTES:**

This section covers the rules for formatting the .xqm file for this project (titled agldt\_search.xqm). I have currently not decided whether or not to have multiple .xqm files, as it is uncertain whether even XQuery will be necessary. I will either create a separate module for PROIEL’s standard, or combine everything together (currently my preferred solution) and rename the files to something more helpful. As it stands, the AGLDT is the only treebank type this is compatible with.

At the top should be the relevant declarations, of course, such as the below:

xquery version "3.1";

module namespace deh = "https://www.youtube.com/channel/UCjpnvbQy\_togZemPnQ\_Gg9A";

import module namespace functx = "http://www.functx.com" at "http://www.xqueryfunctions.com/xq/functx-1.0.1-doc.xq";

The Xquery version and module namespace’s obligatory status should be obvious (the namespace gives a handle for us to import the module and reference the functions). We use several of the helpful FunctX functions here, so I have imported it in this .xqm file as well.

Sections of the file should be marked as so:

(:------------------START deh:postag-andSearch AND DEPENDENCIES------------------------------:)

(: code :)

(:---------------------------END deh:postag-andSearch AND DEPENDENCIES--------------------------:)

A comment contains a line made up of a series of dashes (the number/length not being important). A description is placed in the middle, and it will start with END if it is marking the end of a section. Functions do not have to all be cordoned off into sections. Additionally, although I am currently using it to encapsulate important functions and their helpers/dependent functions, it does not have to be used that way. Just make sure the labels are clear and helpful.

ADD NOTES ON FUNCTION DESCRIPTION FORMATTING HERE

**PROIEL COMPATIBILITY (ONGOING PROBLEM):**

The first function I want to implement in both LDT and PROIEL would be retrieving every single sentence and their citations, so I can identify and remove doubles.

How to deal with SLASHES? (these are <slash/> nodes which are subordinate to (in the XML sense) <token/> nodes.

In all functions going forward from 7/12/2023, if individual <word/> or <token/> nodes are referred to, use the word ‘token’ universally, just for consistency.

**POINTS OF ENTRY:**

**deh:search():**

**FUNCTION LIST**:

deh:find-highest

* Finds the highest

get-duplicates(): Pass the $all-ldt var from AGLDT\_Search\_Test, gets duplicate passages in the trees based on BOTH urn and subdoc.

get-ldt-conj-relations(): returns the @relation of every child of every ‘AuxC’ (i.e. subordinating conjunction) in the LDT (and ONLY LDT).

get-sents(): This takes tokens, retrieves only the UNIQUE sentences

is-finite(): Returns whether any single token (ldt or proiel) is a finite verb

main-verbs(): Returns only the main verbs from a set of documents/sentences/tokens/etc.

deh:mark-node:

* Takes an LDT node, and adds the document URI and sentence ID

deh:pick-random($seq as item()\*): This function picks a random value from a sequence. Useful for sampling from results and testing the validity of a search. You need to make sure that you aren't getting undesirable results in a standard way.

main-verbs(): Returns only the main verbs from a set of documents/sentences/tokens/etc. Note that, for this reason, in much of the LDT, this won't get most main verbs in direct speech; however, if it goes on for multiple sentences, it might include that. ANOTHER CHARACTERISTIC OF THIS is the fact that it retrieves auxiliaries in the LDT, and the participles within a periphrastic in PROIEL; for now, that does not matter, just keep it in mind.

pr-main-verbs(): This is a helper function to deh:main-verbs, which handles extracting PROIEL main verbs. The process is different, since @relation="pred" is allowed in more contexts than the LDT

print():

print-rel(): Like *print()*, but also puts the @relation in parentheses after each word.

deh:proc-highest(): Helper function to deh:find-highest, may be broken due to the changes to deh:word-postag, so, again, it might be good to later take all the term processing stuff and put it into another function

remove(): This removes a series of nodes from their respective sentences (then returning the sentences without them, although none of the sentence metadata is currently retained).

remove-from-sent: Usually used as a helper to *remove()*. This function takes a sequence of tokens FROM A SINGLE SENTENCE and returns their original sentences with said tokens removed. It will NOT work with tokens from multiple sentences ALSO KEEP IN MIND THIS FUNCTION RETURNS A SENTENCE NODE WHETHER IT IS EMPTY OR NOT

Treebank “Axis” Functions:

Named “Axis” functions for their similarity to XPath axes

deh:return-children()

deh:return-parent()

deh:return-siblings()

deh:return-ancestors() (only direct, one parent to another)

deh:return-descendants() (all children, and their children, and their children, etc.; that whole section of the tree)

search-text(): Allows one to search the treebanks by bare text. No punctuation, and spaces between every token. This includes que's and turning 'nec' into 'ne c' for proiel! Although that final one does not seem 100% consistent

deh:tokens-from-unk($tokens) accepts either a treebank document, or any set of elements from within that document, and returns whatever tokens are contained. This simply checks if what you have passed is a sequence of <token/> or <word/> elements, and returns them if so, but if not, returns $tokens//word or $tokens//token. So, you can pass a number of sentences as well, and it will return all the words.

Work-info(): (*description here*)

* Note that you have created a series of tags for retrieving each work by its title. I copy the following for the tested short versions:
  + "Met" - Ovid's Metamorphoses
  + "Pere" - Peregrinatio Aetheriae
  + "Elegie" - Propertius
  + "Elegia" - "Tibullus
  + "In Cat" AND "Again" - Cicero's In Catilinam
  + "Hist" - Sallust's Bellum Catilinae
  + "Aen"- Aeneid, Vergil
  + "Petr" AND "Saty" - Petronius' Satyricon
  + "Fab" - Phaedrus
  + "Res" - Augustus' Res Gestae
  + "Aug" - Suetonius' Divus Augustus
  + "Ann" - Tacitus' Annales
  + "Cael" - Cicero's For Marcus Caelius
  + "Sati" - Juvenal's Satires
  + "Gall" - Caesar's de Bello Gallico
  + "Carm" - Catullus' Carmina
  + "Amor" - Ovid's Amores
  + "Att" - Cicero's Epistulae ad Atticum
  + "off" - Cicero's de Officiis
  + "Vul" - Jerome's Vulgate
  + "agri" - Palladius' Opus Agriculturae
* I have also saved the following sequences for easy repetition:
  + “Small, elite audience”: ("off", "In Cat", "Again", "Cael", "Sati", "Elegia", "Gall", "Aen", "Met", "Petr", "Saty", "Aug", "Ann", "Elegie", "Carm", "Amor", "Hist")
  + “Large audience”: ("Res", "Fab", "agri", "Vulg")
  + “Small, private audience”: (“Att”, “Pere”)

**NOTES:**

* In separating out quotes, you cannot rely on just pulling out that which is directly dependent on the verb of saying in LDT, since some other ancillary stuff often shows up; you have to filter out whatever does not come inbetween quotes primarily, that is the only way, and it has to work across sentences.
* In XQuery, ‘=’ can compare multiple values (i.e., let $seq := (“foo”, “bar”) \n [“foo” = $seq] will return true(), but [“foo” eq $seq] will not), but ‘eq’ can only compare individual values, as just stated.

1. https://basex.org/ [↑](#footnote-ref-1)
2. David Bamman and Gregory Crane, “The Ancient Greek and Latin Dependency Treebanks,” in *Language Technology for Cultural Heritage: Selected Papers from the LaTeCH Workshop Series*, ed. Caroline Sporleder, Antal van den Bosch, and Kalliopi Zervanou, Theory and Applications of Natural Language Processing (Berlin/Heidelberg: Springer-Verlag, 2011), 79–98 (citation requires verification); Dag T. T. Haug and Marius L. Jøhndal, “Creating a Parallel Treebank of the Old Indo-European Bible Translations,” in *Proceedings of the Second Workshop on Language Technology for Cultural Heritage Data (LaTeCH 2008)*, ed. Caroline Sporleder and Kiril Ribarov, 2008, 27–34, http://www.lrec-conf.org/proceedings/lrec2008/workshops/W22\_Proceedings.pdf#page=31; also note that the Late Latin Charter Treebank, and Corpus of the Epigraphy of the Italian Peninsula of the First Millenium are other possibilities (they are mentioned in “Treebanks and Corpora” in the Thesis Guide). However, I have not decided whether to include these yet. [↑](#footnote-ref-2)
3. Keep in mind, you must make the attribute a string, since it is not by nature. [↑](#footnote-ref-3)