Syntactic Sugar for Syntactic Queries: Sequential Representations for Dependency Queries

Niklas Deworetzki[1], Arianna Masciolini[2]
[1]Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg
[2]Språkbanken Text, SFS, University of Gothenburg
Gothenburg, Sweden
{niklas.deworetzki, arianna.masciolini}@gu.se

Abstract

Syntactic query languages such as Grew and dep_search allow looking for grammatical patterns in linguistically annotated corpora. However, these languages are often unsupported by large-scale corpus management tools, where queries are of an essentially sequential nature. In this paper, we present CQP/TREE, a tool to convert syntactic queries into CQL, the Corpus Query Language used in Corpus Workbench, SketchEngine, Korp and several other such systems. In this framework, syntactic queries act as *syntactic sugar* – they allow expressing complex CQL queries in a more readable and concise fashion, thus bridging the gap between expressive linguistic search and large-scale corpora. CQP/TREE is available as a web and command-line tool, as well as an open source Python library.

Keywords: Corpus Search, Treebanks, Corpus Tools

1. Introduction

Treebanks allow corpus linguists to answer a variety of questions relying on syntactic structure. This has led to the development of an array of domain-specific query languages and tools, all designed to facilitate the description of tree and graph structures. A prominent example is the graph matching and rewriting tool Grew-match (Guillaume, 2021), which powers a popular and user-friendly interface to Universal Dependencies (UD; de Marneffe et al., 2021) treebanks.[1]

Large-scale corpora, however, are often only available through more established systems such as Corpus Workbench (Evert and Hardie, 2011), Sketch Engine (Kilgarriff et al., 2014) and Korp (Borin et al., 2025). These tools all rely on CQL, the Corpus Query Language part of Corpus Workbench's Corpus Query Processor (CQP), which allows searching for sequences of tokens with certain properties.

Over time, CQL has been made more accessible through graphical query builders such as Korp's "extended mode"[2], and recent optimization work aimed at reducing the execution time of queries (Ljunglöf et al., 2024; Deworetzki et al., 2024) provides further evidence of the continued investment of the corpus linguistics community in CQP-based corpus tools. However, while the sequential nature of this query language does not technically prevent users from describing syntactic relations between words, it severely compromises the ergonomics of doing so. This results in long, complex queries even for common and relatively straightforward use cases, such as searching for compound tenses (for some examples, see Section 2). It is therefore common practice among linguists to approximate syntactic queries with under- or overgenerating CQL strings that either are partial in their coverage or require substantial manual filtering.

In this paper, we introduce CQP/TREE, a query translator that converts high-level tree queries into CQL. Currently, the tool can translate from three different languages: Grew (Guillaume, 2021), dep_search (Luotolahti et al., 2017) and deptreepy (Ranta and Masciolini, 2025). CQP/TREE is available as a Python library, as well as a standalone application, which can be used as both a CLI tool and through a web-based GUI. The latter, displayed in Figure 1, integrates with Korp (Borin et al., 2025), the linguistic research platform of Språkbanken, the Swedish Language Bank.[3]

The remainder of this paper is organized as follows. In Section 2, we demonstrate the usefulness of syntactic queries for investigating certain linguistic phenomena and compare them with their sequential equivalents. Section 3 outlines the common traits of the various languages explicitly designed for this use case, while Section 4 covers the CQL constructs we leverage for translation. Sec-

[1]universal.grew.fr

[2]spraakbanken.gu.se/en/tools/korp/user-manual#extended-search

[3]CQL queries throughout this paper can be run directly on Korp (spraakbanken.gu.se/korp), using its Advanced mode. Grew, deptreepy and dep_search strings can be converted into valid Korp queries with CQP/TREE. Linguistic examples are selected from the results of such queries on Korp's default corpus selection, consisting of news texts in contemporary Swedish.

Proceedings of the Fifteenth Language Resources and Evaluation Conference (LREC 2026), pages 11669–11678
11–16 May 2026. ©ELRA Language Resources Association (ELRA), 2026

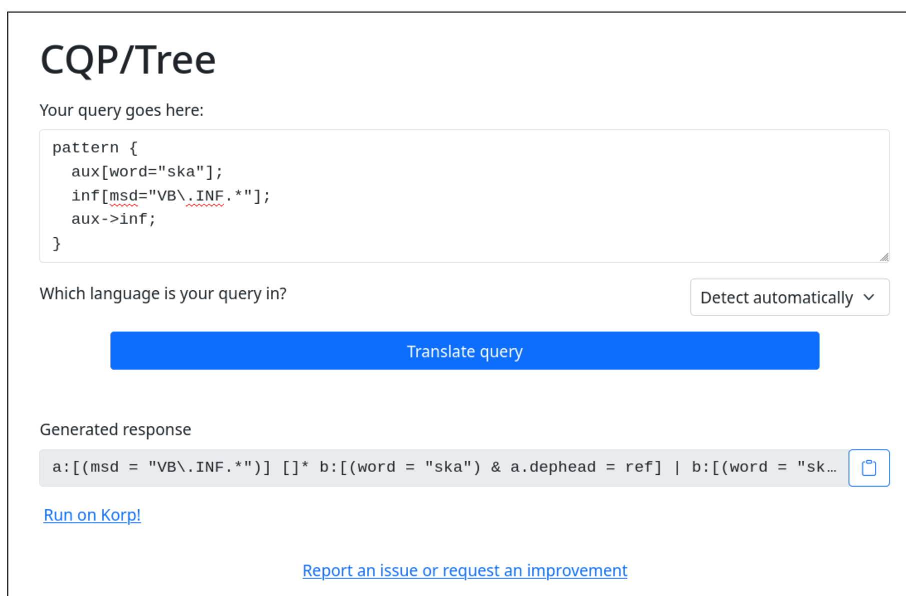

Figure 1: CQP/TREE's web interface.

tion 5 details the inner workings of our tool, whose limitations are discussed in Section 6. In Section 7, we present the results of an evaluation based on randomized testing. Section 8 provides details about language support, followed by an outline for how other corpus systems can be supported (Section 9), as well as some conclusions and directions for future work (Section 10).

2. A case for syntactic queries

Let us consider the problem of retrieving future-tense sentences from a large Swedish corpus. In Swedish, future tense can be expressed periphrastically, with an infinitive introduced by the present-tense auxiliaries *kommer* ('come') – optionally followed by the infinitive marker *att* ('to') – or *ska* ('shall'):

(1)	*Det*	***kommer***	***(att)***	***bli***	*svårt*
	it	comes	(to)	become	hard

'It is going to be hard'

(2)	*Ingen*	*tror*	*att*	*han*	***ska***	***vinna***
	nobody	thinks	that	he	shall	win

'Nobody thinks that he will win'

A first attempt to search for the latter construction (*ska* + infinitive) could consist in simply juxtaposing two CQL tokens in their prototypical order, obtaining the query

(3)
```
[word="ska"] [msd="VB\.INF.*"]
```

This describes a bigram where the surface word form of the first item is *ska*, while the second element is annotated as an infinitival verb form.[4] A more detailed description of CQL constructs is given in Section 4.

The result of this query is indeed a list of *ska* + infinitive constructions, but it does not contain any sentences where other words appear between the auxiliary and infinitive. This is a common occurrence in Swedish, a language with V2 word order. To address this issue, we may allow for an unspecified number of other tokens between *ska* and the infinitive, which is achieved by interposing the string `[]*` (a token with no set characteristics, repeated zero or more times):

(4)
```
[word="ska"] []* [msd="VB\.INF.*"]
```

Now, the results also include phrases like

(5)	*Nu*	***ska***	*det*	***bli***	*vapenvila*
	now	shall	it	become	ceasefire

'Now there will be a ceasefire'

Yet, the query fails to capture the less common cases in which the infinitive precedes the auxiliary, such as

(6)	*Men*	***spela***	***ska***	*hon*	*inte*
	but	play	shall	she	not

'But playing, that she will not do'

Sentences like this can be easily retrieved by mirroring the query:

(7)
```
[msd="VB\.INF.*"] []* [word="ska"]
```

[4] MSD stands for MorphoSyntactic Description; see spraakbanken.gu.se/korp/markup/msdtags.

A word order-agnostic query is obtained by combining the two with a disjunction (|):

(8)
```
[word="ska"] []* [msd="VB\.INF.*"] |
[msd="VB\.INF.*"] []* [word="ska"]
```

However, the last three queries are overgenerating, since nowhere is it specified that the auxiliary and infinitive are in any way related to each other. This becomes even more apparent if we replace the auxiliary *ska* with *kommer* to look for sentences similar to that in example 1. False positives include sentences like

(9) *Vi* *kan* ***hjälpa*** *de* *som* ***kommer***
we can help they who come
'We can help those who come'

where *komma* is used as a lexical verb rather than as an auxiliary. CQL makes it possible to specify such a constraint through token labelling. This results in an even longer query:

(10)
```
aux:[word="ska"] []*
inf:[msd="VB\.INF.*" & dephead=aux.
    ref] |
inf:[msd="VB\.INF.*"] []*
aux:[word="ska" & inf.dephead=ref]
```

With labels (`aux` and `inf`), we specify that the syntactic head of the infinitive should be the ID (`ref`) of the auxiliary. This constraint has to be encoded and adjusted for all the ways in which the tokens can be permuted.

The same query in Grew style, on the other hand, is a more readable

(11)
```
pattern {
  aux[word="ska"];
  inf[msd=re"VB\.INF.*"];
  aux->inf;
}
```

In deptreepy syntax, the query easily fits into a single line:

(12)
```
TREE_ (word ska) (msd VB\.INF.*)
```

Using dep_search, it becomes a mere

(13)
```
ska > msd="VB\.INF.*"
```

In the Grew-style query, we first enumerate the two tokens and their characteristics, then specify that there should be a dependency edge from the auxiliary to the infinitive, independent of their sequential order.[5] The queries for deptreepy and dep_search, on the other hand, describe a tree structure where the surface word form of the syntactic head is *ska* and one of its direct dependents is annotated as an infinitive verb form.

3. Syntactic Queries

A common feature of query languages for syntactically annotated corpora, such as Grew (Guillaume, 2021), dep_search (Luotolahti et al., 2017), deptreepy (Ranta and Masciolini, 2025), TigerSearch (König et al., 2003) and INESS-Search (Meurer, 2012), is that queries are expressed in terms of tokens and relations between them. Consider examples (11), (12) and (13), all of which describe two tokens in relation with each other.

These three queries do not assume any specific token order, and as such they are matched by any sentence containing both the auxiliary *ska* and an infinitive referring to it, independent of their relative position. To specify a certain sequential order, users can provide additional constraints. For example, adding `aux << inf` to (11) indicates that the auxiliary should occur first.

Lastly, another common capability of syntactic query languages is searching for the absence of relations. Note that this is different from and more complicated than searching for the absence of annotations on a token. As an example, consider the following query, searching for *kommer* + infinitive constructions without the marker *att*:

(14)
```
pattern {
  aux[word="kommer"];
  inf[msd=re"VB\.INF.*"];
} without {
  att[word="att"];
  att->inf;
}
```

For every infinitive, this requires inspecting all occurrences of the word "att" in the same sentence to ensure there is no direct link between them.

4. Sequential Queries

CQL is a well-established sequential query language, originally developed as part of Corpus Workbench and later adopted by a variety of other corpus search systems. While details vary across implementations, the overall idea is to formulate queries as sequences of tokens. Subsequently in this paper, when referring to CQL, we mean the language accepted by the Corpus Query Processor of Corpus Workbench. A discussion on the adaptations

[5] These queries assume the data to be annotated according to the dependency formalism used in Korp, Mamba-Dep (Nivre et al., 2006) – in UD, both the categories and the directions of some syntactic relations differ. Therefore, the Grew-style query is **not** meant to be run directly on `universal.grew.fr`, nor can the deptreepy and dep_search queries be applied directly to UD treebanks.

required to make our approach work in Sketch Engine – which uses a dialect of CQL – can be found in Section 9.

Tokens are the basic building blocks of a query. A token is represented by a pair of square brackets, between which the user can specify constraints about the annotations for that specific token. Annotations are described by key-value pairs, where the key indicates an annotation layer and the value is a string or regular expression denoting an annotation value. An arbitrary token can be queried by leaving the pair of square brackets empty `[]`; a token whose form ends with the letter A would be queried using `[word=".*a"]`. As a consequence of this model, all annotations are associated to individual tokens – even dependencies, which intuitively should be placed between two tokens.

Sequence operations allow searching for sequences of tokens. Simply writing two tokens in succession searches for sentences where they appear next to each other. Additionally, tokens (and sequences) can be modified by suffix operators expressing repetition. The operator `?` marks a token as optional, $+$ indicates that it should be repeated at least once, while `*` expresses both, allowing a token to be omitted or repeated arbitrarily many times. Finally, the `|` operator can be used to represent alternatives. For instance, the query

(15)
```
([word="ska"] |
  [word="kommer"] [word="att"]?)
[]* [msd="VB\.INF.*"]
```

matches sentences containing an occurrence of the auxiliary *ska* or of the sequence `[word="kommer"] [word="att"]?` (the auxiliary *kommer* and the optional marker *att*) followed by an infinitive, with any number of unspecified tokens in between.

Labels on tokens make it possible to compare the annotations of different tokens. Writing `a:[]` assigns the label `a` to a matched token, which can then be referenced as part of subsequent tokens. Consequently, `a:[] [word=a.word]` searches for bigrams repeating the same word form. For a CQL query to be considered valid, all labels defined in it need to be referenced later in the query.

Text spans such as phrases, sentences or paragraphs can be encoded in a corpus as well. It is common for the tag `<s>` to signal sentence starts and for `</s>` to represent the end of sentences. To specify that a given CQL query is to be matched within individual sentence, the user can also append the modifier `within s` to the query itself. Without this modifier, matches may cross sentence boundaries or even span the entire corpus.

Set operations with named results enable users to combine the results of multiple queries. Using the assignment operator $=$, results of a query can be bound to a name, storing results for later retrieval and for use with set operations. Given two result sets `S1` and `S2` from prior queries, the query `S3=union S1 S2` computes the set union ($\cup$) of both result sets and assigns it to the name `S3`. The same can be done using the `intersect` and `diff` operators, corresponding respectively to set intersection ($\cap$) and difference ($\setminus$).

5. From Syntactic Queries to Sequential Queries

The biggest challenge when it comes to translating syntactic queries into sequential ones is managing the different ways token positions are handled. As mentioned in Section 3, the default for syntactic queries is to be order-agnostic. In contrast, in a sequential query, token positions are fundamentally encoded in the query itself. To express a query without a fixed token order in a sequential query language, we have to find all the ways tokens can be arranged, that is all permutations. For a set of three tokens A, B and C there are six possible arrangements: $ABC, ACB, BAC, BCA, CAB, CBA$. A query searching for all arrangements is therefore a disjunction over all permutations. Between two tokens within one arrangement, we insert `[]*`, matching an arbitrary number of arbitrary tokens. To prevent token sequences from crossing sentence boundaries, we append `within s` to the entire query. Combining these techniques, we allow searching for a set of tokens occurring within the same sentence in any order, possibly not contiguously, just as it is the case with syntactic queries when no relation between tokens is specified.

5.1. Relations Between Tokens

When searching for a relation between two tokens, this relation has to be encoded in all permutations. As discussed in Section 4, CQP requires information about relations to be encoded on individual tokens. This is typically achieved through two token attributes: `ref`, storing the unique sentence-level id for the token, and `dephead`, containing the id of a token's syntactic head.[6] An edge $A \rightarrow B$ is therefore encoded by a constraint `A.ref=B.dephead`.

In CQL, every use of a label has to come after its definition and a constraint for a token may not

[6]Attribute names vary across annotation schemes. Throughout this paper, we use Korp naming conventions.

refer to the label of the token itself. Therefore, the token on which we can place the relation constraint varies between arrangements. It has to be put on the dependent or head, whichever comes last in an arrangement. For $A \rightarrow B$ (corresponding to `pattern { A->B }` in Grew syntax), we therefore get the sequential query

(16)
```
A:[] []* B:[A.ref=dephead] |
B:[] []* A:[ref=B.dephead]
```

Notice how this captures both ways of arranging A and B, while the relation constraint is always encoded on the second token. A relation type for an edge $A \rightarrow B$ can be specified using the `deprel` attribute of the dependent token, B.

5.2. Token-level Annotations

To translate constraints about token-level annotations, we generalize the strategy used to translate relations. First, we convert all annotations found in the input query into a global, normalized form that detaches them from the tokens and makes token references explicit. An annotation placed on individual tokens, such as `A:[word="hand"]`, is normalized into `A.word="hand"`, whereas an equation of annotations such as `B.msd!=C.msd` does not require any further normalization. Subsequently, we have to find a token within each arrangement to place the annotation on.

Just as with relations, we have to be careful to not use labels before they are defined. A more complex annotation such as `A.word="hand" & B.msd!=C.msd` is therefore assigned to the first token for which all of A, B and C have been defined. For a sequence ABC, this would be the last token, resulting in:

(17)
```
A:[] []* B:[] []*
 [A.word="hand" & B.msd!=msd]
```

This removes the explicit reference to token C, again because annotations on a CQL token may not refer to the label of the token itself. Note that this is only one out of the six possible permutations of three tokens. In the absence of order constraints, all other permutations would contain a similar copy of this predicate on their last token, independent of which token comes last.

5.3. Order Constraints

As mentioned in Section 3, syntactic query languages typically allow specifying constraints on token order, although they do not encode any by default. Interestingly, when order constraints are added to a syntactic query, its sequential translation becomes shorter: instead of listing all possible token permutations, we only list those that align with the constraints. If we take a set of three tokens A, B and C as an example and add the constraint that A has to appear before C, we only get three possible arrangements: ABC, ACB and BAC.

5.4. Distance Constraints

Another constraint sometimes added to syntactic queries is on the distance between two tokens A and B. The most common use case is searching for bigrams, i.e. pairs of adjacent tokens. In Grew, adjacency is expressed with the $<$ operator, e.g.

(18)
```
pattern {
  aux[word="ska"];
  inf[msd=re"VB\.INF.*"];
  aux < inf
}
```

This is trivial to convert to (3). However, some query languages also allow encoding arbitrary distance constraints. This makes it possible to, for instance, only retrieve sentences like (5), where there is (at least) one word between *ska* and the infinitive that depends on it.

In CQL, distance constraints can be expressed through the addition of empty tokens (`[]`). However, this becomes more and more verbose as the distances to encode and the number of tokens involved in the query increase. We therefore rely on two functions built into CQL, `distance` and its variant `distabs`, which compute the (absolute) distance between two tokens.

If we want there to be at least one token between auxiliary and infinitive, the lower bound on the distance between them becomes 2. Therefore, we can add `distabs(aux, inf)>=2` to query (10), obtaining:

(19)
```
aux:[word="ska"] []*
inf:[msd="VB\.INF.*" & dephead=aux.
  ref & distabs(aux, _)>=2] |
inf:[msd="VB\.INF.*"] []*
aux:[word="ska" & inf.dephead=
  ref & distabs(_, inf)>=2]
```

The usages of `distabs` are highlighted. Note that when placing the predicate onto a token, we do not simply omit the token's identifier as part of the function call. Instead, we have to use the special underscore symbol _ here, which is a reference to the current token. Also, we use `distabs` instead of `distance`, as it returns the absolute distance, independent of token order. If the infinitive comes before the auxiliary in a sequence, using `distance(aux, inf)` would result in a negative distance. This approach works for any number and arrangement of tokens, with any distance between them. We use $>=$ for lower bounds, $<=$ for upper bounds and $=$ for fixed distances.

5.5. Use of Labels

In CQL, all labels introduced within a query have to be used. Otherwise, the query processor will reject the query. We therefore add an ulterior analysis step to our translation. It collects all the labels referenced in constraints for tokens in each arrangement. If there is a label for a token that does not appear anywhere, we omit the label when printing the query in CQL format.

5.6. CQP Recipes

Not all syntactic queries can be translated into a single sequential query. An example of this is (14), where we search for the absence of a relation between two tokens. By using named result sets and set operations, we can combine the results of multiple queries into one, enabling the translation of these more complex queries.

(14) can be translated into two queries: one looking for all infinitives, with or without the marker *att*, and one searching for those introduced by such particle. Subtracting the two result sets only leaves sentences with *att*-less infinitives. Upon translation, we get the following sequence of operations, which we call a *CQP recipe*:[7]

(20)
```
A=[word="kommer"] []*
      [msd="VB\.INF.*"];
B=[word="kommer"] []* a:[word="att"]
     []* [msd="VB\.INF.*" & dephead=
    a.ref];
diff A B;
```

While this enables us to translate a much larger set of queries, there is a trade-off in generating recipes: our approach relies on Corpus Workbench's support to perform set operations on result sets, which is not available in most online tools.

6. Limitations

While our presented approach covers a large set of functionality from different syntactic query languages, translation into a sequential query format comes with some limitations.

Potential undergeneration for multi-token queries arises from the way Corpus Workbench executes queries. Consider (21) below – a slightly simplified version of query B in recipe (20):

(21)
```
[word="kommer"] []* a:[word="att"]
    []*
[msd="VB\.INF.*" & dephead=a.ref]
```

[7] To avoid combinatorial explosion and improve readability, the order constraints aux ≪ att; att ≪ inf have been added to (14) before conversion.

Corpus Workbench starts by searching for all possible starting points for matches for this query. In this case, it finds the positions of all occurrences of the word *kommer* in the encoded corpus as a set of potential matches. For each of these potential matches, Corpus Workbench then expands token by token to the right. This is done until a token is encountered that violates one of the constraints encoded in the query, upon which the potential result is discarded. Alternatively, once the entire query has been matched, the sentence is returned to the user. What this means for our example is that, after the word *kommer*, Corpus Workbench expands to the right until the first occurrence of the word *att* is encountered. From there on, each potential match is further expanded until finding an infinitive with a matching dependency relation. However, this comes with a subtle issue: if there are multiple occurrences of the word *att* between *kommer* and the infinitive, Corpus Workbench always chooses the first one, even if this is not the *att* introducing the infinitive. Poorly edited texts with two consecutive occurrences of this particle, for instance, are typically parsed as follows:

(22) (dependency arcs: kommer –VG→ att; att –IF→ att; att –IF→ kosta)

Det	***kommer***	***att***	***att***	***kosta***	*mer*
it	comes	to	to	cost	more

'It is going to cost more'

In cases like this, queries can "get stuck" on an irrelevant token (the first *att*, whose only direct dependent is the second one) and fail to find a match, even if backtracking would yield one. As a consequence, our generated queries might under-report results for queries containing three or more tokens. This issue, however, is entirely dependent on the underlying query engine and different CQL implementations might allow backtracking.

Reporting of matches differs between CQP and other corpus search systems. To demonstrate that, consider the somewhat contrived example of a query looking for two identical word forms within the same sentence. In Grew, such a query would be expressed as

(23)
```
pattern {
  A[]; B[]; A.word = B.word;
}
```

and in CQL as

(24)
```
a:[] []* b:[a.word = word] |
b:[] []* a:[word = b.word]
```

Both queries will find the same token pairs in a corpus. However, Grew reports every pair of tokens twice, once matching A as the first token and once

matching B first. With CQP, on the other hand, every pair is only reported once. Generally, whenever there are multiple ways of matching a token sequence, CQP will only report one match. This also happens when there are several possible overlapping matching sequences that end with the same token. In this case, CQP will only report the longest match.

Query size might be the biggest limitation of our translation approach, deeply rooted in finding the token arrangements for a query. For a set of n tokens there are $n!$ permutations, meaning that the size of the generated sequential queries grows quickly. While our tool can generate queries larger than the examples presented so far,[8] big queries can get rejected by the systems executing them. In the case of Corpus Workbench, there is an upper limit of 5000 token patterns for an input query. A syntactic query with 6 tokens and no order constraints already surpasses this limit. As each arrangement consists of 11 token patterns (6 for the tokens + 5 for the `[]*` between them), a total of 11 $\times$ 720 = 7920 token patterns would be required.

7. Evaluation

To evaluate the capabilities of our translator, we use a strategy akin to *property-based* or *random* testing (Fink and Bishop, 1997; Hamlet, 1994), where a program is provided with randomized inputs to see whether certain properties hold. In our case, we generate randomized Grew queries to evaluate whether a given Grew query and its CQP translation find the same matches.

7.1. Experimental Setup

To compare queries, we encode the UD version of the Swedish treebank Talbanken (Nivre and Smith, 2025) with the same annotation layers in both Corpus Workbench and Grew. To do that, we strip enhanced dependency relations from the corpus and restrict the feature set to word forms, lemmas, universal and language specific part-of-speech tags and relation labels. Sentence boundaries and dependency relations are encoded as presented in, respectively, Sections 4 and 5.1.

Random query generation is limited to queries with at most five tokens, only referring to annotations and values present in the encoded corpus. We then filter the generated queries according to two additional criteria: 1. queries containing empty tokens not part of any dependency relation are discarded, as such tokens are trivially matched on any token in a corpus and therefore do not test the translation in any meaningful way and 2. queries yielding more than 1000 matches are also excluded, as Grew only provides detailed information – which we rely on to correctly count and compare matches – about the first 1000 hits.

We compare query systems by counting the number of unique matching sentences found by each query. This is a workaround for the issues arising from the different ways in which Corpus Workbench and Grew report results (cf. Section 6). In Grew, we count the number of sentences by fetching detailed information for individual matches and then extracting the sentence id of each match. To prevent Corpus Workbench from reporting multiple matches per sentence, we expand matches until the end of their sentence, so that they become overlapping and only one instance is reported, as explained in Section 6.

At this point, it would be reasonable to expect both systems to report the same number of matches, but there is one additional difference between them. In Grew, each sentence is added a virtual *anchor node* pointing to the syntactic root of the sentence with an edge labeled `root`. From the point of view of querying, this is a regular node that can be matched like any other. However, since anchor nodes do not exist in Corpus Workbench, reported matches diverge every time a query matches the anchor in Grew. To prevent this from happening, we require every token to have a universal part-of-speech annotation, which is absent for the anchor.

7.2. Experimental Results

Whenever there is a mismatch between the number of queries reported by Grew and Corpus Workbench, we look for the underlying cause. As outlined in Section 6, some deviation is expected, as Corpus Workbench sometimes under-reports matches for certain queries. Aside from this, our randomized testing also led us to discover differences in how the two systems execute regular expressions, as well as an undocumented feature in Grew, where `"*"` can be used to match arbitrary values for an annotation.

After excluding queries leading to discrepancies of the latter two kinds, we were left with 4937 queries for evaluation. Table 1 shows the amount of tested queries by the number n of tokens described in the query. Notice how the distribution is not uniform. This is partly due to the generation algorithm preferring smaller queries, but also caused by bigger queries being rejected more frequently by our filtering criteria.

For each value of n, the table shows how many queries were executed, the percentage of the

[8] A query with 9 tokens is generated within 15 seconds on the author's laptop

queries for which both Grew and Corpus Workbench reported the same number of matches and the mean relative deviation (Δ_{rel}), computed as follows:

$$\Delta_{rel} = \frac{1}{N} \sum_{i=1}^{N} \frac{|g_i - c_i|}{g_i}$$

Here, g_i and c_i are the number of matches reported by Grew and Corpus Workbench respectively. This measure describes the deviation between the number of matches reported by Corpus Workbench and the actual number of matches in a corpus.

As expected, both the percentage of queries with diverging number of matches and the extent to which reported matches diverge increase when more tokens are involved. Still, Corpus Workbench finds a majority of the expected matches. The deviation is 8.07% for queries with three tokens and 14.57% for queries with four tokens, which means that Corpus Workbench was able to find more than 90% and 85% of the expected matches respectively. For the limited number of queries with five tokens, all expected matches were found. We want to note here that this experiment using randomized queries presumably over-reports the error rate. Anecdotally, we did not encounter any issues with translations during development and initial testing with linguistically relevant queries.

We also measure the execution times for the translated queries. The main contributor for execution time appears to be query complexity, which follows a factorial relationship with the number of tokens. Figure 2 visualizes the relationship between the number of tokens and the measured execution times, while also including a fitted factorial function as a trend line. The measured execution times vary greatly, which we show by including the 10th and 90th percentile in form of error bars. The cause of this variance is that query complexity varies dramatically between queries with the same number of tokens. Constraints on token order, for example,

Tokens	Queries	Correct [%]	Δ_{rel} [%]
1	2228	99.96	0.00
2	1789	99.78	0.12
3	511	67.47	8.07
4	402	55.47	14.57
5	7	100.00	0.00
All	4937	92.93	2.07

Table 1: Number of queries, percentage of queries where Grew and Corpus Workbench report the same number of matches and mean relative deviation, grouped by number of tokens per query. The last row gives these values over the entire dataset.

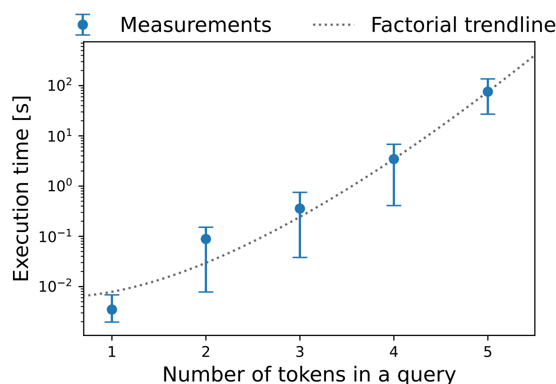

Figure 2: Execution times for translated queries.

drastically reduce the number of token arrangements and therefore the complexity of a query.

Running the queries on the Talbanken corpus, which contains 96820 tokens in 6026 sentences, the execution of queries of up to three tokens terminates within a second. For more complex queries, we can see a drastic increase in execution time, as the amount of checks performed by the query processor exhibits factorial growth.

8. Query language support

CQP/Tree is currently able to translate the query languages Grew-match (Guillaume, 2021), dep_search (Luotolahti et al., 2017) and deptreepy (Ranta and Masciolini, 2025). In all three cases, support was initially limited to word-order agnostic tree queries and some types of token-level annotations and is being steadily expanded.

To facilitate adding support for new query languages, the codebase is structured similarly to that of a programming language compiler. In other words, CQP/Tree consists of a number of *frontends*, each of which parses syntactic queries in a particular language and converts them into an intermediate representation, and of a separate *backend* that translates such intermediate representation into a CQL query. Therefore, adding a new query language only requires implementing an additional frontend, without intervening on the backend. Language support may be partial, as a frontend can report two different error states, one for cases in which a query is invalid and one to signal that it contains (yet) unsupported features.[9]

[9] For the technicalities of adding new frontends, see `github.com/Niklas-Deworetzki/cqp-tree/tree/main/src/cqp_tree/frontends`.

9. Adapting Queries to Sketch Engine

The open source corpus management software NoSketch Engine and the commercial Sketch Engine based on it use a dialect of CQL. This dialect is defined by Manatee (Rychlý, 2007), which is the underlying, shared software component that executes queries in Sketch Engine and NoSketch Engine (from now on abbreviated as (No)Sketch Engine).

First of all, in order to execute CQL queries via the (No)Sketch Engine interface, the user has to navigate to the “Concordance” tool for a corpus, switch to the “Advanced” tab and then select “CQL” as a query type. Figure 3 shows the advanced search interface in NoSketch Engine.

To execute the CQL queries presented in this paper on (No)Sketch Engine, which are designed for the Corpus Workbench dialect of CQL, some adaptions have to be made:

- Instead of using identifiers for labels, Manatee uses numerical labels. Descriptive label names like `aux` and `inf` have to be replaced by `1`, `2`, ...
- Within a token, only expressions in the form `attr="value"` are allowed, where `attr` is one of the encoded annotation layers and `"value"` is an arbitrary regular expression or string. In order to compare the values of two tokens (for example when searching for dependency relations), *global constraints* have to be used, which are written following a `&` operator at the end of a query.
- Distance constraints from Section 5.4 are more complicated to adapt, as Manatee's CQL dialect lacks the `absdist` function. This requires explicitly using empty tokens `[]` to encode distance constraints.

To demonstrate the required adaptations, recall examples (16) and (19). We will now translate them into the dialect required for (No)Sketch Engine. Consequently, examples (25) and (26) are **not** meant to be executed on Korp or Corpus Workbench.

Example (16) requires renaming the labels to numerical values and extracting the comparison representing the dependency edge into a global constraint. The newly introduced labels and the global constraint are highlighted in the adapted query:

(25)
```
1:[] []* 2:[] |
2:[] []* 1:[] &
(1.ref = 2.dephead)
```

Example (19) additionally uses a distance constraint, requiring that the auxiliary and the infinitive are at least two tokens apart. This is translated into

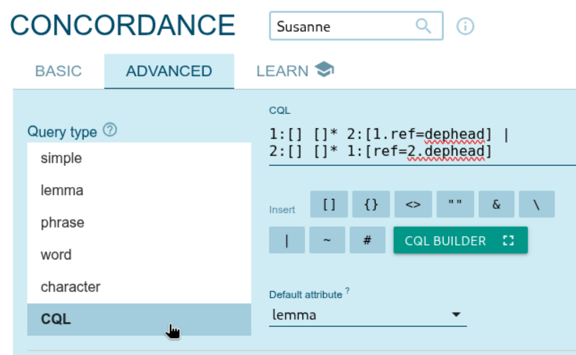

Figure 3: A screenshot of the NoSketch Engine advanced search interface, which allows execution of CQL queries.

`[] [] []*`, a sequence of two or more arbitrary tokens, which is highlighted in the adapted query below:

(26)
```
1:[word="ska"] [] [] []*
  2:[msd="VB\.INF.*"] |
2:[msg="VB\.INF.*"] [] [] []*
  1:[word="ska"] &
(2.dephead=1.ref)
```

10. Conclusions and future work

We presented CQP/TREE, a corpus query converter available as a web application, command-line tool and Python library. Given a query in any of the supported languages (cf. Section 8), it outputs a CQL translation, i.e. a single CQL string or, in cases where the input query needs to be decomposed into multiple steps (cf. Section 5.6), a CQL recipe.

Randomized testing confirms that, aside from minor differences concerning the treatment of wildcards and regular expressions, CQP/TREE can correctly translate a large fragment of the Grew query language. The Corpus Workbench query execution engine, however, causes a certain degree of under-reporting for translated CQL queries describing three or more tokens. Query complexity, and therefore execution time, correlate with the number of tokens involved, which for fully order-agnostic queries leads to factorial growth, but strategically placed order constraints can significantly mitigate combinatorial explosion.

CQP/TREE is under active development. There are ongoing efforts to add integrations with SketchEngine and other CQP-based corpus search platforms, as well as support for graphical tree queries and additional query languages. The codebase is organized to facilitate the latter and we encourage third parties to participate in the development of new frontends.

Acknowledgements

This work has been supported by Språkbanken – jointly funded by its 10 partner institutions and the Swedish Research Council (2025–2028; project id 2023-00161).

11. Bibliographical References

Lars Borin, Markus Forsberg, Martin Hammarstedt, Louise Holmer, and Arild Matsson. 2025. Korp: Språkbanken's word research platform. In *Sixty years of Swedish computational lexicography / Dana Dannélls, Kristian Blensenius and Lars Borin (eds.)*, page 175–193. De Gruyter, Berlin.

Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics*, 47(2):255–308.

Niklas Deworetzki, Peter Ljunglöf, and Nicholas Smallbone. 2024. Towards an algebraic approach for corpus queries. In *Swedish Language Technology Conference (SLTC 2024)*, Linköping, Sweden.

Stefan Evert and Andrew Hardie. 2011. Twenty-first century Corpus Workbench: updating a query architecture for the new millennium. In *Proceedings of the Corpus Linguistics 2011 conference*, Birmingham, Great Britian.

George Fink and Matt Bishop. 1997. Property-based testing: A new approach to testing for assurance. *SIGSOFT Software Engineering Notes*, 22(4):74–80.

Bruno Guillaume. 2021. Graph matching and graph rewriting: GREW tools for corpus exploration, maintenance and conversion. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2021): System Demonstrations*, pages 168–175, Online.

Richard Hamlet. 1994. Random testing. *Encyclopedia of Software Engineering*, 2:971–978.

Adam Kilgarriff, Vít Baisa, Jan Bušta, Miloš Jakubíček, Vojtěch Kovář, Jan Michelfeit, Pavel Rychlý, and Vít Suchomel. 2014. The Sketch Engine. *Lexicography*, 1(1):7–36.

Esther König, Wolfgang Lezius, and Holger Voormann. 2003. *TIGERSearch User's Manual*, v2.1 edition. IMS, University of Stuttgart, Stuttgart, Germany.

Peter Ljunglöf, Nicholas Smallbone, Mijo Thoresson, and Victor Salomonsson. 2024. Binary indexes for optimising corpus queries. In *Proceedings of the 20th Conference on Natural Language Processing (KONVENS 2024)*, pages 149–158, Vienna, Austria.

Juhani Luotolahti, Jenna Kanerva, and Filip Ginter. 2017. Dep_search: Efficient search tool for large dependency parsebanks. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, pages 255–258, Gothenburg, Sweden.

Paul Meurer. 2012. INESS-Search: A search system for LFG (and other) treebanks. In *LFG Online Proceedings*, pages 404–421.

Joakim Nivre, Jens Nilsson, and Johan Hall. 2006. Talbanken05: A Swedish treebank with phrase structure and dependency annotation. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy.

Aarne Ranta and Arianna Masciolini. 2025. deptreepy. deptreepy on GitHub.

Pavel Rychlý. 2007. Manatee/Bonito – a modular corpus manager. In *First Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2007*, pages 65–70, Brno, Czech Republic.

12. Language Resource References

Nivre, Joakim and Smith, Aaron. 2025. *Swedish-Talbanken-UD*. Universal Dependencies Consortium. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL) as part of Universal Dependencies 2.16. PID http://hdl.handle.net/11234/1-5901.