Instantiation and Implementation of a Corpus Query *Lingua Franca*

Master Thesis
by

Joachim Bingel

February 2015

University of Heidelberg

Department of Computational Linguistics

Thesis supervisor	Prof. Dr. Andreas Witt
Second referee	Prof. Dr. Anette Frank

Contents

Abstract

The present thesis introduces KoralQuery, a protocol for the generic representation of queries to linguistic corpora. KoralQuery defines a set of types and operations which serve as abstract representations of linguistic entities and configurations. By combining these types and operations in a nested structure, the protocol may express linguistic structures of arbitrary complexity. It achieves a high degree of neutrality with regard to linguistic theory, as it provides flexible structures that allow for the setting of certain parameters to access several complementing and concurrent sources and layers of annotation on the same textual data.

JSON-LD is used as a serialisation format for KoralQuery, which allows for the well-defined and normalised exchange of linguistic queries between query engines to promote their interoperability. The automatic translation of queries issued in any of three supported query languages to such KoralQuery serialisations is the second main contribution of this thesis. By employing the introduced translation module, query engines may also work independently of particular query languages, as their backend technology may rely entirely on the abstract KoralQuery representations of the queries. Thus, query engines may provide support for several query languages at once without any additional overhead.

The original idea of a general format for the representation of linguistic queries comes from an initiative called *Corpus Query Lingua Franca* (CQLF), whose theoretic backbone and practical considerations are outlined in the first part of this thesis. This part also includes a brief survey of three typologically different corpus query languages, thus demonstrating their wide variety of features and defining the minimal target space of linguistic types and operations to be covered by KoralQuery.

Zusammenfassung

Die vorliegende Arbeit präsentiert KoralQuery, ein Protokoll für die allgemeine Repräsentation von Anfragen an linguistische Korpora. KoralQuery definiert eine Menge von Typen und Operationen, welche als abstrakte Repräsentationen linguistischer Einheiten und Operationen dienen. Das Protokoll ist in der Lage, durch Verschachtelung dieser Typen und Operationen linguistische Strukturen von beliebiger Komplexität auszudrücken. Es erreicht ein hohes Maß an Neutralität in Bezug auf linguistische Theorien, indem es flexible Strukturen bietet, welche mit Hilfe von gewissen Parametern mehrere ergänzende sowie konkurrierende Annotationsebenen auf den selben Textdaten ansprechen können.

Als Serialisierungsformat für KoralQuery wird JSON-LD verwendet, was einen wohl-definierten und normalisierten Austausch linguistischer Anfragen zwischen mehreren Korpusanfragesystemen ermöglicht und somit deren Interoperabilität fördert. Die automatische Übersetzung von Anfragen aus drei konkreten Anfragesprachen in solche KoralQuery-Serialisierungen ist der zweite zentrale Beitrag dieser Arbeit. Die Verwendung des vorgestellten Übersetzungsmoduls ermöglicht Anfragesystemen, unabhängig von bestimmten Anfragesprachen zu arbeiten, da ihre Backend-Technologien lediglich die abstrakten KoralQuery-Repräsentationen der Anfragen interpretieren müssen. Die Anfragesysteme können somit gleichzeitig mehrere Anfragesprachen unterstützen, ohne diese direkt interpretieren zu müssen.

Die ursprüngliche Idee zur Entwicklung eines allgemeinen Formats zur Repräsentation linguistischer Anfragen entspringt einer Initiative mit Namen *Copus Query Lingua Franca* (CQLF), deren theoretischer Hintergrund und praktische Überlegungen im ersten Teil dieser Arbeit wiedergegeben werden. Dieser Teil umfasst ebenfalls eine kurze Studie dreier typologisch verschiedener Korpusanfragesprachen, welche die Mannigfaltigkeit derer Eigenschaften demonstriert und die minimale Zielvorgabe bezüglich der Typen und Operationen definiert, die KoralQuery abdecken muss.

Part I

Corpus Query Lingua Franca

1 Introduction

Text corpus querying systems provide users with access to possibly large collections of linguistic data by means of specific query languages (QLs). A (corpus) QL is a formal language which allows the user to express certain linguistic patterns at various levels of complexity, ranging from single word forms to nested structures of higher-level linguistic phenomena such as syntactic constituents or anaphoric relations. Upon the submission of a query to a system, its engine tries to interpret the query and searches its stored linguistic data for all instances of text that match the pattern expressed in the query, potentially presenting the user with all passages of text that were found, whole documents in which the specified pattern is present, or occurrence statistics, depending on the specific system or user demands.

Just like there is a large number of existing corpus querying systems, the amount of query languages is too great to keep track of. However, a first glance at only a few of them reveals a huge diversity in terms of a number of qualities. Perhaps most evidently, QLs differ in their respective *syntax*. In this respect, a particular QL is often, at least unconsciously, influenced by existing query languages.[1] While lines of tradition in the design and implementation of the syntax of query languages may have established certain QL families, there are still many isolated languages, and even within the less exotic languages, syntactic idiosyncrasies are abundant. The syntax of a query language certainly has a great impact on another quality which is highly diverse across languages, namely its *usability*, which relates to more subjective categories such as the ease by which a language can be learnt or the verbosity of language elements. While some languages are very straightforward in these respects, others comprise a rather vast amount of various operators and thus need more time to learn and type. Naturally, the complexity and verbosity of a language increase with its *expressive power*, which denotes the set of linguistic patterns that can be expressed in this language and is therefore probably the most important fac-

[1]Those need not necessarily be languages that are specifically designed for querying text corpora, but may, for instance, be languages that are used to access and manipulate more structured information in a database, e.g. SQL.

tor that determines the quality of a QL. As with the previous categories, languages differ very strongly in their respective expressiveness. Some only allow very shallow patterns to be searched for, such as the combinations of certain word forms or words bearing certain grammatical values, while languages on the other end of the spectrum can express very complex (meta-) linguistic structures.

The reasons for this profusion of querying systems and query languages, as well as their large qualitative differences, are manifold. Most centrally, they are often custom-tailored to be capable of providing answers to rather specific types of linguistic questions, or at least assume certain types of linguistic annotation. They are thus usually developed with respect to particular linguistic phenomena and to a specific data format. Furthermore, the fact that the development of corpus query tools and languages is subject to economic limitations, and that it is often grounded within isolated projects with uncoordinated funding, has impeded the investment of additional work to cover use cases that would exceed the specific needs within a project.

As a result of the depicted variety of query languages and their properties, users as well as developers of corpus querying tools are faced with obvious difficulties. Users may be forced to learn a new query language whenever they work with an unfamiliar system, which in turn may be necessary because of new research questions they investigate or simply because they turn their attention to other data that is not accessible through their accustomed system. The requirement to learn a new QL consumes the researchers' time and energy and may cause frustration to a degree that they feel uncomfortable using a particular system. Developers, on the other hand, face the challenge of having to analyse whether or not they are able to use some existing QL for their newly developed querying tool, which means that they first need to define a set of requirements a QL needs to satisfy (mostly relating to expressiveness, but also to performance issues) in order to support their particular case, and then look into a lot of languages and decide for each language whether it meets the requirements or not. Then, in case a suitable QL has been identified, developers have to take measurements to deploy the language in their search engine. This is by no means trivial, as an engine is usually expected to search a potentially very large amount of data in very short time, and attention must be paid to how certain elements of a QL increase the workload of the engine backend. If, however, the search for such a language has been unsuccessful, a new language must be created, which possibly amounts to a great deal of work involving various steps, including the definition of available operators to support all requirements in terms of expressiveness while paying attention to questions of computational performance, the definition of a vocabulary and syntax, and as in the first case, the deployment in the query engine. Eventually, this will result in yet another custom-tailored QL and only increase the sketched problem rather than solve it.

The present thesis faces these issues within the framework of the Corpus Query Lingua Franca (CQLF) endeavour and proposes a specification of a concrete CQLF instantiation, in particular the definition of a general and flexible data format for representing queries that are formulated in any of (currently) three different QLs representing different QL families. As a proof-of-concept of this specification, translation modules are provided that map concrete queries from those QLs onto the general format. The remainder of this document is structured as follows. The sections on related work and the KorAP project introduce the reader to the theoretical and practical background of this work, and the introductory part will be concluded by an outlook on the aim of the thesis. The second part will identify some caveats and prerequisites that underlie the construction of a corpus query 'metalanguage', and will specify its concrete instantiation. The third part introduces a serialised data format for the representation of queries in the meta format, as well as the translation modules that are used to map user-defined queries onto that format. The thesis concludes by outlining its central contributions and proposing items of future work.

1.1 Related work

The apparent problems that have been briefly addressed in the introductory section, and will be further explicated in the first part of this thesis, have been known to the corpus linguistic community for quite some time. In a meeting of scholars from the humanities, library science and information technology, which took place in November 2010, the participants debated those issues and discussed the possibility of proposing the development of a corpus query *lingua franca* to the International Standardization Organization (ISO). The report (Mueller, 2010) sketches the two possible outcomes of such a proposal and their consequences: "If it leads to a standard it will make development easier. If it does not lead to a standard it may at least help articulate the points where interoperability becomes difficult or breaks down."

Inspired by this suggestion, the work on CQLF was commenced within the ISO

TC37 SC4 WG6[2] and currently has the status of a Working Draft. While the original idea of CQLF, as it is suggested in Mueller (2010), revolves around the development of an interlingua that enables communication across different QLs, the ISO committee has from the outset been aware of the second scenario mentioned in Mueller's report (that this work would not lead to a standard) and of the great amount of difficulties that come with the intended creation of such an interlingua. Therefore, the preliminary goal of the CQLF proposal is (i) to serve as a metric for query languages to be compared to each other with respect to certain properties and (ii) as a system of target features that may be considered in the creation of new corpus query languages (see Section 3 for a more detailed description of this aspect). The aspect of translation between QLs (and thereby interoperability between query systems), however, has not been an official part of the work on CQLF, but has instead been pursued within the KorAP project (Section 1.2).

Nevertheless, research on the interoperability between corpus query systems is no unknown territory, although most previous work has tried to achieve such interoperability on the level of the data rather than from the query language point-of-view. Also within the ISO TC37 SC4, standards have been developed that ensure common data formats or format frameworks. Most notably, the Linguistic Annotation Framework (LAF, ISO 24612:2012) provides an abstract model for the general graph-based representation of a wide range of linguistic objects and relationships, which lays the conceptual foundation for the mutual translation between different annotation encoding schemes via mapping to and from a pivot format. As an extension to and a concrete serialisation format of LAF, Ide and Suderman (2007) introduced the Graph-based Annotation Format (GrAF), which encodes the linguistic information as defined by the LAF model in XML.

While LAF and related annotation standards such as the Morpho-syntactic Annotation Framework (MAF, ISO 24611:2010) and the Syntactic Annotation Framework (SynAF, ISO 24615:2010) largely remain in the abstract and, at best, are instantiated by concrete formats such as GrAF and <tiger2/> (Bosch et al., 2012), they do not actually provide facilities for the mapping between language resources (Zipser and Romary, 2010). This gap is filled by the SaltNPepper software (Zipser, 2009), a suite of Java classes which use an internal meta-model (Salt) and a converter framework (Pepper) to allow the development of mappers to and from specific formats.

[2]ISO Technical Committee 37 (Terminology and Other Language Resources), Subcommittee 4 (Language Resource Management), Working Group 6

By means of the intermediate pivot model, a complete mutual translation between n formats requires only $2n$ mapping components (one mapper to and one mapper from the meta-model for every format), rather than n^2 mappers that would be needed for direct translations between all formats.

1.2 The KorAP project

In July 2011, the *Institut für Deutsche Sprache* (IDS) in Mannheim, Germany, commenced work on the KorAP[3] project, which was initially granted funding for three years but was later extended until mid-2015. The objective of the project has been to develop a modern corpus analysis software to succeed COSMAS II (Bodmer, 2005) as the user interface to the German reference corpus DEREKO (Kupietz et al., 2010), one of the most important resources for German linguistic research with currently more than 20,000 registered users of COSMAS II. However, implemented almost twenty years ago, COSMAS II has by now become very difficult to maintain and adapt to the growing demands of corpus analysis. Those demands are of two kinds. Firstly, with corpus-based methods gaining more and more importance in present-day linguistic research (Lüdeling and Kytö, 2008), user expectations with respect to the functionality and usability of a query engine grow. Depending on their particular research questions or level of proficiency in corpus linguistics, researchers will expect features such as a co-occurrence analysis function, highlighting and grouping options, or scripting interfaces. Secondly, as ever larger corpora are being compiled or existing corpora are growing in size, computational costs are becoming more and more drastic and ultimately unbearable for outdated corpus query software. In particular, the inflation of DEREKO from over six billion tokens to more than 24 billion tokens within a single year (Kupietz and Lüngen, 2014) cannot be efficiently mastered by COSMAS II.

In response to those demands, the current project defines a range of requirements to be met by the KorAP software as a state-of-the-art corpus query tool that is expected to face the challenges of present-day corpus analysis and to be prepared for upcoming ones (Bański et al., 2012). Besides the ability to scale to primary text data in the petabyte range, complex annotations and relationships between data points must be available for querying and display, which further aggravates computational complexity. Furthermore, in order to provide researchers with a maximally undis-

[3]"KorAP" is short for *Korpusanalyseplattform der nächsten Generation* ("Next-generation corpus analysis platform").

torted and theory-neutral view of the data, KorAP is designed to allow concurrent annotation[4] from different sources on the same piece of primary text. This particular principle is followed by separating primary text and annotations by means of so-called standoff mark-up, where position indices are used to declare a certain annotation information to hold at some place in the primary data, in the case of text usually via character offsets. Another advantage of applying standoff mark-up techniques is that the annotations can be treated in an (almost) equivalent manner as the primary data itself with regard to searching. In order to retrieve text passages that correspond to a specified query, KorAP employs the open-source information retrieval software Lucene[5], which builds an index of the data (both primary data and standoff annotations) for a fast lookup of the linguistic patterns defined in the query. Thus, rather than loading all the corpus data into memory (which some query engines do, but which is utterly impossible for large corpora), KorAP uses widespread information retrieval techniques to manage its data.

The development of KorAP follows the principle of modularity, which will enable the developers to react flexibly to possible demands that will arise in the future. For example, integrating new modules into the software could enable it to index and search data in other modalities, such as the *Archiv für Gesprochenes Deutsch* ("Archive for Spoken German", AGD, cf. Fiehler et al. (2007)). Another extension, which is currently under development, is the employment of an alternative Neo4j backend that can be used in place of the Lucene index by request of the user.[6]

Another challenge lies in legal restrictions that many corpora or partial corpora are subject to. In the case of DeReKo, a big chunk of the texts (especially newspaper texts) is available to the IDS only through contracts with copyright holders that restrict the access to the data for the end user. Additionally, different restrictions might hold for different user groups, e.g. researches affiliated with certain institutions. KorAP thus implements a mechanism to ensure that no unavoidable legal restrictions are violated while providing individual users with maximally possible access to the data.

[4]In the remainder of this work, concurrent annotation is defined as (several layers of) annotation that encodes the same kind of information (e.g. the same grammatical category) for the same data stream, but comes from different sources and is therefore *possibly* conflicting. For a more formal definition, see (Dekhtyar and Iacob, 2005).

[5]http://lucene.apache.org/

[6]The rationale behind providing this alternative is that the graph-based Neo4j software may scale better to particular types of data. For instance, queries that ask for patterns involving syntactic constituency relations may be better processed in a graph than a "linear" index, and that Neo4j would outperform the Lucene backend in that case.

Finally, it is targeted to release the KorAP software under an open-source license by the time of its completion. This will allow corpus linguists to deploy (custom modifications of) KorAP to provide access to their own corpora and thus to profit from the efforts that have been made in order to meet the describes challenges.

One system - several QLs

The mentioned diversity of QLs in terms of expressiveness as well as their usually problem-specific design mean that there are problems that can only be solved by means of certain QLs. This implies that the decision for one particular language to be supported by a corpus query engine drastically limits the spectrum of research questions that the engine is capable of answering. While this problem may not be critical for most corpus query applications, DEREKO's principle of theory-neutrality and its purpose to serve as a "primordial sample from which virtual corpora can be drawn for the specific purposes of individual studies" (Kupietz et al., 2010) makes such specificity very undesirable for KorAP. In fact, the very general KorAP annotation format (cf. the section on the KorAP data model below) virtually discourages any specific tailoring of a QL towards a particular data model.

In the early stages of the KorAP project, the developers thus made a decision not only to support one specific (existing or newly created) QL, but rather to let the users choose from a set of languages the one that they believe is best suited with regard to their particular research questions and, of course, their familiarity with the individual languages. Through this decision, researchers find themselves presented with a much greater flexibility, but also with more responsibility in the design of their corpus research. The possibility to choose from different QLs and therewith from different dimensions of expressiveness demands careful planning on the part of the user. Finally, the initial repertoire of QLs to by served by KorAP was decided to include Poliqarp QL (Przepiórkowski et al., 2004), COSMAS II QL (Bodmer, 1996) and ANNIS QL (Rosenfeld, 2010), but the possibility to include a theoretically unlimited amount of QLs at a later point is provided.

The support of several QLs naturally leads to higher requirements of the system. In the common case, a query engine backend directly parses and interprets the various parts of a query, and then attempts to retrieve text records for those query parts and finally for the entire query. As the development of the backend technology is probably the most critical and cost-intensive work item within the construction of a query engine, creating separate backends for the individual QLs is obviously not a promising strategy. Besides the plain expenditure for the creation of additional backends, a lot of attention would need to be paid to ensure a total equivalence of the various backends in terms of the results they return for equivalent queries formulated in different QLs.

As a solution to these problems, KorAP separates its backend technology from the interpretation of the available query languages. This is achieved by making the backends[7] work independently of a particular query language, i.e. that they do not directly process the query string itself, but rather some general and language-independent representation of the query. The development of such a general representation of queries, as well as the implementation of translation modules that convert queries formulated in a certain language into that abstract format, is the core of the present thesis and a major work item within the KorAP project and CQLF, such that KorAP is in fact a driving force and a reference implementation of CQLF.

The KorAP data model

The ideal of achieving a maximal degree of theory-neutrality in KorAP has two major dimensions. Firstly, KorAP aspires to allow querying for arbitrary types of linguistic annotation as far as permitted by available state-of-the-art text processing systems. Secondly, the user shall be enabled to perform those searches in arbitrary collections of documents as defined by freely adjustable parameters such as document metadata.[8]

These requirements call for a highly flexible and unrestricted data model that allows for the unproblematic addition or modification of corpus data whenever, for instance, new texts are to be included in a corpus or new annotations may be made available. The most efficient and most elegant way to meet these requirements is to structure all data in a modular fashion. At the level of corpus structuring, this is achieved quite straightforwardly by separating all documents and placing them as individual entities within one directory that corresponds to a (sub-) corpus. The more challenging aspect, however, is the modularisation of document-level annotations.

[7]Remember that the modularity principle in KorAP also applies to the backend technology. Rather than deploying a single backend, KorAP reserves the possibility to incorporate several backends into a single system and to let the user or a query optimisation component decide which backend to use for a particular type of query.

[8]For example, a user may restrict the set of searched texts to newspaper articles on politics published between 2000 and 2010.

It is important to note that a modular encoding of linguistic annotation is not merely a matter of elegance and efficiency, but utterly necessary when several concurring annotation layers need to be made available for a single text. In traditional SGML- or XML-based text encoding schemes such as the (X)CES standards (Ide, 1998; Ide et al., 2000), concurring annotation is highly problematic because of non-hierarchical and possibly overlapping (and thus conflicting) entities. Research to overcome these problems has lead to the proposal of three major groups of annotation formats and mechanisms: architectures employing Prolog-style fact bases, XML-related formats and graph-based formats following the XML syntax (Stührenberg and Goecke, 2008). Especially the latter group of architectures, which were initiated by the Annotation Graph model (Bird and Liberman, 1999) and include the GrAF framework (Ide and Suderman, 2007), have proven to be helpful points of reference for the encoding of several annotation layers in the KorAP data model. The common denominator of those graph-based architectures is the separation of primary text and markup. Called *stand-off annotation*, this encoding principle uses positional indices of the primary data (usually via character offsets in the case of text or time anchors for audio/video data) to declare annotations of all kinds at particular spans within the data stream (e.g. words or phrases) or between various spans of data (e.g. pointing relations between two phrases).

Beyond their primary text and linguistic annotation, documents in KorAP are equipped with metadata (e.g. title, authorship, publication date, text genre etc.), which allows for a user-side definition of virtual document collections as the search space of a linguistic query. This metadata is also encoded in a separate file and thus detached from the primary data.

As a result, the KorAP data model is the following: a corpus (which might be recursively composed of sub-corpora) consists of documents that are represented by directories. Each of these directories contains a text file of primary data (the text), a text file of metadata information, and a set of directories grouping files of linguistic annotation layers. There are typically several such directories, because annotations may come from different processors each of which contributes its own set of annotation layers (e.g. tokenisation, morphosyntactic tagging or constituent parsing). The document structure is illustrated in figure 1.1.

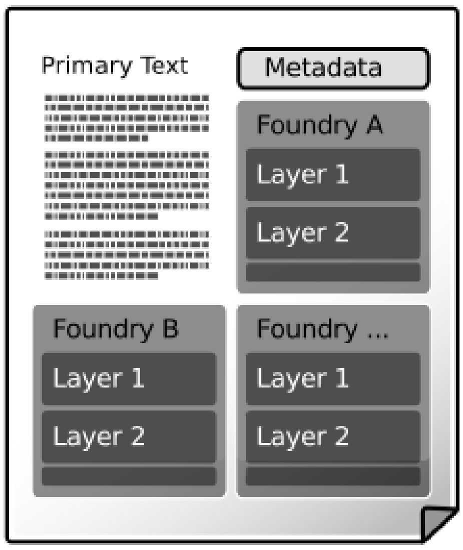

Figure 1.1: The KorAP document model.
Image source: (Bański et al., 2013)

1.3 Aim of the thesis

This thesis has, up to this point, sketched several problems in the field of corpus linguistics and around the design of corpus query systems. Against the backdrop of those problems, specifically those regarding the support of various query types and query languages in KorAP as well as the interoperability of query systems in general, it appears worthwhile to construct a maximally flexible and general representation of linguistic queries and thus to arrive at some sort of interlingua, very much in the spirit of the original vision of a corpus query lingua franca as formulated by Mueller (2010).

To realise this vision, the present work first provides an overview of some query languages, representing different QL families, and their features. This set of features informs the definition of the array of linguistic and meta-linguistic objects and relations that need to be included in the aspired lingua franca. The thesis then aims to develop a specification of a *grammar* for CQLF, i.e. a set of rules that determine the structure of the respective CQLF objects and how they can be combined with others. Based on this specification, a concrete JSON-LD-based *serialisation* format will be proposed.

Finally, this work includes the development of individual serialisation *processors*

Feature Group	Poliqarp	COSMAS II	ANNIS
Plain-text search	✓	✓	✓
Annotation-based search	(✓)	(✓)	✓
Constraints on match size and position	✓	(✓)	✓
Boolean operators	(✓)	(✓)	(✓)
Universal quantification and implication			
Fuzzy and pattern search	(✓)	(✓)	(✓)
Metadata access	(✓)	(✓)	(✓)
Display directives	(✓)	(✓)	(✓)

Table 2.1: Support of feature groups by the QLs examined by Frick et al. (2012). Check marks in parentheses indicate the support of a subset of the features in the respective feature group.

that translate a query formulated in any of three query languages into the common CQLF format.

2 Corpus query languages: A brief survey

As a prerequisite to arrive at a clearer picture of the concrete requirements that a CQLF must meet both as a metric and as a metalanguage (cf. sections 3 and 4, respectively), this section investigates the specificities of three existing query languages. A pilot study by Frick et al. (2012) evaluated Poliqarp QL, COSMAS II QL and ANNIS QL against a set of functional criteria that were derived from a multitude of actual user queries covering diverse linguistic phenomena. Thus, rather than examining a great amount of QLs under very few aspects, the authors chose to scrutinise the three mentioned languages in great detail. The QLs are regarded as representatives of distinct QL families, stemming from different traditions and being tailored towards very different data models, and may therefore be expected to be quite complementary in their syntax and expressive power.

The evaluation criteria were grouped into the eight categories ('feature groups') that form the first column of Table 2.1. We observe that only few feature groups are really fully supported by any of the languages, in fact the language that implements all features for most groups is ANNIS QL, with a full support of only three groups. Note, however, that the indication of partial support (by check marks in parantheses) is quite an oversimplification, as sometimes all but one minor feature in a group are supported by a certain language, and the language thus receives the same coarse rating for the category as another language that only supports one feature. The reader is therefore referred to the original publication for a fine-grained analysis and a detailed description of the features and feature groups.

The authors conclude in their study that the languages indeed differ in terms of their functionality, and that some query types (e.g. directed relations) are quite specific to certain languages, while others appear to be rather universal. Importantly, none of the examined languages displays a total coverage of the authors' criteria. In particular, some features such as negation are only supported to a certain degree, while others such as universal quantification ("find a sentence containing verbs in the first person singular only") are not supported at all. Nevertheless, a combination of the QLs (i.e. a common metalanguage) would in fact pose quite a powerful language, and covering a relatively wide array of QL features, they constitute an informative sample for the development of a corpus query lingua franca. The remainder of this section introduces the individual QLs in more detail, both in respect to their feature sets and to their syntax by way of examples.

2.1 Poliqarp QL

Poliqarp QL is a query language that is based on the syntax of the CQP query language (Evert, 2005) and has been developed to query corpora indexed with the Poliqarp tool (Przepiórkowski et al., 2004), which in turn has been created for the IPI PAN corpus and was later used to provide access to the National Corpus of Polish.

The language is designed to query for (sequences of) tokens. Consequently, a valid query in Poliqarp QL is a string of segments that correspond to individual tokens, where a token is expressed as a pair of brackets containing a set of constraints (attribute-value-pairs) that the matched tokens satisfy. The constraint set may be empty (resulting in an underspecified 'match-any' token) or predicated by (recursive) logical operations on the individual constraints (conjunction, disjunction and negation). A controlled vocabulary of attributes is available for querying the IPI PAN corpus and comprises orthographic forms via the keyword `orth`, lemma forms via `base`, parts-of-speech via `pos` and several grammatical categories such as `number`, `case`, `gender` etc. The value constraints of those attributes may be specified as plain strings or regular expressions. Additionally, the user may indicate by means of specific flags whether the value shall be interpreted sensitive to orthographic case (`/i`) or if regular expressions are allowed to match only parts of a word (`/x`).

By means of repetition operators (minimal and maximum occurrences in braces or Kleene quantifiers), users can express constraints on the repeated occurrence of tokens and sequences. In combination with underspecified tokens, a user can also define distance constraints between two tokens. To express that a certain sequence may not exceed sentence or paragraph boundaries, the keywords `within (s|p)` may be used. The following are thus examples of valid Poliqarp queries:

(1) `[base=corpus]`
All tokens with the lemma form *corpus*.

(2) `[orth=corpus]`
All tokens with the surface form *corpus*.

(3) `[orth=corpus/i]`
All tokens with the surface form *corpus*, ignoring case.

(4) `[orth=queries & pos=V]`
All tokens with the surface form *queries* that are verbs (3rd person singular of *to query*).

(5) `[base=corpus & orth!=corpus]`
All tokens with the lemma *corpus*, but with a different surface form.

(6) `[orth=”corp.*”/xi]`
All tokens that contain a match of the regular expression *corp.** (regex indicated by quotes).

(7) `[orth=corpus][orth=query]`
All sequences of the tokens *corpus* and *query*.

(8) `[pos=”JJ.*”]+[orth=query]`
All sequences of at least one adjective followed by the word *query*.

(9) `[orth=corpus][]{2,5}[orth=query]`
All sequences containing the tokens *corpus* and *query*, with a distance of two to five other tokens in between.

(10) `[orth=corpus][]{2,5}[orth=query] within s`
As above, restricted to sequences within the same sentence.

Poliqarp QL also allows the specification of display directives (via the alignment operator ^) and the restriction of the search space to texts that satisfy certain metadata constraints, such as authorship, publication date or text genre (using the `meta` keyword). The following examples illustrate these operators:

(11) `[orth=corpus]^[]{2,5}[orth=query]`
As (9), aligns results after *corpus*.

(12) `[orth=corpus] meta published>2000`
As (2), restricted to texts published later than 2000.

Additionally, Poliqarp QL provides operators that concern syncretisms in inflectional forms. This is desirable, for instance, for the retrieval of some Polish pronouns, where the respective case cannot always be disambiguated on the syntactic level. By means of the operator `==`, a user can require that only clearly unambiguous matches of a query (e.g. `[case==acc]`) are returned.

The language lacks support for the explicit retrieval of syntactic constituents as well as hierarchical relations between them. Neither can dependencies or other token-level relations (e.g. anaphoric relations) be expressed. Furthermore, the language relies on a data model that features a single layer of annotation, i.e. one cannot choose between several concurrent annotation sources.

2.2 COSMAS II QL

The COSMAS (Belica et al., 1992) corpus search engine, as well as its successor COSMAS II (Bodmer, 2005), have been developed at the IDS, mainly to provide access to the German reference corpus DeReKo (Kupietz et al., 2010). The query language that this system employs, COSMAS II QL (Bodmer, 1996), is characterised by fine-grained options on distance and positional constraints between two query segments.

In particular, the language features a distance operator which requires that a certain number of words, sentence or paragraph boundaries (denoted by `w`, `s` and `p`) occur between the arguments of the operator. It is also possible to negate those distance constraints, such that the query returns all instances of the first argument that do not co-occur with the second argument within the specified distance.

The operators `#IN` (inclusion) and `#OV` (overlap) express positional relations between their operands, e.g. requiring that the span denoted by the second argument is fully included in the span that is matched by the first argument. The inclusion/overlap constraint can be further explicated by way of several options, such as `#IN(L)`, which demands that the arguments are left-aligned, i.e. that the start offsets of the operands are identical. Similarly to the distance operator, positional constrains can be negated, again returning only those contexts of the first argument that are not connected with any match for the second argument by the respective positional relation.

The language relies on data in XML format, such that XML elements can be retrieved via the `#ELEM()` operator, which takes the desired element tag as its argument. In the COSMAS II data, this markup is restricted to sentences and paragraphs (as far as linguistic information is concerned), but in theory, complex constituency trees could be annotated in the XML format and thus be retrievable via COSMAS II QL.

The possibility to query for morphosyntactic information is provided by the `MORPH()` operator, which takes as its arguments one or more (possibly negated) values that are interpreted as belonging to certain grammatical attributes. For example, the user may apply this operator to retrieve tokens of a specific part-of-speech class or all tokens that are in superlative comparison (whether they are adjectives or adverbs). The respective tag sets stem from one of three available annotated archives, which the user must select at the beginning of a session if they need access to morphosyntactic annotation. Further possibilities to retrieve tokens other than by their surface form are case-insensitive search and querying for the lemma form.

Additionally, COSMAS II comprises a number of match-modifying possibilities, e.g. the `#ALL()` operator, which extends the match area of a distance query to all tokens in between the arguments, rather than only the arguments, which is the default for distance queries. Other examples include `#EXCL()`, which works inversely to `#ALL()` and only matches the tokens between the operands of a distance query, or `#BEG()` and `#END()` which only match the first or last token of a sequence, respectively.

The following are examples of COSMAS II QL queries and illustrate its expressiveness:

(13) `corpus`
All tokens with the surface form *corpus*

(14) `corpus /w5 query`

All sequences of the tokens *corpus* and *query*, with a distance of at most five words (four words in between)

(15) `corpus /+w5 query`
As above, but with a directional constraint (*corpus* followed by *query*, not the other way around).

(16) `(corpus /+w5 query) #IN #ELEM(S)`
As above, but sequence must be within one `S` element (sentence)

(17) `(corpus /+w5 query) #IN(L) #ELEM(S)`
As above, but sequence must be left-aligned in the sentence.

(18) `(corpus /+w5 query) #OV #ELEM(S)`
The sequence overlaps[9] with an `S` element.

(19) `MORPH(N) /+w1:1 query`
All sequences of any noun directly followed by the token *query*.

(20) `MORPH(N pl) /+w1:1 query`
All sequences of any plural noun directly followed by the token *query*.

2.3 ANNIS QL

The ANNIS platform is a corpus search and visualisation tool which has been developed to explore the data of a collaborative research centre dedicated to information structure.[10] This data is annotated at various linguistic levels, ranging from morpheme segmentation to hierarchical relationships building on Rhetorical Structure Theory (Mann and Thompson, 1988). In order to ensure maximal flexibility in indexing the data, the ANNIS developers have opted to take a graph-based approach, i.e. to represent all linguistic entities in a text (i.e. tokens, phrases etc.) as *nodes* that are connected via *edges*, which in turn represent linguistic relations between the

[9]In COSMAS-II, two sub-spans are defined to 'overlap' iff the intersection of those sub-matches is not empty. Note that, in the case of partial matches (e.g. a distance query with two tokens where only those tokens form the match, not the whole span including the words between them), this constraint is already satisfied when one token occurs in both sub-query matches, regardless of their positional relation.

[10]The Collaborative Research Centre 632 is funded by the German Research Foundation (DFG) and is located at the University of Potsdam, the Humboldt University of Berlin and the Free University of Berlin. More information can be found at the centre's website: http://www.sfb632.uni-potsdam.de/.

nodes, e.g. precedence, constituency relations, dependency relations or information structural relations, among others.

This graph-based annotation format is directly reflected in the syntax of the query language (Rosenfeld, 2010), which differs drastically from those of Poliqarp QL or COSMAS II-QL. Rather than hierarchically embedding sub-queries in higher-level structures, a query in ANNIS QL consists of a sequence of node declarations and assertions about the nodes themselves or relations between two nodes. Those declarations are connected via a conjunction operator (`&`).

Nodes may be declared using a set of keywords (e.g. `tok` for a token or `node` for a generic node) or directly expressing certain linguistic constraints for retrieving named entities that must hold for the returned nodes (basically unary relations, e.g. `pos="NE"`). To formulate binary relations between the declared nodes, those can be referenced using numerical values that indicate the order in which they have been declared (e.g. `#1` for the node that was declared first in the query string).

ANNIS QL offers a high flexibility in accessing different layers of annotation by parametrising its operators, most notably the operators `>` (hierarchical relation between two nodes) and `->` ('pointing' relation between two nodes). For instance, to express a dependency relation between two nodes A and B, the user may state the relation `A ->dep B`, while the relation `A ->coref B` demands coreference between A and B. Note that the parameters `dep` and `coref` are not a part of the query language proper, but indicate specific annotation layers in the data which carry those names, such that the query language is unaware of the actual meaning of the annotation, but still capable of accessing it by a generic operator. Those parameters may also be refined by expressing attributes and values that must hold for the respective relations, e.g. `A ->dep[func="obj"] B` requires that the edge between the nodes is labelled with an object function. Of course, the query language is agnostic to the available set of attributes and values of those labels as well, which instead depend solely on the annotations. Additionally, it is possible to negate attribute-value statements (both for edge labels and node properties) such that all instances are returned that do not satisfy the respective constraint. All string values may be also specified as regular expressions, which are delimited by forwards slashes in the language.

Besides those generic operators for linguistic relations between nodes, ANNIS QL can directly express precedence constraints using the `.` (dot) operator. This operator, as well as `>` and `->`, may even be quantified using a Kleene star or numerical ranges, such that indirect (precedence) relations can be specified. Operators that express positional relations between spans of text are also included in the language and comprise exact matching, inclusion, overlap and left/right alignment. ANNIS QL also provides operators that draw on the tree structure of hierarchical relations, expressing that one node is the leftmost or rightmost child of another node, that two nodes share the same parent/ancestor node, that nodes govern a specific number of direct children or leaf nodes (tokens), or that a node is the root of a tree.

Furthermore, ANNIS QL supports the formulation of metalinguistic criteria, for instance the language of a text, its publication date or genre, using the `meta` keyword. Examples 21-35 illustrate the presented operators and the ANNIS QL syntax.

(21) `tok="corpus"`
All tokens with the surface form *corpus*

(22) `tok="corpus" | tok="query"`
All tokens with the surface form *corpus* or *query*

(23) `pos=/A.*/`
All tokens whose part-of-speech tag matches the regular expression *A.**

(24) `tok="corpus" & tok="query" & #1 .  #2`
The token *corpus* immediately followed by *query*.

(25) `tok="corpus" .  tok="query"`
Shortcut: the token *corpus* immediately followed by *query*.

(26) `tok="corpus" & tok="query" & (#1 .  #2 | #2 .  #1)`
The token *corpus* immediately followed by *query*, or vice versa.

(27) `tok="corpus" & tok="query" & #1 .2,5 #2`
The token *corpus*, followed by *query* in a window of 2 to 5 words.

(28) `cat="NP" & tok="corpus" & #1 >* #2`
A noun phrase that (possibly indirectly) governs the token *corpus*.

(29) `node & pos=/VA.+/ & #2 ->dep[func="sbj"] #1`
Any (generic) node that serves as the subject of an auxiliary verb.

(30) `node & pos=/VA.+/ & cat="NP" & #2 ->dep[func="sbj"] #1 & #1 ->coref #3`
Any (generic) node that serves as the subject of an auxiliary verb, and is coreferent with another NP.

(31) `cat="S" & cat="PP" & #1 _l_ #2`
All sentences that begin with a prepositional phrase (the PP is left-aligned in S).

(32) `cat="PP" & cat="PP" & #1 $ #2`
Two prepositional phrases that share the same parent node.

(33) `cat="NP" & cat="VP" & cat="PP" & #1 > #2 & #2 . #3`
Noun phrases that dominate a verb phrase that is directly followed by a prepositional phrase.

(34) `cat="NP" & #1:arity=3`
All noun phrases with exactly three children.

(35) `cat="NP" & #1:arity=3 & meta::pubDate="2000"`
As above, restricted to texts published in 2000.

2.4 Some other query languages

Beyond the three presented languages for corpus querying, there is an abundance of corpus QLs in use in the community. Those languages vary strongly in their syntax, expressiveness and purpose. A rather basic language that is largely intended for the retrieval of single words or word sequences is the Contextual Query Language (OASIS, 2013). This language is mainly used for information retrieval purposes in bibliographic contexts and thus encompasses methods for querying based on bibliographic metadata.

The language that is used for the Corpus Query Processor (Christ et al., 1999), called the CQP Query Language (Evert, 2005), is targeted towards querying segmental information, similar to Poliqarp QL. It contrasts with Poliqarp QL in that it provides a host of displaying directive and result processing functions and allows for a considerably higher flexibility in defining macros.

Another query language that, similar to ANNIS QL, is designed to query syntax graphs, is the TIGER language (König and Lezius, 2003). It strongly resembles ANNIS QL also in its syntax, but interestingly, it works on a very different backend technology as it loads the full XML-encoded corpus into memory rather than using a relational database.

There exists also a fair amount of QLs that are not primarily designed for querying corpora of written text, but speech corpora for the purpose of phonetic analysis. Those comprise the Q4M language developed in the MATE project (Heid and Mengel, 1999) or the Emu language (Cassidy and Harrington, 1996), which is, quite notably, capable of querying concurrent layers of annotation on the same primary data (Cassidy and Harrington, 2001), cf. also (Bird et al., 2000).

3 CQLF as an evaluation ground for query languages

As stated in the introductory section, the immediate goal of the work on CQLF has been to provide some kind of metric that can be used to compare and evaluate corpus query languages, much like in Frick et al. (2012). The establishment of a standardised metric with well-defined criteria of query language expressiveness would bring apparent benefits to such evaluations, including mutual comparison and sparing researchers the effort of having to define their own feature space against which to evaluate a certain query language .

Up to this point, the development of CQLF as a metric has been pursued in two separate, but interrelated stages. The first stage has defined an abstract meta model, while the second stage has involved the definition of a more fine-grained set of concrete QL features. These two parts of CQLF are sketched in the following two sections.

3.1 Meta model

CQLF distinguishes three basic levels of analysis, corresponding to different degrees of complexity or different 'feature groups'. As a consequence, the CQLF meta model comprises three *Levels* that are defined as follows:

Level 1 (linear) At the most elementary level of complexity, a QL allows the formulation of queries for segmental annotations, i.e. the (relative) linear placement of words in text.

Level 2 (complex) Languages qualifying for Level 2 can query hierarchical structures, i.e. linguistic units that are connected through hierarchical relations such as dominance (in constituency grammar) or dependencies.[11]

[11]For a definition of the hierarchical dominance relation between spans and a demarcation from

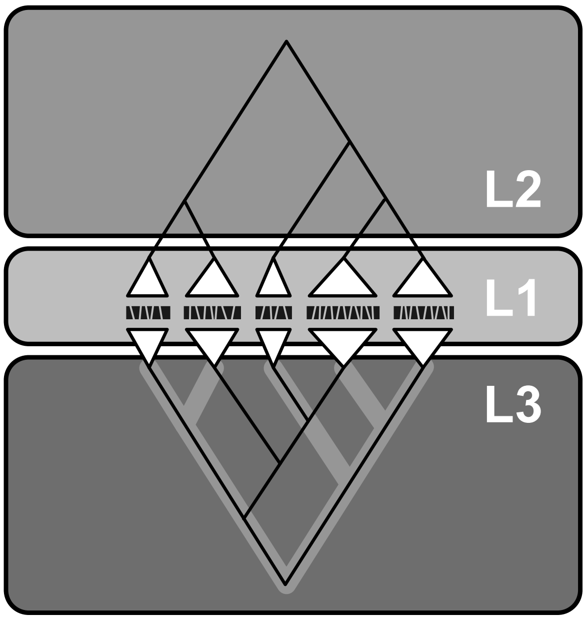

Figure 3.1: Schematic illustration of the CQLF meta model.
Image source: (Bański et al., 2013)

> **Level 3 (concurrent)** The potential of a QL to perform searches in multiply and concurrently annotated data makes it compliant with Level 3. In particular, it is possible for a Level 3 language to formulate various possibly conflicting constraints, stemming from concurrent annotations, on the same data (e.g. on a single word).

These basic definitions of CQLF Levels are illustrated in the schema in figure 3.1. As both Level 2 and Level 3 rely on segmental annotations, any L2 or L3 language is also L1-compliant. The resulting hierarchy is partial, because L3 does not necessarily build on L2 (although L3 languages may, of course, be capable of accessing complex annotations as required by L2).

By way of example, a query language needs to be able to express the following requests to qualify for the respective CQLF Levels. Note that this particular L3 example does not require the annotation of hierarchical structures, but only demands word-level annotations. However, L3 queries may of course involve hierarchical structures, too, and therefore require an L2-compliant QL.

positional containment see (Sperberg-McQueen and Huitfeldt, 2008).

L1 *Find all occurrences of 'corpus' within five words of 'query'.*

L2 *Find all noun phrases that contain the word 'query'.*

L3 *Find all words that are annotated as NE in CoreNLP and NN in Mate.*

Following the above definitions, and recalling the respective expressiveness properties of the QLs presented in sections 2.1-2.3, we can locate those languages at the following CQLF Levels.

> **Poliqarp QL** shows compliance with **Level 2** thanks to its `within` operator (cf. example 10) that allows the retrieval of structures contained within certain spans (sentences and paragraphs).
>
> **COSMAS II QL** is compliant with **Level 2** for its ability to query for containment of words, sequences or phrases within other structures via the `#IN` and `#OV` operators (cf. examples 16–18)
>
> **ANNIS QL** qualifies for **Level 3** due to its ability to access concurrent annotation layers within the language proper (cf. example 30). It is also L2-compliant because of the possibility to query for dominance and dependency annotation (cf. examples 28 and 29).[12]

3.2 Feature ontology

The second aspect of the establishment of CQLF as a metric involves the specification of a feature ontology. In principle, this is a more fine-grained collection of QL expressiveness criteria that allows for a more exact comparison and evaluation of QLs, especially such that belong to the same coarse CQLF Level. This collection aims to comprise the set of features attested in a wide range of QLs.

It shall be noted that the work on this feature ontology is still in progress and nowhere near complete, such that more a detailed account cannot be given at this point. However, this aspect of the theoretical side of CQLF is certainly the one which

[12]The reader might question the *raison d'être* of other QLs if ANNIS is powerful enough to subsume them in terms of expressiveness. The answer to this has two dimensions: (i) not every corpus query engine aspires to be powerful enough to support all features of a QL, be it for concentration on specific research questions or for a trade-off between speed and linguistic detail and (ii) there are, besides expressiveness, several factors that determine the 'quality' of a QL (see the introductory section of this thesis), and users might have grown accustomed to specific QLs or want to replicate results from earlier work using the exact same queries.

is of most relevance to the practical work of defining a concrete CQLF serialisation format. In particular, the present thesis, with its detailed study of three query languages and its careful analysis of the equivalence of certain features in those QLs, may well inform future work in the specification of the CQLF feature ontology.

Part II

Specifications of the KoralQuery Protocol

4 KoralQuery: A CQLF metalanguage

The original proposal of a corpus query lingua franca in Mueller (2010) conceived of it as a generic query language that can be processed by a wide range of corpus query systems and thus enables them to interoperate. The creation of such a metalanguage has (besides the described development of a metric for query languages) essentially formed the second major work item of the CQLF endeavour and is presented in the remainder of this thesis under the name *KoralQuery*. Before the concrete approach to the development of the metalanguage as well as its detailed specifications and properties are presented in the remainder of the thesis, this section discusses the requirements and caveats that are raised by such an enterprise.

To avoid getting on the wrong track with the term 'metalanguage' and, consequently, the entire project, it is worth noting what is actually meant by the term in the present context. Rather than a true interlanguage in the sense of Jespersen (1930), which would be understood and spoken by 'speakers' of different query languages, KoralQuery is more to be thought of as a pivot language that defines the target space for one-way mappings from concrete QLs. Importantly, this pivot language is not primarily intended to be understood or spoken by humans, but by corpus query systems. As a consequence, KoralQuery is not designed to promote the mutual intelligibility between QLs from the human point-of-view (e.g. allowing a user familiar with language A to understand a query in language B by means of the metalanguage), but instead to allow query systems to understand different QLs, provided that those can be translated into the metalanguage and that the query systems can process the metalanguage.

This illustrates the two central benefits that come with a corpus query lingua franca:

(i) query engines are able to use different query languages interchangeably,

(ii) query engines can interoperate.

The first of these two points, which is concretely put into practice in KorAP, is primarily of interest for the end user, who may choose to formulate a query in any of the available QLs. The second aspect, by contrast, allows for a cross-platform retrieval of query results. This can be very helpful in a distributed architecture of query systems, where different systems have access to different data but can (possibly depending on user permissions) collaboratively process user queries and combine their results in a federated search environment (an example of which is the CLARIN Federated Content Search).

In order to provide the sketched benefits to a full extent, the development of the KoralQuery specifications is subject to certain principles and requirements addressing the expressiveness of the metalanguage as well as aspects of maximal flexibility and minimal redundancy. The remainder of this section discusses these points and makes some preliminary suggestions of possible solutions, which will serve as guidelines for the subsequent elaboration of the KoralQuery type and operation specifications in section 5.

4.1 Expressive power

A (formal) language is a set of strings that are composed of primitive symbols according to certain production rules (Harrison, 1978). Naturally, a minimal requirement for a true *lingua franca* is to cover all features of the languages which it is supposed to subsume, i.e. it needs to be a superset of all its covered languages in the sense of the previous definition. However, as KoralQuery is not intended to be a query language proper that would be directly used by researchers to send queries to a system, but instead is a set of abstract representations of query concepts, we approach this requirement in an indirect fashion. In particular, the superset quality of KoralQuery relates to the abstract representation of queries rather than actual query strings, such that our goal is to define KoralQuery and its serialisation in such a way that every concrete query (in any of the supported QLs) can be mapped to what is a valid KoralQuery query representation.

Note that, of course, KoralQuery does not claim to be a *global* lingua franca in the sense that it could be used to represent queries formulated in any existing language, but is for now limited to the three QLs that were presented in the first part of this thesis. However, those languages represent quite different QL families and comprise a fairly broad array of features, such that the most important demands of query languages in general are covered by the unification of these languages (cf.

table 2.1).

4.2 Linguistic theory neutrality

Most existing corpus query languages have been designed with specific data models in mind. The choice of a data model, in turn, is often motivated through specific types of queries that a querying system is supposed to answer, and therefore influenced by certain linguistic theories. For instance, ANNIS QL is tailored towards exploring graph-based annotations such as constituents and pointing relations, whereas COSMAS II QL and Poliqarp QL are mainly intended to query for sequential structures. Along similar lines, in the case of Poliqarp QL, morphosyntactic annotations are restricted to a limited number of categories with fixed signifiers, e.g. `pos` for parts-of-speech or `degree` to indicate adjectival comparison (positive, comparative or superlative form).

Clearly, such specificity is unwanted in a metalanguage, which needs to be flexible enough to cover as many linguistic phenomena and theories as possible. KoralQuery is therefore required to be neutral with regard to

(i) the type and structure of linguistic annotation on the data (e.g. hierarchical constituent structure, graph-based dependencies, flat morphosyntactic annotation etc.), and

(ii) the choice of specific tag sets, e.g. for part-of-speech annotations or dependency labels.

The KoralQuery protocol that this thesis proposes intends to achieve such neutrality as follows. A maximal independence from the annotation type and structure is ensured through a modular, nestable system of types and operations. The combination of those different types and operations allows for the representation of arbitrarily complex linguistic patterns. In particular, the highly flexible `relation` operation (cf. Section 5.8) allows access to any arbitrary annotation layer that encodes linguistic relations between two or more linguistic entities (e.g. tokens or phrases). This is possible through setting particular constraints on the foundry and layer, i.e. the `relation` operation encodes this information by means of certain parameters.

Neutrality with respect to grammatical category names and tag sets is guaranteed by the unrestricted specification of key-value pairs within a `term` object (cf. Section 5.5). This object also allows the specification of a particular `foundry` and `layer` in order to access certain annotation sources and layers. It is even possible to access different foundries and layers on the same data, using different values for the aforementioned attributes within a single query.

Of course, language-side neutrality is not enough for a query system to allow access to any sort of linguistic annotation. Firstly, this annotation needs to be made accessible through an index or some other backend technology (e.g. a graph or a relational database). Secondly, the system still needs to interpret the query representation and its various combinations of operations and types. Both these requirements are by no means trivial to meet, and it might in many cases be sensible for a query system to only actively support a subset of the many different KoralQuery features, depending on resources and the intended application.

4.3 Redundancy avoidance

Another principle for KoralQuery as a metalanguage addresses the avoidance of redundant structures, i.e. the presence of multiple language elements expressing equivalent linguistic concepts. While this is certainly a very straightforward and (obviously) well-motivated claim, putting it into practice is not always trivial when trying to map concepts from very different QLs to a single super-language. The principal reason for this is probably the difficulty to recognise equivalent concepts across QLs, possibly due to different intentions or implicit assumptions behind certain features in different languages, or plainly because operators differ highly in their syntax. The latter aspect also becomes challenging in programming, when implementing an automatic translation from different QL syntax types to a common format.

The goal of avoiding redundancy is equivalent to keeping the language minimal, e.g. not defining a dedicated operation for optionality (i.e. the optional realisation of a query element) when an operator for repetition is already present and can capture optionality (through claiming that a query element occur 0 or 1 times at a certain position). In general, where applicable, one should try to express certain QL concepts through (combinations of) existing concepts in order to spare as many individual types and operations as possible.

However, while looking to discover equivalences between operations across QLs, it is crucial to be sensitive to potentially subtle differences between seemingly equivalent structures. An example of this is the `#OV` operator in COSMAS II and the `_o_` (overlap) relation in ANNIS QL, see footnote 9.

4.4 Object nesting and return values

Another central characteristic of the KoralQuery protocol is that the types and operations listed below are *nestable*, i.e. one object can be dominated by another. This means that the former is an argument of the latter or, in other words, that an operation defined in the parent object takes scope over the the nested object. For example, a query that asks for the phrase *query language* contained in a verb phrase will be represented as an object specifying a containment operation with two operands (arguments), namely the verb phrase object and the object for the phrase *query language* (which in turn has two operands, namely the tokens it consists of).[13]

A KoralQuery instance is thus a tree structure with a single root element that corresponds to the central linguistic relation defined in the query and leaf elements that usually correspond to low-level linguistic structures such as token attributes (e.g. the requirement that a matched token has a certain part-of-speech tag or surface form). The depth of the query tree depends directly on the complexity of the query, more concretely on the level of embedding of linguistic and meta-linguistic constraints as defined below. The different types take different numbers of operands, ranging from no embedded objects for the typical leaf nodes to an infinite number of operands for sequences (see the `sequence` operation in Section 5.3.1). The ordering of an objects' operands, which are represented as a list, is generally of relevance as it regulates which argument has which function in operations that are not commutative.

Nesting several KoralQuery objects means that *return values* are passed on from lower-level objects (i.e. more deeply-nested sub-queries) to higher-level objects. Generally, the return value of a KoralQuery object[14] is a text span, which in turn is defined through a start and an end offset with respect to the underlying data. Thus, in the above example (a sequence that is embedded into a positional relation query with another phrase) the individual sub-queries (the sequence and the verb phrase) return text spans which are then evaluated for satisfying the positional constraint.

[13]Note, however, that this nesting is by no means arbitrary or unregulated. In fact, there are different *categories* or *types* of objects that are compatible with each other. The definitions of those types and the constraints on their combination are provided in the following section.

[14]Strictly speaking, this is only true for *basic types* and the `group` type (again, see the following section).

5 KoralQuery types and operations

The following KoralQuery specifications list a range of linguistic and operational types which can be grouped into the following two classes.

• **Span types** denote spans of text. They can be further sub-classified into (i) *basic types* that represent shallow linguistic entities such as words, phrases and sentences[15] and (ii) *complex types* that define linguistic or result-modifying operations on a set of elements of basic or complex types.[16] Central to complex types is that they maintain a list of *operands*, which are the elements on which the introduced operation works.

• **Parametric types** contain specific information that is required by certain span types. They are intended to normalise the usage and representation of similar or equal parameters used across those types. Many complex types, for instance, require some way of expressing a numeric range of a minimum and maximum value to quantify the instantiation of a certain relation. Rather than expressing such ranges in the complex types directly, those types contain a `boundary` type with a consistent syntax and semantics across its usages.

All of those types are themselves complex structures in that they are composed of a specific set of obligatory and optional attributes that carry corresponding values. Those values, in turn, are also constrained to be of specific data types. They can either be primitives (like string, integer or boolean), parametric KoralQuery types, or controlled values. In the latter case, the attribute type is indicated by `@id` in the specifications below.[17]

To fully specify the different KoralQuery types, they are defined below with respect to three criteria.

i. Their type class (i.e. whether they are basic, complex or parametric types)

ii. Their denotation and function

[15]The reader may doubt the 'shallowness' of phrases and sentences, as these types are complex in the sense that they consist of sub-phrases that in turn may be composed of other elements. In the context of a data model that defines these types as spans over character offsets, however, phrases do not differ significantly from tokens given their definition through start and end offsets that are independent of any potential subordinate spans or tokens.

[16]Such linguistic operations may be precedence or dominance relations between two spans, while result-modifying operations allow for, e.g., disjunction of matches or the re-definition of match spans according to specific constraints.

[17]The choice for `@id` to denote controlled values is grounded in the usage of JSON-LD as a serialisation format for KoralQuery (see Sec. 8), where the `@id` keyword serves "to uniquely identify things that are being described in the document with IRIs or blank node identifiers" (Sporny et al., 2014).

iii. Their obligatory and optional attributes as well as the respective types and semantics of those attributes

Optional attributes are prefixed with an asterisk (*). Attribute type indications in brackets denote an array of the types enclosed in the brackets.

5.1 The `token` type

A `token` is a *basic type* and denotes a string of characters making up a single surface word form with respect to a tokenisation layer in the annotations. It is defined through the start and end offsets of that word form, whose requested properties are specified in a `term` or `termGroup` held by the `wrap` attribute.

Attribute	Type	Values/Description
`wrap`	`term` or `termGroup`	Holds information on search key, foundry, layer, value.

5.2 The `span` type

The *basic type* `span` denotes a linguistic entity that stretches across one or more tokens and typically corresponds to a grammatical phrase, although paragraphs or other discourse units may be represented as spans, too, depending on the underlying data. Spans are interpreted with respect to a `foundry`, a `layer` and a `key`, the latter of which indicates the phrase type (or, more generally, *category*) of the requested entity (e.g. `s` for a sentence or `np` for a noun phrase, again depending on the annotations). Graph-theoretic properties of the span element (such as the number of children or whether this is the leftmost child of its parent) can be specified using `attr` (see query (34) for an exemplary use case).

Attribute	Type	Values/Description
*`foundry`	string	The foundry in which the span is annotated.[18]
*`layer`	string	The layer in which the span is annotated.
`key`	string	The span category.
*`match`	`@id`	Matching behaviour for `key` (see Sec. 5.9). Available values: `eq`, `ne`.
*`attr`	`term` or `termGroup`	Holds information on span attributes.

[18]If the optional `foundry` is not set, the query system is expected to interpret the `layer` and `key` specifications with respect to a default foundry.

5.3 The `group` type

A KoralQuery `group` is a *complex type* in that it establishes a certain relation between its operands via the obligatory `operation` attribute, which specifies the exact nature of the relation and is explicated in further detail in the following section. The arguments of the operation, i.e. the objects upon which the operation is defined, are expressed in a list called `operands`. While these arguments are usually span types regardless of the concrete operation, the number of operands as well as additional parameters of the group depend on the operation. For instance, a *position* operation demands the specification of a `frame` attribute as well as exactly two arguments, while a *disjunction* operation does not call for any parameter, but requires a minimum of two operands. Those constraints are explicated for every individual operation below.

Where not otherwise stated, the match that the operations yield is defined as the text area reaching from the start offset of the match for the leftmost span as defined in the operands to the offset of the rightmost span match. However, individual query language processors (see Section 9) will define classes around the arguments of certain operations to make those available for referencing when the match is expected to only consist of one or more operands rather than the entire text area that they span.[19]

Attribute	Type	Values/Description
`operation`	`@id`	Operation definitions: see Section 5.3.1.
`operands`	[*span types*]	Depends on `operation`. See operation definitions for details.

5.3.1 Group operations

This section lists all linguistic relations and result modifiers as made available by the `operation` attribute of the `group` type. The distinction between linguistic relations and result modifiers may not always be clear-cut (e.g. in case of the `repetition` operation) and is to be thought of as an intellectual aid rather than a functional dichotomy. Attached to each operation is a listing of the obligatory and optional parameters (if any) that closer defines the relation introduced by the respective operation.

[19]The concepts *class* and *reference* will be introduced and defined later in this chapter.

Linguistic operations

The `position` operation

This operation establishes positional constraints between its operands, which means that the spans denoted by the operands are in some linear relation (e.g. overlapping, left-aligned, ...) with respect to surface text offsets. The specific valid positional relations are given by the `frames` attribute, whose values are to be treated as *alternatives*. More concretely, this operation matches all spans that satisfy any of the constraints expressed in `frames`. The values for this attribute are usually third-person verbs, implying that in the case of asymmetric frames, the first operands 'governs' the second (cf. the `startsWith` frame). Position operations can be negated using the Boolean `exclude` attribute, which matches all but the respectively specified frames. See Section 5.9 for definitions of all frames.

Attribute	Type	Values/Description
`frames`	`[@id]`	Lists the allowed positional relations (see table 5.1).
*`exclude`	`boolean`	If `true`, negate the positional relations[20]
`operands`	[*span types*]	No. of operands: 2. First operand corresponds to A in the definitions in table 5.1, second operand to B.

The `sequence` operation

As another linear position constraint, a `sequence` operation requires that its operands are in a sequential order. They need not immediately precede each other, but may be separated by other elements as specified by the `distances` attribute (if set). The Boolean `inOrder` attribute specifies whether the ordering of the operands is of relevance for the sequential relation.

Attribute	Type	Values/Description
*`distances`	`[distance]`	Distance constraints between operands .
*`inOrder`	`boolean`	`false`: order of operands is irrelevant.
`operands`	[*span types*]	No. of operands: 2+. Objects occurring in sequence. Distance constraints hold between each two subsequent operands in the list.

[20] In this case, only the first operand will be returned.

The `relation` operation

This is a versatile operation that introduces a directed graphical (i.e. 'pointing') linguistic relation between its arguments. The exact nature of the relation (expressed by its foundry, layer and key bindings with respect to the underlying data) is indicated by the `relation` attribute, such that a great degree of flexibility is retained for this type. Depending on the annotation defined in that attribute (and if such an annotation layer exists in the data), this group may express constraints that pertain to syntactic dependencies, coreference relations, or any other two-place linguistic relation.

Attribute	Type	Values/Description
`relation`	`relation`	Specifies relation between operands.
`operands`	[*span types*]	No. of operands: 2. The directed relation holds between the first and the second operand.

Result-modifying operations

The `disjunction` operation

This is a non-exclusive disjunction operation on its operands. Any match of any of the arguments will be returned. Hits can be grouped using a `merge` operation (see below).

Attribute	Type	Values/Description
`operands`	[*span types*]	No. of operands: 2+. The operation returns any span that matches one of the operands in the list.

The `repetition` operation

This operation requires that its argument is repeated a certain number of times, forming a coherent sequence in the text. The number of times that the argument is repeated is specified using the `boundary` attribute.

Attribute	Type	Values/Description
`boundary`	`boundary`	The argument's minimum and maximum repetition.
`operands`	[*span types*]	No. of operands: 1. The repeated object.

The `class` operation

This operation introduces a numbered identifier (called a *class*) for its operand. Classes can be used for grouping and referencing individual elements in a KoralQuery tree. This is, for instance, necessary in cases where several constraints are expressed on the same linguistic entity. Other constructs that make use of classes are displaying directives.

Attribute	Type	Values/Description
*`classIn`	`[int]`	Input class numbers
`classOut`	`int`	Output class number
*`classRefCheck`	`[@id]`	Set-theoretic condition on input classes. Possible values: see 5.9. Requires exactly two input classes. Results that do not fulfil this condition are excluded from the result set.
*`classRefOp`	`@id`	Operations on input classes, creates new output class. Possible values: see 5.9.
`operands`	[*span types*]	No. of operands: 1. The object on which the class is declared or the operation works.

The `merge` operation

Wrapped in this operation, the result set of the operand is condensed by merging equivalent matches (which start and end at the same offset), also merging their classes.[21]

Attribute	Type	Values/Description
`operands`	[*span types*]	No. of operands: 1. The object whose matches are condensed.

5.4 The `reference` type

The *complex type* `reference` offers two basic modes by which specific sub-elements of the query may be referenced using the parameters `classRef` or `spanRef`.

The first mode defines a sub-query in its `operands` list and reduces each span returned by that sub-query to sub-spans. The specific elements to be returned (e.g.

[21]An example of this would be a match for a query that features a disjunction, and where some span in the result set satisfies both constraints in the disjunction. In the default case, the match appears twice in the result set, but is condensed to one hit using a `merge` operation.

a certain token) can be defined using a `class` operation and corresponding class ID entries in the `classRef` attribute. Alternatively, this mode allows to 'shrink' a result to a specific set of counting tokens, e.g. the first three tokens of the result or tokens five to eight, similar to the `substring` methods in many programming languages. These sub-spans are indicated using a `spanRef`.

Applied in the second mode, a `reference` does not contain an `operands` list, but a mere `classRef` attribute which refers back to classes previously defined elsewhere in the query tree.

In both its modes, the central benefit of this type is to provide an embedding `operation` with access to specific elements defined in the `operands` of an `operation` embedded by the reference which would otherwise be unavailable. This is essential for queries that comprise multiple predications on the same entity. More concrete illustrations of the necessity of this type are provided in section 9.4.

Attribute[22]	Type	Values/Description
`operation`	`@id`	`focus`: Reduce the operand match to classes as defined in `classRef` or tokens as defined in `spanRef` (mode 1). Get a copy of the match for the class in `classRef` (mode 2). `split`: Spans in the operand bearing the same class form individual matches (mode 1 only).
*`classRef`	[`@id`]	Classes to which the operation is applied.
*`spanRef`	[`@id`]	Requires one or two entries: The first number is the start token position, the second, if specified, the number of tokens to return.
*`operands`	[*span types*]	No. of operands: 1. The object whose match is to be reduced.

5.5 The `term` type

This *parametric type* defines properties encoded in a certain `foundry` and `layer` on tokens or other elements. A string value, e.g. the surface or lemma form (depending on the respective layer), is held by the `key` attribute, while `value` is used in combination with `key` to express two-place properties such as morphological key-value pairs, e.g. `tense:pres`. A key is interpreted with respect to case by default, which

[22]The `reference` type requires that (in mode 1) either `classRef` or `spanRef` is set.

can be disabled setting the Boolean caseInsensitive attribute to true. Beyond that, the interpretation of the key may by manipulated using the type attribute, which allows the key to be interpreted as a regular expression, among others. Ultimately, the match attributes allows for manipulations of the result set, e.g. by demanding that only tokens *not* satisfying the key/value condition be returned.

Attribute	Type	Values/Description
*foundry	string	The annotation foundry.
layer	string	The annotation layer.
key	string	The search key.
*value	string	Denotes the value in key-value annotations, e.g. tense:pres in a morphology layer.
*match	@id	Matching behaviour for value (if given, else key). Available values: eq, ne
*type	@id	Type interpretation for value (if given, else key). Available values: string, regex, wildcard, punct
*caseInsensitive	boolean	true: key matching not case-sensitive
*root	boolean	true: This element forms the root of a tree, e.g. a dependency annotation (dep. on layer). false: Must not be root.
*arity	boundary	The element's number of children in a certain annotation (indicated by layer).
*tokenarity	boundary	The number of tokens a span governs.

5.6 The termGroup type

Several metadata constraints expressed in individual term objects can be grouped in the *parametric type* termGroup that specifies a logical operation (AND/OR) on its operands. Depending on this relation, either all or at least one of the constraints in the operands must hold. A termGroup may also embed another termGroup, such that conjunctions and disjunctions can be nested.

Attribute	Type	Values/Description
`relation`	`@id`	`and`: Conjunction of constraints in operands `or`: Disjunction of constraints in operands
`operands`	[`term` or `termGroup`]	No. of operands: 2+. The constraints on which the logical operation works

5.7 The `distance` type

This *parametric type* allows to specify a distance between the operands of a `sequence` with respect to some measure `key` (e.g. words, sentences). The distance range, i.e. the minimum and maximum values of `key` elements that one would need to 'travel' from one operand to the other, is indicated using the `boundary` attribute. Thus, the distance from one token to another token directly succeeding it is specified through the measure `w` (for word-distance) and the range `min:1`, `max:1` contained in a boundary object. The type also defines the Boolean `exclude` attribute, which excludes the occurrence of the operands within the present distance if set to `true`.

Attribute	Type	Values/Description
`key`	`string`	Measure of distance, may be `w` for words, `s` for sentences, `p` for paragraph, or `t` for text.
*`foundry`	`string`	Foundry in which the distance measure is annotated.
*`layer`	`string`	Layer in which the distance measure is annotated.
`boundary`	`boundary`	Indicates degree of distance.
*`exclude`	`boolean`	`true`: Containing `sequence` returns first operand if not within specified distance of second operand.

5.8 The `relation` type

Operations specifying directed two-place relations between their first and second operand encode details on those using the *parametric type* `relation`, which in turn contains a `term` or `termGroup` held by the `wrap` attribute, analogously to the practice for tokens. In order to capture a transitive relation[23] between the operands, the degree of the relation may be indicated using the `boundary` attribute.

[23]That means an indirect relation, like the dependant of a dependant of some token.

Attribute	Type	Values/Description
wrap	term or termGroup	Holds information on layer key, foundry, layer, value
*boundary	boundary	Indicates degree of relation.

5.9 Attribute values

The following specifications list and define the possible values that attributes with controlled values may take.

The match attribute

This attribute determines the matching behaviour for a certain attribute of a certain KoralQuery type.[24] For instance, this attribute may postulate that a given attribute shall *not* be matched.

Attribute	Description
eq	The key/value matches the data.
ne	The key/value does not match the data.
leq	The value is less than or equal to ($\leq$) the key data.[25]
geq	The value is greater than or equal to ($\geq$) the key data.
contains	The value is contained in the data of key.[26]
excludes	The value is not contained in the data of key.

The type attribute

This attribute declares the type of another attribute contained in the same object, thus providing information on how to interpret that attribute. This is necessary to distinguish regular expressions or dates represented as RFC-3339 strings from plain strings.

[24]To which attribute this applies is determined by the respective type that holds the match attribute, and might be a key (for span objects) or a value (for terms in which value is specified, e.g. as a morphological value).

[25]Obviously, this only applies to numerical values, such as dates in virtual collection queries (see Sec. 6.1).

[26]This value only applies to virtual collection queries. An example of this is a query that requires a certain word to be contained in the header of a document.

Attribute	Description
`string`	Interpret `key/value` as plain string.
`regex`	Interpret `key/value` as regular expression.
`punct`	Interpret `key/value` as punctuation symbol.
`date`	Interpret `key/value` as date.[27]

The `classRefCheck` attribute

This attribute postulates some set-theoretic relation to hold between two classes that are defined in the `classIn` array of the containing `class` operation group (see Section 5.3.1). That group then only returns those results from the `operands` that fulfil the condition expressed by the `classRefCheck`. Those that do not fulfil that condition are excluded from the result set. In the table below, A and B denote the sets of tokens that are subsumed under the first and the second class in `classIn`, respectively.

Attribute	Condition	Description
`disjoints`	$A \cap B = \emptyset$	The classes do not share a token.
`intersects`	$A \cap B \neq \emptyset$	The classes share at least one token.
`includes`	$A \supset B$	Class A is a proper superset of class B.
`equals`	$A = B$	The classes subsume the same tokens.
`differs`	$A \neq B$	The classes do not subsume the same tokens.

The `classRefOp` attribute

This attribute specifies a set-theoretic operation on the input classes in `classIn` which results in a new `classOut` to be returned by the containing `class` operation group. The operations in the table below are the available class definition functions. The set notation defines which of the tokens t in the span are to be included in the new class. C_i stands for the i-th class defined in `classIn`.

Attribute	Definition	Description
`union`	$\{t \mid t \in \bigcup_i C_i\}$	Tokens that are in any of the classes.
`intersection`	$\{t \mid t \in \bigcap_i C_i\}$	Tokens that are in all of the classes.
`inversion`	$\{t \mid t \notin \bigcup_i C_i\}$	Tokens that are in none of the classes.
`deletion`	$\emptyset$	No tokens (creates 'empty' pseudo class).

[27]This type only applies to metadata queries.

The `frames` attribute

This attribute specifies the exact positional relation between two spans connected by a `position` operation. The list of position frames in table 5.1 is adapted from the typology of overlap relations of markup elements in (Durusau and O'Donnell, 2002). In the table, two spans A and B are represented as `<a>...</a>` and `<b>...</b>`, respectively, for illustration.

6 Meta-information on the query

While the elements specified in the previous section all describe the linguistic query itself, the KoralQuery protocol reserves a set of objects that may contain various meta-information on the query. Those currently pertain to document filtering by metadata constraints and displaying directives, and are specified below.

6.1 Document-level filtering

KoralQuery provides a possibility to specify metadata constraints that act as filters on document collections using the `collection` attribute. This functionality is motivated through metadata-filtering devices such as the `meta` keyword in Poliqarp and the concept of *virtual collections* in KorAP. In the latter case, those metadata constraints serve a dual purpose. Besides the obvious benefit of allowing users to restrict their search to documents that meet specific requirements such as publication date, authorship or genre, they can be used on the system side to control access to texts that the user has no permission to read, e.g. for copyright reasons.[28] Upon receiving a query, the KorAP backend may then add a constraint in `collection` to restrict the set of searched documents to specific subcorpora, cf. (Bański et al., 2014).

6.1.1 The `doc` type

This type represents a single metadatum constraint using its `key` and `value` attributes. Metadata constraints pertain to documents, such that a `doc` object describes a document that fulfils (or does not fulfil, depending on the `match` attribute) the specified condition.

[28]Many of the texts in DeReKo, for example, are accessible for IDS members only, and require the user to belong to a respective user group.

Frame	Definition	Illustration
`succeeds`	`a.start > b.end`	`              <a>....</a>` `<b>....</b>`
`succeedsDirectly`	`a.start == b.end`	`          <a>........</a>` `<b>.......</b>`
`startswith`	`a.start == b.start &&` `a.end > b.end`	`<a>..................</a>` `<b>......</b>`
`endswith`	`a.start < b.start &&` `a.end == b.end`	`<a>..................</a>` `            <b>......</b>`
`overlapsRight`	`a.start > b.start &&` `a.end > b.end &&` `a.start < b.end`	`      <a>............</a>` `<b>.........</b>`
`alignsRight`	`a.start > b.start &&` `a.end == b.end`	`         <a>.........</a>` `<b>..................</b>`
`isWithin`	`a.start > b.start &&` `a.end < b.end`	`      <a>....</a>` `<b>..................</b>`
`matches`	`a.start == b.start &&` `a.end == b.end`	`<a>..................</a>` `<b>..................</b>`
`alignsLeft`	`a.start == b.start &&` `a.end < b.end`	`<a>.......</a>` `<b>..................</b>`
`isAround`	`a.start < b.start &&` `a.end > b.end`	`<a>..................</a>` `      <b>....</b>`
`overlapsLeft`	`a.start < b.start &&` `a.end < b.end &&` `a.end > b.start`	`<a>............</a>` `         <b>.........</b>`
`precedes`	`a.end < b.start`	`<a>....</a>` `              <b>....</b>`
`precedesDirectly`	`a.end == b.start`	`<a>.......</a>` `          <b>........</b>`

Table 5.1: Definitions and illustrations of available `frames` values.

Attribute	Type	Values/Description
`key`	`string`	The metadatum attribute.
`value`	`string`	The metadatum value.
*`type`	`@id`	The type of the `value` (see Sec. 5.9).
*`match`	`@id`	Matching behaviour for `value` (see Sec. 5.9).

6.1.2 The `docGroup` type

This type introduces a Boolean conjunction or disjunction on its `operands`, which in turn are `doc` or other `docGroup` objects. It it thus possible to define several metadata constraints and to connect them logically, describing documents that fulfil the combined conditions.

Attribute	Type	Values/Description
`operation`	`@id`	`and`: Conjunction of constraints in `operands`.
		`or`: Disjunction of constraints in `operands`.
`operands`	[`doc` or	No. of operands: 2+. The constraints on which
	`docGroup`]	the logical operation works.

6.2 Displaying directives

KoralQuery offers the definition of displaying directives that influence the presentation of search results in a KWIC view. Two such directives are implemented, both using lists of integers to refer to classes defined in the query. The `highlight` attribute specifies which of the classes are to be *highlighted* in the results display. Implementations of corpus search engines that employ KoralQuery may use this to typeset the spans covered by different classes in different styles, e.g. to underline or color them, or to set them in bold face or italics. Similarly, the `align` attribute refers to classes that serve as alignment anchors in KWIC, creating one or more columns across the results on which the respective classes are aligned.

Attribute	Type	Values/Description
`highlight`	`[int]`	IDs for classes to be highlighted in KWIC.
`align`	`[int]`	IDs for classes that serve as alignment anchors in KWIC.

7 PoliqarpPlus QL: a KoralQuery model language

The process of specifying KoralQuery was complemented with the definition of a concrete model language that was used to simultaneously illustrate KoralQuery concepts and to assist the implementation of KoralQuery types and operations in the KorAP backend using concrete queries. This model language was constructed by gradually extending Poliqarp QL[29] with numerous constructs as defined below. An exact definition of the query language is provided by the ANTLR grammar that is used to parse PoliqarpPlus input in the translation component Koral (see Section 9.2).[30]

Tokens

PoliqarpPlus extends the functionality of Poliqarp to express token properties at a more fine-grained level, allowing for the specification of foundries and values besides the native layer and key. An example of a token definition using foundry and value is provided in (36), while a negated constraint is illustrated in (37). Those constraints may also be connected by logical operations (38), which also makes it possible to query different annotation sources (foundries) on the same data (39).

(36) `[mate/m=tense:pres]`
A token in present tense, annotated in the morphology layer of the `mate` foundry.

(37) `[p!=VVFIN]`
A token whose part-of-speech is not *VVFIN* (finite main verb in the STTS tag set).

(38) `[orth=sie & m=number:sg]`
A token with the surface form *sie*, that is in singular number according to the morphology layer (of the default foundry).

(39) `[mate/p=N & cnx/p=NE]`
A token that is annotated as a noun (N) in the `mate` foundry but as a proper noun (NE) in the `cnx` foundry.

[29]The choice of Poliqarp QL over ANNIS QL was mainly motivated by the straightforward introduction of nestable operators, which made for an easy translation of the query language to KoralQuery, cf. Sections 9.2 and 9.4.

[30]The grammar is available at the Koral GitHub repository.

Spans

With the exception of restricting matches to occur inside single sentences or paragraphs using `within s|p`, native Poliqarp QL does not provide a way to query for span annotations representing sentences or other phrases. PoliqarpPlus offers this functionality by specifying spans in angle brackets. As for tokens, spans can be specified with respect to a foundry, layer and key ().

(40) `<NP>`
A noun phrase.

(41) `<cnx/c=NP>`
A noun phrase according to the constituency layer of the `cnx` foundry.

Regular expressions

PoliqarpPlus allows the user to specify certain attributes with regular expressions, indicating them by double quotes. Regular expressions are generally permitted for `key` attributes, as is exemplified for tokens and spans, respectively, in (42) and (43).

(42) `[orth="B(u|ü)ch(er)?"]`
A token whose surface form matches the given regular expression.

(43) `<cnx/c="S.*">`
A span whose category as defined in the `cnx` constituency layer starts with *S*.

Relations

The `relatesTo()` operator provides access to hierarchical relations as defined by CQLF Level 3 and represented in KoralQuery by the `relation` operation. As this type can be parametrised with a `term` or `termGroup` to specify constraints on the relation (such as the foundry and layer in which it is annotated), the `relatesTo()` operator allows the specification of parameters. The `dominates()` operator is a short form for `relatesTo(c:)`, i.e. it automatically binds the relation constraint to the constituency layer, given that the constituency layer in the default foundry is identified by `c`. Consequently, examples (45) and (46) are equivalent.

(44) `relatesTo(<s>,<np>)`
A sentence span that is in some (unspecified) relation to a noun phrase.

(45) `relatesTo(c:<s>,<np>)`
A sentence for which a relation to a noun phrase is annotated in the constituency layer.

(46) `dominates(<s>,<np>)`
Short form for the previous query.

(47) `relatesTo(mate/d=SBJ:[orth=trifft],[pos=N])`
A token *trifft* which has a noun as a `SBJ` (subject) dependant, annotated in the dependency layer of the `mate` foundry.

Position operators

Positional relations between two spans can be indicated in PoliqarpPlus using the operators listed in the table below.

Operator	Meaning	KoralQuery Frames
`matches`	The two spans have the same start and end offsets.	`matches`
`startsWith`	Equal start offsets, the second span does not exceed the first.	`startsWith,matches`
`endsWith`	Equal end offsets, the second span does not start before the first.	`endWith,matches`
`contains`	The second span does not start before or exceed the first.	`contains,startsWith, endsWith,matches`
`overlaps`	The two spans overlap, neither contains the other.	`overlapsLeft, overlapsRight`

The syntax for position operations requires two comma-separated span-denoting arguments that are enclosed in parentheses. The usage is illustrated in the following examples:

(48) `endsWith(<s>,[pos=PREP])`
A sentence that ends with a preposition.

(49) `matches(<np>, [pos=ART][pos=N])`
A noun phrase that consists of an article and a noun.

Classes

KoralQuery `class` operations can be introduced in PoliqarpPlus by placing braces around spans, where the span can additionally be prefixed with an integer and a colon to define a specific class number. If no class number is indicated (as in example 50), the class will receive the default ID `1`.

(50) `[orth=see]{[pos=N]}`
A sequence of a token *see* and a noun, with a class defined on the latter.

(51) `[orth=see]{1:[pos=N]}`
As above, explicitly assigning class ID `1`.

References

PoliqarpPlus lets the user reduce the match span of a sub-query using the `focus()` and `submatch()` operators, the first of which is defined to refer to a specific class and therefore takes an optional class number as a parameter. Similar to class definitions, if no class number is provided in a `focus` query, class `1` will be referenced. Examples 52 and 53 are thus equivalent. It is also possible to combine several classes in a `focus()` query and specify logical operations (AND/OR) on them, such that only spans that are covered by all or any of the defined classes are returned (example 54).

The `submatch()` operator is used to reduce a span to a specific sub-sequence of tokens that it contains. To this end, two integer-valued parameters indicate which token is the first to be included in the sub-sequence (with the first token in the span assigned index 0, the second index 1, etc.) and how many tokens to include. A negative value for the first parameter denotes the *n*-th last token in a sequence. If no second parameter is provided, the remainder of the span (starting from the token denoted by the first parameter) is returned.

(52) `focus([orth=see]{[pos=N]})`
A sequence of a token *see* and a noun, with the match span reduced to the latter.

(53) `focus(1:[orth=see]{1:[pos=N]})`
As above, explicitly assigning and referencing class ID `1`.

(54) `focus(1&2:overlaps({1:[pos="V.*"][]{0,5}[pos="V.*"]},{2:<VP>}))`

Two verbs separated by at most 5 tokens, which overlap with a verb phrase. The match span is reduced to all tokens that occur in both operands of the `overlaps` relation.

(55) `submatch(0,2:<np>)`
The first two tokens of a noun phrase.

Punctuation

Using the `punct` pseudo-layer, the user can specify a punctuation symbol to search for. Additionally, by enclosing several symbols in double quotes, a pseudo character class is defined (consisting of a full stop, a question mark and an exclamation mark in example 57), such that any of those symbols is matched.

(56) `[punct=.]`
A full stop.

(57) `[punct=".?!"]`
A full stop, question mark or exclamation mark.

Part III

Serialisation of KoralQuery

This last part of the thesis addresses the concrete KoralQuery serialisation format as well as the Koral query serialisation component which is used to translate queries from a set of query languages to the meta format. To this end, occasional references will be made to certain pieces of source code or other resources used by the serialisation component. The reader is pointed to the public GitHub repository[31] where all of those resources reside under the BSD-2 license.

8 JSON-LD as a serialisation format for KoralQuery

A serialised data format allows the platform-independent communication and interoperability between different (web) services. [32] Following the recommendation of the ISO TC37 SC1 WG1, the KoralQuery protocol is serialised in a format based on JSON-LD (Sporny et al., 2014), an adaptation of JSON (ECMA-404, 2013) directed towards the transportation of linked data (Berners-Lee et al., 2001). The following sections 8.1 and 8.2 give an introduction to the JSON-LD format and the use of linked data in linguistic web services, respectively.

8.1 JSON-LD

JSON (JavaScript Object Notation) is a serialisation format for the text-based representation of complex objects. Being more lightweight and human-readable than XML, it is today a common alternative to XML serialisations and gaining popularity, especially because of its more natural way of organising data into key-value pairs versus the high level of abstraction that is found in XML. Importantly, JSON also differs from XML in that it is not a markup language. This makes it generally much more suitable for the representation of native data structures that can be found in the most common programming languages, especially dynamic ones. At the basic level, JSON offers the primitive types `string`, `number`, `true`, `false` and `null`. Next to these, the JSON data model relies on two principal complex structures:

[31]http://www.github.com/jbingel/Koral. The tagged release `v0.1` is relevant for this thesis.

[32]Serialisation here means the representation of a complex data structure in a sequential (possibly textual) format, which can be stored and later resurrected in the same or in a different environment.

objects, which are collections of key-value pairs. The key is always a string (indicated by double quotes), while the value may be of any type. Objects are denoted by braces, and key-value pairs are delimited by commas. Colons are used to delimit keys and values.

arrays, which ordered lists of values. Again, the value may be of any type. Arrays are denoted by brackets, and values are delimited by commas.

Embedding several levels of objects and arrays, JSON is capable of serialising data of arbitrary complexity as long as it can be represented using the basic types.

JSON-LD adds to this flexibility the unique identification of objects across different serialisations by means of URIs. To this end, the format provides a vocabulary of special keys with well-defined semantics, which are consistent across individual serialisations and thus allow for the uniform handling of JSON-LD objects in applications. For instance, JSON-LD objects carry a `@type` key whose value is a URI that serves as an unambiguous identifier of the object's type, such that any application may refer to the type specifications to know exactly what other keys and values to expect in this object as well as what they denote. Ideally, this very principle of unique identification also holds for all keys and, depending on their respective type, all values of an object. Consequently, following this principle in releasing JSON-LD serialisations, also keys and values can be universally and unambiguously interpreted and processed. The examples in listings 1 and 2 contrast plain (unlinked) key and value definitions with the expression of keys and values through unique URIs in the context of geographic data.

It is apparent from the examples that the serialisations can become quite verbose because of the replacement of short key and value strings with possibly long URIs. As a strategy to prevent this, JSON-LD reserves the `@context` key, whose value is an object that contains a set of mappings from shorthand names to URIs, thus essentially forming a namespace for the remaining file and all key and value names therein. Additionally, rather than providing the entire name-URI mappings with every serialisation, the `@context` may also reference an externally accessible JSON-LD object that defines the mappings, which has the obvious benefits of a short form and consistency (cf. listing 3).[33]

[33] The `@context` (or, in fact, any declaration of key-value types and conformances) may also act as an extenuated form of a document grammar, similar to an XML schema. While it cannot directly be used to validate the 'grammaticality' of a JSON-LD object in terms of key-value conformance, a failed resolution of a key or value by means of the `@context` at least indicates that some value is not

```
{
  "name": "Heidelberg",
  "inCountry": "Germany"
}
```

Listing 1: Plain JSON representation of a city. There is no mechanism that ensures that the keys and values, or the object in general, are identified uniformly by different applications

```
{
  "@type": "http://schema.org/City",
  "http://schema.org/name": "Heidelberg",
  "http://schema.org/containedIn":
    { "@id": "http://www.geonames.org/countries/DE/germany.html" }
}
```

Listing 2: Typed and linked JSON-LD representation of a city. The `@id` keyword declares that its value is a URI denoting a unique identification of the value (here through the GeoNames database).

```
{
  "@context": "http://example.org/contexts/geo.jsonld",
  "@type": "City",
  "name": "Heidelberg",
  "containedIn": "http://www.geonames.org/countries/DE/germany.html"
}
```

Listing 3: Condensed JSON-LD representation of a city. Using an external `@context` specification in which all the key terms and their value types are declared, the serialisation gets less verbose.

8.2 Linked data in linguistic web services

While the linked data paradigm has been established in a number of applications and scenarios that work with more discrete data and more concrete objects, it has received relatively little attention in linguistic web applications. In fact, there is as of now and to the knowledge of the author no such service that makes use of linked linguistic data. However, as mentioned in the section on related work, an effort

defined for a certain key or that some key is not defined for a certain object `@type`.

to define a linguistic Web Service Exchange Protocol (WSEP) is currently being pursued by Ide (2013). Through the specification of a common terminology of linguistic objects, such a protocol would facilitate the communication and interoperability between services such as (web-interfaced) NLP tools , language resources or corpus query systems. Also following the ISO recommendations, WSEP is based on JSON-LD. While no definitive account of the specifications can currently be given due to the state of WSEP as work-in-progress, the intention as formulated in Ide (2013) is to distinguish three basic linguistic types ("labels", "features" and "links") that can be used to capture linguistic objects, i.e. the input and output data types of various NLP tools, at different levels of complexity. Thus, by specifying their required input types and their provided output types, and as those types are well-defined within a common format and framework, web services are able to know exactly how they relate to and interoperate with other web services, e.g. other upstream or downstream NLP tools that they are pipelined with.

The KorAP search engine makes direct use of the JSON-LD based serialisation of queries formulated in any of the supported QLs. With the benefits of JSON addressing mainly the aspects of lightweight and human-readable representations as outlined above, the motivation for the use of a linked data paradigm such as JSON-LD lies in the potential to communicate and exchange data with other web services. In particular, sharing a common format for the representation of queries (and, on the backend level, query responses) allows for federated search strategies across query engines. For instance, a query that is submitted to KorAP can be forwarded to other query engines that have access to other data, and the responses from those systems (retrieval results, statistics, messages etc.) can be interpreted by KorAP and integrated into the federated search response.[34] An instance of a federated content search can be found in the CLARIN project.

9 Translating queries to KoralQuery

The representation of KoralQuery as a JSON-LD protocol is possible through the organisation of KoralQuery objects into key-value pairs, where the key can always be described as a string (or, more precisely, as a URI which is expressed as a string that is well-defined in the `@context`), and all values can be expressed using the types

[34]Naturally, this requires that all involved systems agree on the same vocabulary and definitions of query and response types.

```
{
  "@type" :"korap:token",
  "wrap" :{
    "@type" :"korap:term",
   "layer" :"orth",
    "key" :"corpus",
    "match" :"match:eq"
  }
}
```

Listing 4: JSON-LD representation of a token query.[35]

provided by JSON (see the previous section). For instance, a KoralQuery `token` (as specified in Sec. 5.1) is serialised as the JSON-LD structure in listing 4. More generally, every KoralQuery serialisation is a JSON-LD object with the following obligatory keys:

- `@context` provides the JSON-LD context file
- `query` holds the actual query, i.e. nested KoralQuery structures

Optionally, the following attributes may be included:

- `collection` specifies the metadata for virtual collection creation
- `meta` contains, e.g., displaying directives

Listing 5 thus illustrates a skeleton KoralQuery serialisation . Naturally, the end user of a corpus query engine cannot be expected to enter those relatively verbose structures by hand – in fact, she not even expected to ever be confronted with these. Therefore, the remainder of this section introduces a translation module that is capable of generating KoralQuery serialisations for queries that are issued in any of the three supported QLs that have been presented in Sections 2.1-2.3. This section will first draw on the general process of query translation and then scrutinise the details of processing the individual QLs.

[35]Note that this snippet is incomplete in that it lacks a `@context`, i.e. a binding of the keys and values to types and URIs. In addition, the values for the `@type` keys are undefined.

```
{
  "@context" :
     "http://ids-mannheim.de/ns/KorAP/json-ld/v0.2/context.jsonld",
  "collection" :{},
  "query" :{},
  "meta" :{}
}
```

Listing 5: A skeleton KoralQuery instance

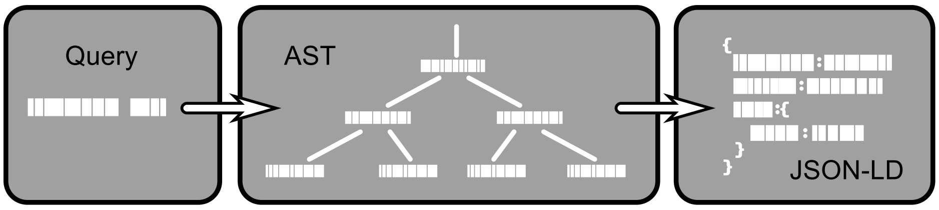

Figure 9.1: Query translation workflow illustration
Image source: (Bański et al., 2013)

9.1 General process

The translation of queries issued in some concrete query language to KoralQuery serialisations is a two-step process. In the first step, the query string is parsed and transformed to an intermediate representation, a so-called *abstract syntax tree* (AST). If the query is well-formed and has been successfully parsed, the AST is translated to KoralQuery-JSON by a QL-specific processor class in the second step. These two steps are illustrated in figure 9.1. Sections 9.1.1 and 9.1.2 discuss the individual steps in more detail.

The two steps correspond to two distinct translation units for each individual QL. Consequently, to be able to add support for a new QL to the system, two units or components must be implemented and plugged into a query engine, namely a grammar to parse the input query and a processor to transform the resulting parses into KoralQuery serialisations.

9.1.1 Query parsing with ANTLR

A query string is a hierarchical and/or sequential ordering of operators and arguments (e.g. the `MORPH()` operator in COSMAS II, which may embed, or be embedded in, other operations or sequences). In order to retrieve this syntactic structure of a query string (and thus to reveal the syntactic relations between query operations), the present work uses the ANTLR framework (Parr and Quong, 1995). With ANTLR, Java code[36] is generated from a provided context-free EBNF grammar that specifies the input language. The generated code then provides methods to recognise an input string and, if successful, derive its syntactic structure and visiting nodes in the parse tree. In the present implementation, QL-specific processor classes (see the following section) make use of this code when they call the generated parsers to obtain a derived AST.

For illustration, listing 6 shows a grammar for a reduced version of the Poliqarp-Plus language. From this grammar, the syntax tree in figure 9.2 is generated for a fairly complex query. The grammar is divided into two separate files, one of which is used to generate a *lexer*, and the other to generate a *parser*. The lexer is used to tokenise the input string and assign pre-terminal labels to the input tokens. Those are then used by the parser to analyse the syntactic structure over the tokens.

In (co-) developing[37] ANTLR grammars for the QLs supported in KorAP, one principle has proven especially desirable to follow. As natural as it might seem, grammars for additional languages should be modelled as closely as possible to the target serialisation in order to keep the development of the AST processors relatively straightforward. In other words, grammar development should pay attention to the structures defined in KoralQuery such that the processors (see section 9.1.2), in the optimal case, can translate the AST nodes directly through a 1-to-1 mapping between AST categories and KoralQuery objects rather than having to be overly context-sensitive in their translations, e.g. because a sequence would not be explicitly marked as such in the grammar (through a proper node) but instead would have to be inferred from an implicit sibling relation of several nodes.

[36]Beyond Java, the current ANTLR release can also generate C# code, while earlier versions of the software also supported several other languages such as C, Perl, Python and Ruby, to name a few.

[37]The grammars for ANNIS QL and COSMAS II QL were developed by Thomas Krause (HU Berlin) and Franck Bodmer (IDS), respectively, and slightly modified by the candidate to satisfy specific needs in KorAP. The grammar for PoliqarpPlus QL was developed by the candidate, based on a previous version by Nils Diewald (IDS).

```
lexer grammar PoliqarpPlusLexer;

LBRCKT : '[';
RBRCKT : ']';
LANGLE : '<';
RANGLE : '>';
LPAREN : '(';
RPAREN : ')';
COMMA  : ',';
EQ     : '=';
FRAME  : 'contains'|'startswith'|'endswith';
WORD   : ([a-zA-Z]|[0-9])+;
```

```
parser grammar PoliqarpPlusParser;

span     : LANGLE WORD RANGLE ;
token    : LBRCKT WORD EQ WORD RBRCKT ;
position : FRAME LPAREN segment COMMA segment RPAREN ;
sequence : segment segment+ ;
segment  : span | token | position | sequence ;
query    : segment ;
```

Listing 6: Toy ANTLR grammar for PoliqarpPlus consisting of a lexer and a parser file.

9.1.2 Processing abstract syntax trees

This second step in the translation workflow amounts to processing the AST that has been derived in the first step and generating corresponding Koral objects for the elements in the AST. The serialisation to JSON-LD is done using the Jackson JSON generator, which translates plain Java objects to JSON data structures. As one of several ways to create JSON from Java, Jackson uses a `java.util.Map` to represent JSON objects and a `java.util.List` to represent JSON arrays. The Koral translator thus uses these Java types as intermediate structures to build the query tree.[38] Thus,

[38]This work uses `java.util.LinkedHashMap` and `java.util.ArrayList` as the concrete implementations for the `Map` and `List` interfaces. The `LinkedHashMap` ensures an ordering of the key-value pairs in the map (by the order in which they are added to the map), which is not strictly necessary given that JSON objects are unordered anyway, but facilitates human readability. Furthermore, recall that JSON objects always carry string keys and values of any type, thus the map is parametrised by `<String,Object>`. Similarly, the list is parametrised by `<Object>`.

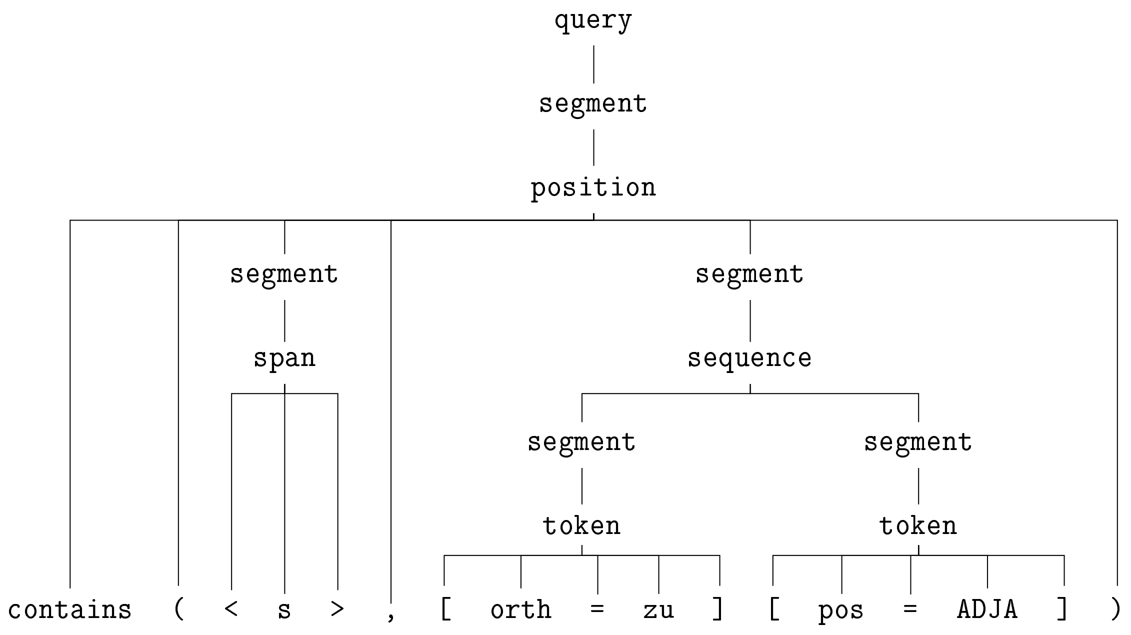

Figure 9.2: ANTLR parse tree for the query `contains(<s>,[orth=zu][pos=ADJA])`

when this section and the following mention the ingestion of KoralQuery types into the JSON tree, this is technically not correct. Instead, the types are represented as Java maps or arrays, and are in fact inserted into a nested structure of maps and arrays.

In general, first a QL-specific processor class calls the respective parser (as generated from the ANTLR grammar) to obtain the AST for a query (see the `process()`[39] function in listing 7). Then, the AST (as exemplified in figure 9.2) is processed in a top-down and depth-first fashion. More concretely, the processing of the AST starts with a function `processNode()` applied to the root, then this function recursively calls itself with its argument's children. Depending on the node type that is being processed at a given recursion, `processNode()` also calls a function that handles the respective node, which amounts to creating an appropriate KoralQuery object and inserting it into the KoralQuery tree. In order to know where exactly to ingest an object in the generated tree, the algorithm makes use of an `objectStack`, onto which any newly created object is pushed upon its creation. This allows descendent ob-

[39]For brevity, this section refers to methods in the QL-specific processor classes using only their names, without the classes they are contained in and without their full signature (including arguments or thrown exceptions).

```java
public class SampleQLProcessor extends Antlr4AbstractQueryProcessor {
    public void process() {
        ParseTree root = SampleQLParser.parse(query); // SampleQLParser is
            generated from ANTLR grammar
        processNode(root);
    }
    public void processNode(ParseTree node) {
        // process
        switch (node.getCategory()) {
            case "sequence":
                processSequence(node);
            case "position":
                processPosition(node);
        }
        // recursively call this method on children
        for (ParseTree child : node.getChildren()) {
            processNode(child);
        }
    }
    private void processPosition(ParseTree node) {
        String frame = node.getChild(0).getText(); // e.g. "contains", always
            first child of position node
        Map<String, Object> positionMap =
            KoralObjectGenerator.makePosition(frame);
        putIntoSuperObject(positionMap); // inserts object into KoralQuery
            tree
    }
}
```

Listing 7: Toy processor class that transforms an AST to a KoralQuery tree[40]

jects, e.g. those generated from a node's children, to be inserted into the `operands` array of that object. When a node is completely processed (i.e. when the recursive `processNode()` method has run on the node and all its descendants), the object generated from this node is popped from the `objectStack` and makes way for its right sibling (if existent) to be inserted into the parent object and ultimately contain new objects in its `operands` as well.

For instance, when processing the `position` node in the parse tree in figure 9.2,

[40]This pseudo-Java code will obviously not compile without import statements etc., and lacks several methods that it references. It is intended to demonstrate the relevant structures and constructs that the QL processor implementations display.

it is passed to a function by the name `processPosition()` which will create a corresponding KoralQuery object (here: a `group` with a `position` operation and `contains` as its frame). This group will then be inserted into the correct place inside the Koral query tree by the method `putIntoSuperObject()`. In this case, the KoralQuery `position` group takes the root of the query tree as it is generated from the first fully processed AST node.[41] Later, when `processNode()` is called with the `span` node, it generates a KoralQuery `span` element and inserts it into the `operands` of the `position` group (see listing 8).

It is of course possible that the user input is not well-formed with respect to the ANTLR grammar. In that case, no parse tree can be derived, and consequently, no KoralQuery tree can be generated. While standard error messages from ANTLR are often relatively cryptic, the AST processors register error listeners which transform the ANTLR error response to a JSON object that contains an error code and an error message, as well as an indication of the offensive symbol that prevents a successful parsing of the input. This JSON object is included under an `errors` attribute at the top level of the KoralQuery tree. The error listeners also apply some logic on their own, e.g. to detect missing arguments for operators or an unbalanced number of brackets or parentheses.

Koral object generation

As KoralQuery objects are by definition not sensitive to the query language from which they are translated, the routines that are responsible for their generation are not located in the individual QL processor classes. Instead, they are provided by the utility class `KoralObjectGenerator` (see the class diagram in figure 9.3). Those object generation methods usually take certain mandatory and optional parameters, depending on the object. For instance, `makePosition()` asks for an array of valid frames[42] that the generated position group is to contain in its `frames` attribute. In the example, only a single string (`"contains"`) is passed to the method, and is thus wrapped in an array by the method and placed in the `position` object.

[41]Note that the `position`'s parent `segment` node only has an auxiliary syntactic function – it does not translate to a particular KoralQuery type.

[42]Recall that the `position` operation is defined to possibly contain several `frames` which are then treated as alternatives. The group will thus match position relations that match any of those frames.

```
{
  "@context" :
     "http://ids-mannheim.de/ns/KorAP/json-ld/v0.2/context.jsonld",
  "query" :{
    "@type" :"korap:group",
    "operation" :"operation:position",
    "frames" :[ "frames:isAround" ],
    "operands" :[ {
      "@type" :"korap:span",
      "key" :"s"
    }, {
      "@type" :"korap:group",
      "operation" :"operation:sequence",
      "operands" :[ {
        "@type" :"korap:token",
        "wrap" :{
          "@type" :"korap:term",
          "layer" :"orth",
          "key" :"zu",
          "match" :"match:eq"
        }
      }, {
        "@type" :"korap:token",
        "wrap" :{
          "@type" :"korap:term",
          "layer" :"pos",
          "key" :"ADJA",
          "match" :"match:eq"
        }
      } ]
    } ]
  }
}
```

Listing 8: KoralQuery serialisation for the PoliqarpPlus query `contains(<s>,[orth=zu][pos=ADJA])`

AST traversal

In addition, the query translation module makes use of an inheritance pattern, which allows for the uniform treatment of the processor classes and provides them with a number of utility methods mostly pertaining to AST traversal. More concretely, in extending the `AbstractQueryProcessor` class, the processors are required to implement the `process()` method and are equipped with several utility fields as well as the initial KoralQuery tree skeleton. As can also be seen in the class diagram, the query processors extend abstract processor classes that correspond to the respective ANTLR versions. This is not a strict requirement for the processors to be used by the `QuerySerializer` (they must only extend the top class), but it facilitates AST traversal by providing methods that, for instance, check whether a node has children of a specific node category.[43]

9.2 PoliqarpPlus QL

The specifications of the KoralQuery protocol and the development of the PoliqarpPlus (PQ+) query language mutually influenced each other to a considerable extent, with the latter serving as a concrete model implementation for the former (see section 7). Therefore, the syntax of a PQ+ query strongly resembles that of its respective KoralQuery (essentially forming a homomorphism), which in turn makes it relatively straightforward to translate this language to KoralQuery – in fact, almost every single element in a PQ+ syntax tree can be mapped directly onto some Koral object, whether simple or complex. This is mainly due to the nested syntactic structure of PQ+ queries, which is reflected in the nesting of Koral objects.

For instance, the PQ+ element that corresponds to a KoralQuery `position` group is specified by a frame identifier which is followed by two arguments that are nested inside parentheses and delimited by a comma. This structure translates well to KoralQuery, as the frame (which clearly identifies the structure as a position) directly goes into the `frames` attribute of the resulting position group, and the two arguments can directly be inserted into the `operands` of that group.

A special case is the `meta` operator in the original Poliqarp language which restricts the search space for the query to the set of documents that satisfy certain metadata constraints. Those constraints are attribute-value pairs (e.g., `author=Smith`

[43]Individual AST traversal classes for ANTLR versions 3 and 4 are necessary as these versions generate different data structures.

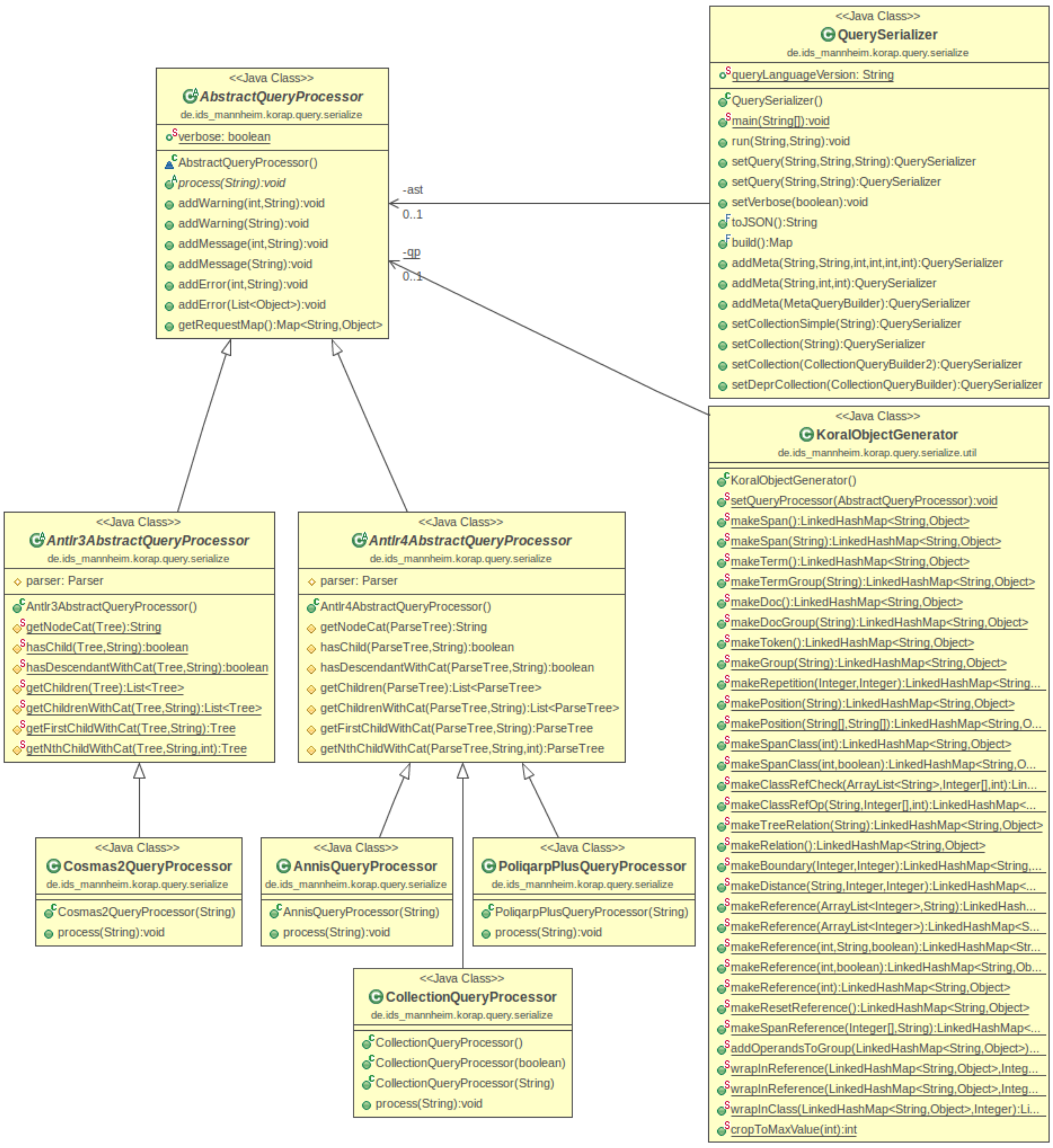
<<Java Class>>
AbstractQueryProcessor
de.ids_mannheim.korap.query.serialize
verbose: boolean
AbstractQueryProcessor()
process(String):void
addWarning(int,String):void
addWarning(String):void
addMessage(int,String):void
addMessage(String):void
addError(int,String):void
addError(List<Object>):void
getRequestMap():Map<String,Object>
<<Java Class>>
QuerySerializer
de.ids_mannheim.korap.query.serialize
queryLanguageVersion: String
QuerySerializer()
main(String[]):void
run(String,String):void
setQuery(String,String,String):QuerySerializer
setQuery(String,String):QuerySerializer
setVerbose(boolean):void
toJSON():String
build():Map
addMeta(String,String,int,int,int,int):QuerySerializer
addMeta(String,int,int):QuerySerializer
addMeta(MetaQueryBuilder):QuerySerializer
setCollectionSimple(String):QuerySerializer
setCollection(String):QuerySerializer
setCollection(CollectionQueryBuilder2):QuerySerializer
setDeprCollection(CollectionQueryBuilder):QuerySerializer
-ast
0..1
-qp
0..1
<<Java Class>>
KoralObjectGenerator
de.ids_mannheim.korap.query.serialize.util
KoralObjectGenerator()
setQueryProcessor(AbstractQueryProcessor):void
makeSpan():LinkedHashMap<String,Object>
makeSpan(String):LinkedHashMap<String,Object>
makeTerm():LinkedHashMap<String,Object>
makeTermGroup(String):LinkedHashMap<String,Object>
makeDoc():LinkedHashMap<String,Object>
makeDocGroup(String):LinkedHashMap<String,Object>
makeToken():LinkedHashMap<String,Object>
makeGroup(String):LinkedHashMap<String,Object>
makeRepetition(Integer,Integer):LinkedHashMap<String...
makePosition(String):LinkedHashMap<String,Object>
makePosition(String[],String[]):LinkedHashMap<String,O...
makeSpanClass(int):LinkedHashMap<String,Object>
makeSpanClass(int,boolean):LinkedHashMap<String,O...
makeClassRefCheck(ArrayList<String>,Integer[],int):Lin...
makeClassRefOp(String,Integer[],int):LinkedHashMap<...
makeTreeRelation(String):LinkedHashMap<String,Object>
makeRelation():LinkedHashMap<String,Object>
makeBoundary(Integer,Integer):LinkedHashMap<String,...
makeDistance(String,Integer,Integer):LinkedHashMap<...
makeReference(ArrayList<Integer>,String):LinkedHash...
makeReference(ArrayList<Integer>):LinkedHashMap<S...
makeReference(int,String,boolean):LinkedHashMap<Str...
makeReference(int,boolean):LinkedHashMap<String,Ob...
makeReference(int):LinkedHashMap<String,Object>
makeResetReference():LinkedHashMap<String,Object>
makeSpanReference(Integer[],String):LinkedHashMap<...
addOperandsToGroup(LinkedHashMap<String,Object>)...
wrapInReference(LinkedHashMap<String,Object>,Integ...
wrapInReference(LinkedHashMap<String,Object>,Integ...
wrapInClass(LinkedHashMap<String,Object>,Integer):Li...
cropToMaxValue(int):int
<<Java Class>>
Antlr3AbstractQueryProcessor
de.ids_mannheim.korap.query.serialize
parser: Parser
Antlr3AbstractQueryProcessor()
getNodeCat(Tree):String
hasChild(Tree,String):boolean
hasDescendantWithCat(Tree,String):boolean
getChildren(Tree):List<Tree>
getChildrenWithCat(Tree,String):List<Tree>
getFirstChildWithCat(Tree,String):Tree
getNthChildWithCat(Tree,String,int):Tree
<<Java Class>>
Antlr4AbstractQueryProcessor
de.ids_mannheim.korap.query.serialize
parser: Parser
Antlr4AbstractQueryProcessor()
getNodeCat(ParseTree):String
hasChild(ParseTree,String):boolean
hasDescendantWithCat(ParseTree,String):boolean
getChildren(ParseTree):List<ParseTree>
getChildrenWithCat(ParseTree,String):List<ParseTree>
getFirstChildWithCat(ParseTree,String):ParseTree
getNthChildWithCat(ParseTree,String,int):ParseTree
<<Java Class>>
Cosmas2QueryProcessor
de.ids_mannheim.korap.query.serialize
Cosmas2QueryProcessor(String)
process(String):void
<<Java Class>>
AnnisQueryProcessor
de.ids_mannheim.korap.query.serialize
AnnisQueryProcessor(String)
process(String):void
<<Java Class>>
PoliqarpPlusQueryProcessor
de.ids_mannheim.korap.query.serialize
PoliqarpPlusQueryProcessor(String)
process(String):void
<<Java Class>>
CollectionQueryProcessor
de.ids_mannheim.korap.query.serialize
CollectionQueryProcessor()
CollectionQueryProcessor(boolean)
CollectionQueryProcessor(String)
process(String):void

Figure 9.3: Class diagram for the query language processors. Private methods, which mostly pertain to the specific processing of AST nodes, are excluded from this view for brevity.

or `pubDate<2010`) and are translated to the `key` and `value` attributes of `doc` objects, which in turn are located under the top-level `collection` attribute (thus specifying a virtual collection, cf. section 6.1). If further constraints are provided directly by a virtual collection definition in the KorAP interface, those are subsumed along with the constraints provided in the PQ+ query under a `docGroup` conjunction.

Another language element that does not have a direct translation in KoralQuery's `query` object is the alignment operator `^`. As the function of this operator is restricted to a displaying directive and as it has no impact on the result set in any way, it is serialised to an object in the `meta` field, bearing a `class` attribute to refer to a class in the KoralQuery tree. The starting position of this class serves as an anchor for the alignment.

9.3 COSMAS II QL

The translation of COSMAS II queries to KoralQuery is not as straightforward as in the case of PQ+. While the original syntax of the QL is not actually that far away from the metalanguage (see examples 16-18 in section 2.2 for instances of nested structures in the language), the ANTLR v3 grammar which is used to parse the queries applies a substantial amount of rewriting which, on the one hand, makes the AST more readable for humans, but on the other hand distorts the original syntactic structure and moves it away from KoralQuery.[44] Thus lacking the syntactic similarities with Koral, there are a number of constructions in the language that cannot be mapped to the metalanguage in a context-free manner.

An example of this are implicit sequences, i.e. two or more sibling nodes in the parse tree that form a `sequence` group without any ancestor node explicitly specifying this. Of course, not all siblings in a tree make for a sequence – instead, only a certain set of node types are "sequentiable", such that the decision whether to group a set of nodes into a sequence must be based on the respective types of the nodes. To make things more complicated, this decision must be made before finally visiting all these nodes, or in fact before generating the KoralQuery object for the first of the siblings and ingesting it into the KoralQuery tree. Thus, the applied mechanism is

[44]Of course, it would have been possible to write a new grammar to parse COSMAS II queries, however writing a grammar for a language as complex as this amounts to a sizeable load of work. However, given the following complications in translating the ASTs and the aggravating fact that the grammar is not compatible with ANTLR v4 but requires v3 to be integrated into the workflow as an additional resource, it might be worthwhile to write a new ANTLR v4 grammar for COSMAS II in the future.

to check for any first (leftmost) child of any node whether it belongs to the set of sequentiable node and whether any of its siblings also meet this requirement. In the positive case, a `sequence` group will be introduced and the node as well as its sequentiable siblings are inserted into the operands of that sequence.

Handling `MORPH()` arguments

The support of COSMAS II in KoralQuery is slightly constrained in the expressiveness of the `MORPH()` operator, which originally provides access to three different sets of morphosyntactic annotation in the corpus data. Crucially, the syntax of this operator only expects a list of tags as arguments, without any explicit indication of the linguistic categories that these values belong to – neither are such attribute-value relations explicated in the data. Instead, the token in the corpus are assigned a list of 'plain' tags. This means that the tags must not be ambiguous with respect to grammatical category, i.e. the same tag X may not be used to denote a value for category C_1 and another value for category C_2. However, there are in fact ambiguities across annotations, e.g. SUB is used in the TreeTagger data to denote a substituting pronoun, in the Connexor annotation to denote a verb in subjunctive mood, and in the data annotated with MECOLB the tag denotes a subordination. This ambiguity, however, does not pose a problem for the system, as it requires the user to select one of those annotations prior to the submission of the query.

Yet, as KorAP has no way of knowing which of the alternatives the user has in mind, the system cannot infer which grammatical category (i.e. which `key` in a standard morphology layer) a given tag implies.[45] Consequently, the KoralQuery translation module expands the expressiveness of the `MORPH()` operator and requires the user to make explicit indications of at least the `layer` and the `key`, optionally a `foundry` and a `value`.[46] The operator thus provides access to all `term`-based annotations in the underlying corpus data. By allowing regular expressions to be used for the `key` and `value` (denoted by quotation marks), the query translation module also covers the shortcut tags that are used in the COSMAS II system to denote a set of tags, e.g. `V` which is used for all verb tags in the STTS.

[45]In theory, the category could in many cases be inferred from a sufficient amount of data, e.g. when several values are provided as the arguments, one of which is unambiguous with respect to the annotation source (as those may not be mixed).

[46]Foundry, layer, key and value are specified using the same syntax as the one that is used as for the specification of *terms* in PoliqarpPlus tokens, i.e. they follow the scheme `[foundry/layer=key:value]`. The equality operator (=) may be replaced with the inequality operator (`!=`) to negate the constraint.

Partial matching

Another special COSMAS II property is the matching behaviour in, for example, sequences. In contrast to the Poliqarp query `der []{0,4} Mann`, which matches any span of up to six tokens that starts with *der* and ends with *Mann*, the corresponding COSMAS II query `der /+w1:5 Mann` only matches the explicitly stated operands of the operation, i.e. only the tokens *der* and *Mann*. While the match can be extended to the whole span using the `#ALL()` operator, the default behaviour calls for a special treatment of such partial matches, especially when the sequence is embedded in another operation that works on the returned match of the sequence. As a solution, the Koral translator declares classes on the operands that both bear the same class number. By means of these classes, operations that require a distinction between partial matches and full span matches (e.g. overlaps with other spans) can be accounted for using a `classRefCheck`, see also the following paragraph on COSMAS II position queries. If necessary, the classes can also be used to reduce the match span with a `reference` object for that class around the operation.

Position options

The described partial matching behaviour has direct consequences for another COSMAS II feature, and those consequences also demand a few measurements to be taken in the translation to KoralQuery. Concretely, the COSMAS II operators `#IN` (inclusion) and `#OV` (overlap) as well as their relatively fine-grained options cannot be fully covered by the position frames provided by Koral. As for the operators themselves (disregarding their options), overlap in COSMAS II is defined as a non-empty intersection of two spans (see footnote 9), while inclusion requires the first match to be a subset of the second match. Thus, `X #OV Y` holds if any token in the match of `X` is also in the match of `Y`. Now, as matches can be discontinuous (see the paragraph on partial matching), KoralQuery's `overlaps` frame (or any other frame) alone is not suitable for covering this relation, as frames only impose constraints on the starting and ending positions of the spans in question. As a remedy, KoralQuery uses the `classRefCheck` attribute of the `class` operation, which evaluates the set-theoretic relationship between the sets of tokens covered by two classes. Thus, for `#OV`, this `classRefCheck` receives the value `intersects`, which denotes a non-empty intersection between the matches of the operands. The class operation is then wrapped around the position group.

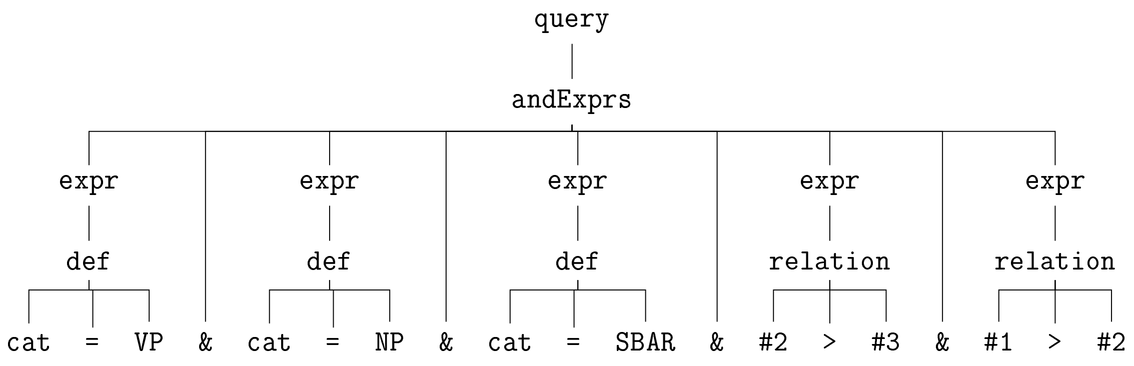

Figure 9.4: ANTLR parse tree for the ANNIS QL query
`cat="VP" & cat="NP" & cat="SBAR" & #2 > #3 & #1 > #2`

As exemplified in (17), the operators `#IN` and `#OV` also allow the specification of certain options which, in addition to the set-theoretic relationship between the operands, further restrict the positional relation. For instance, `X #IN(L) Y` requires that the match of `X` be fully included and left-aligned in the match of `Y`. To account for those cases, a combination of `frames` and `classRefCheck` is used (in this case, `frames:startswith` and `classRefCheck:includes`, with the `operands` order switched). Using these two attributes with their values as defined in the second part of this thesis, all position queries in COSMAS II are covered.

9.4 ANNIS QL

As demonstrated in section 2.3, the syntactic layout of ANNIS QL to query graph-based annotations differs strongly from the nested structures of PQ+, COSMAS II QL and, most importantly, KoralQuery. With queries being sequences of variable declarations and relation constraints that are connected by conjunction (with exception of the occasional disjunctions nested inside the conjunctions), the used ANTLR grammar generates very flat parse trees. Thus, the processor needs to find a way to transform the flat parse into a nested structure of binary relations as prescribed by the Koral protocol. This calls for a very different approach to translation, most notably to identifying the correct structure between the generated objects, as we can not rely on (more or less) directly mirroring the AST structure in the KoralQuery tree.

For instance, consider the (rather simple) query and the parse tree generated from it, given in figure 9.4. This query defines three objects (tokens of different surface forms) and asks for two relations that hold over these tokens (the second token directly precedes the third, the same holds between the first token and the second). The strategy that the processor class applies here is the following: first, it scans the entire tree for any defined objects and counts how often each of these are referenced within relation statements (in the example, the token *corpus*, which is identified by `#2`, is referenced twice, the others once). Based on their '`#`'-identifiers, the defined objects are stored in a table with those counts. Then, in a second traversal of the tree, the relations are processed in a left-to-right order and wrapped around each other, retrieving the operands referenced by the relations from the table.[47] As mentioned above, the difficulty lies in finding a way to correctly nest the relations. This is not trivial, as in KoralQuery nesting means inserting one object into the `operands` of another, consequently embedding one linguistic structure into another (e.g. a dependency relation into a sequence), while these structures were specified as completely equal and independent in the original query. To make things more complicated, most KoralQuery objects take a fix number of operands (usually one or two), which becomes problematic when one object needs to be inserted into another although the latter is already saturated with its actual operands.[48]

As a solution to this problem, KoralQuery provides the `reference` type by means of which certain sub-matches of an object (e.g. only one operand) can be referred to. Thus, in the above example, the second sequence relation will be wrapped around (rather than inserted into) the first sequence, embedding it as an operand, but only referring to the shared object (here: the token *query*, identified by `#2`) using a `reference` (see listing 9).

In the concrete implementation, the ANNIS QL processor addresses this problem by maintaining an `operandStack`, where all pre-final KoralQuery objects are stored for retrieval during the processing of later objects. Generally, the first processed relation is always 'filled' with both its operands, whereas for following objects, the operand references are evaluated for either being references to 'new' objects (that have not been inserted into the KoralQuery tree) or whether they can be expressed through a reference around the first object on the `operandStack`. In retrieving the

[47]However, there may also be shortcut token/span definitions directly within the scope of the operator, as in example 25. In these cases, no retrieval from the table is required, and the object is necessarily only used once as it cannot be referenced outside the relation.

[48]In the example query in figure 9.4, one of the two relations needs to be inserted into the other. However, the `group` objects with the `"operation:relation"` that are created from the $>$ relations only take two operands, but both relations already have to arguments.

```
{ "@context" :
   "http://ids-mannheim.de/ns/KorAP/json-ld/v0.2/context.jsonld",
  "query" :{ "@type" :"korap:group",
    "operands" :[ { "@type" :"korap:span",
          "key" :"VP",
          "layer" :"c"
        },
        { "@type" :"korap:reference",
          "classRef" :[ 1 ],
          "operands" :[ { "@type" :"korap:group",
              "operands" :[ { "@type" :"korap:group",
                  "classOut" :1,
                  "operands" :[ { "@type" :"korap:span",
                      "key" :"NP",
                      "layer" :"c"
                    } ],
                  "operation" :"operation:class"
                },
                { "@type" :"korap:span",
                  "key" :"SBAR",
                  "layer" :"c"
                }
              ],
              "operation" :"operation:relation",
              "relation" :{ "@type" :"korap:relation",
                 "wrap" :{ "@type" :"korap:term",
                   "layer" :"c"
                 }
              }
            } ],
          "operation" :"operation:focus"
        }
      ],
    "operation" :"operation:relation",
    "relation" :{ "@type" :"korap:relation",
       "wrap" :{ "@type" :"korap:term",
         "layer" :"c"
       }
    }
  }
}
```

Listing 9: KoralQuery serialisation for the ANNIS query `cat="VP" & cat="NP" & cat="SBAR" & #2 > #3 & #1 > #2`

operands for the second sequence in the example, the object for the first operand reference (`#1`) is thus taken from the table mentioned above, while the second operand reference (`#2`) is recognised to have been processed before (in the first sequence), such that a `reference` with the class number assigned to the operand is wrapped around the first object on the `operandStack` and inserted as an operand for the second sequence. Crucially, the object thus created is inserted into the KoralQuery tree only if it corresponds to the final relation in the query (which it does here) – otherwise, it will itself be pushed back onto the `operandStack`.

When both operands of a relation have already been added to previous relations, it is also necessary to create 'empty' references which do not have any `operands` as a scope for their `focus` operation, but only provide a cross-reference to a class elsewhere in the tree. This necessity is grounded in the apparent circumstance that the wrapping relation must not re-define any of the operands. If a certain operand is defined twice in the KoralQuery tree (as an operand of two distinct operations), it is possible that two separate matches are retrieved for those two definitions, when actually the operations are required to share an operand. Using a `reference`, Koral ensures that both operations in fact take scope over the same match.

In the contrary case, i.e. when none of the operands of a relation have previously been processed, a similar problem arises. With exception of the first relation, of course, those relations cannot simply be wrapped around previous relations and reference their operands, because they do not share any common operands. Therefore, the ANNIS QL processor implements a mechanism to queue such relations until a following relation has provided the definition of one of the operands, thus allowing for referencing them.[49]

9.5 Virtual collections

Limiting the search scope of a query to virtual collections of documents through the specification of metadata constraints is supported by KoralQuery through the `collection` attribute. This contains a `doc` objects (or logical conjunctions of `doc` and `docGroup` objects) which is generated from a virtual collection query. To this end, an additional language is defined, which is processed in the same general manner

[49] An example of this is the query `A & B & C & D & #1 . #2 & #3 . #4 & #1 > #3`. Here, the second relation (`#3 . #4`) cannot be wrapped around the first as they do not share any operands. Instead, this relation is queued until the third relation is processed and provides a definition of `#3`, which can then be referenced by the queued relation.

as the 'proper' QLs, i.e. parsing the query string using an ANTLR grammar (here, ANTLR v4 is used) and deriving a KoralQuery representation by means of a specific processor (see also the class diagram in figure 9.3). This language is used in KorAP, although any other language could be defined (along with an ANTLR grammar and a processor) and plugged into the `QuerySerializer`, in a similar fashion as new QLs can be added.

The Koral specifications for virtual collection queries allow to address any of the metadata fields of a document using the `key` attribute and to constrain its value using the `value` attribute. The `match` attribute specifies the relation between the `key` and the `value`, and takes the possible values specified in 6.1. Notably, KoralQuery allows the definition of virtual collections based on tokens or phrases contained in, for instance, the title string using the operator ~ (or `!~` to negate such a constraint). For numerical values, e.g. the publication date of a document, KoralQuery also allows inequations, enabling search for all documents published, e.g., after a certain day.[50]

The grammar of the provided virtual collection query language is very close to the term specifications in Poliqarp. More concretely, a key and a value are connected through an operator indicating the relation between them, such as equality or containment. Then, these constraints can be connected with other constraints using the logical operators `&` (conjunction) and `|` (disjunction). The query is thus a nested structure which maps directly onto the corresponding KoralQuery, such that the translation task is relatively easy and context-free, as is the case for Poliqarp-Plus. The only aspect that requires some context-sensitivity is to ensure that only numerical values are used with inequation operators ($<$, $>$, $\leq$ and $\geq$).

10 Conclusions

10.1 Contributions of the thesis

The present thesis has introduced KoralQuery, a general protocol for the representation of queries to linguistic corpora. The protocol defines a range of different linguistic and operational as well as parametric types which, nested into each other, are capable of expressing highly complex linguistic structures. The protocol forms a wide target space for one-way mappings from existing corpus query languages

[50]A date is specified using an RFC-3339 string with a granularity of year, month or day.

```
{
  "@context" :
     "http://ids-mannheim.de/ns/KorAP/json-ld/v0.2/context.jsonld",
  "collection" :{
    "@type" :"korap:docGroup",
    "operation" :"operation:and",
    "operands" :[{
      "@type" :"korap:doc",
      "key" :"title",
      "value" :"Gerrard",
      "match" :"match:contains"
    }, {
      "@type" :"korap:doc",
      "key" :"pubDate",
      "value" :"2005-05-25",
      "type" :"type:date",
      "match" :"match:eq"
    }]
  },
  "query" :{
    "@type" :"korap:group",
    "operation" :"operation:sequence",
    "operands" :[{
      "@type" :"korap:token",
      "wrap" :{
        "@type" :"korap:term",
        "layer" :"orth",
        "key" :"zu",
        "match" :"match:eq"
      }
    }, {
      "@type" :"korap:token",
      "wrap" :{
        "@type" :"korap:term",
        "layer" :"pos",
        "key" :"ADJA",
        "match" :"match:eq"
      }
    }]
  }
}
```

Listing 10: KoralQuery serialisation for the PQ+ query `[orth=zu][pos=ADJA]` and the virtual collection definition `title~Gerrard & pubDate=2005-05-25`

and can thus be seen as a reference for the assessment of their expressive power in the sense of the CQLF endeavour. However, a claim that the types and structures defined in KoralQuery be exhaustive and that therefore the mentioned target space be a definitive and comprehensive collection of all possible linguistic structures is certainly more than optimistic. On the other hand, the protocol does in fact cover the functional features of three very distinctive QLs, whose combined feature space covers – with the exception of universal quantification – the needs expressed by grammarians and lexicographers in a use case analysis by Frick et al. (2012). Notably, the protocol manages to unify the feature sets of those three languages, which differ greatly in their expressive powers, while maintaining a common syntactic structure to represent queries (even though one of those languages, ANNIS QL, differs strongly in its syntax from the nested structure of KoralQuery).

In addition to the rather theoretically oriented application of providing a model feature space for other QLs, corpus query systems such as KorAP may use the KoralQuery protocol to express highly complex linguistic queries at a very abstract and normalised level, with a well-defined (but still neutral in terms of linguistic theory) set of types and operations. This allows a query system that opts to employ KoralQuery as its internal representation of queries to communicate with other corpus query engines over a shared protocol. Within this scenario, systems that, for instance, only support a subset of the types and operations defined in KoralQuery have the possibility to clearly communicate the supported features using normalised definitions of those.

The second central contribution of this thesis is the Koral translation module, which encodes queries from currently three supported corpus query languages to KoralQuery. Koral enables corpus query engines to work independently of particular query languages, and thus to potentially support several of those with almost no additional overhead. To this end, the Koral translator employs a two-step process that consists of (i) parsing a query string using a context-free grammar and the ANTLR framework and (ii) the processing of the resulting parse tree by generating KoralQuery objects for the individual nodes in that tree. While it is not strictly necessary to follow this approach in providing support for additional query languages, the utility classes that Koral provides for the traversal of parse trees generated from ANTLR v3 and v4 grammars pose a considerable simplification of the second step of the translation process.

Both the conception of KoralQuery as well as the development of the Koral translator have taken place in an interplay and in close cooperation with the KorAP project, in particular with the development of KorAP's backend technology. Consequently, the introduced format as well as the translation units have been tested in practice and will be directly used in the KorAP software and available to its users upon KorAP's release. The development of the translation module was test-driven, with test cases likewise defined in coordination with the backend development and, simultaneously, informed by the official documentations of the supported query languages.

10.2 Future work

As outlined in the previous section, the support of "universal quantification", which was identified by Frick et al. (2012) as a desired feature of corpus query languages but is not present in any of the three QLs presented here, has not found its way into the KoralQuery specifications. Adding support for this feature (and possibly implementing a corresponding operator in PoliqarpPlus) is thus a natural item for future work. KoralQuery may also reach its limits when applied to corpora of other modalities, e.g. corpora of spoken language, where different levels of analysis (such as phonemes) are of interest.

Beyond the core KoralQuery protocol, the support for additional query languages by the Koral translator is an obvious work item. The translation component may further profit from a mechanism that gives feedback to the user by means of generating natural language paraphrases for complicated queries. Especially inexperienced users with no or little background in a particular query language may feel uncertain about the 'correctness' of a query, i.e. whether it expresses exactly what they would like to know. Addressing this issue, the hierarchical structure of KoralQuery serialisations would make it relatively easy to generate feedback to the user in a controlled subset of English (or any other natural language) of the form: "You searched for a sequence of two tokens. The first token has the surface form 'der', the second token has the part-of-speech tag 'ADJA'".

10.3 Acknowledgements

The first part of the present thesis touches on work by colleagues at IDS Mannheim, most notably Elena Frick and Piotr Bański. The second and third part, i.e. the definition of the KoralQuery protocol and the development of the translation module,

are work by the author of the thesis. The author acknowledges the valuable aid by various colleagues in the specification of KoralQuery, most notably Nils Diewald, Michael Hanl and Eliza Margaretha, who also provided support for a fourth query language, a subset of the Contextual Query Language (CQL). Elena Frick and Piotr Bański provided valuable comments from a CQLF perspective.

List of Figures

List of Tables

List of Listings

References

Bański, P., Bingel, J., Diewald, N., Frick, E., Hanl, M., Kupietz, M., Pęzik, P., Schnober, C., and Witt, A. (2013). KorAP: the new corpus analysis platform at IDS Mannheim. In Vetulani, Z. and Uszkoreit, H., editors, *Human Language Technologies as a Challenge for Computer Science and Linguistics. Proceedings of the 6th Language and Technology Conference*, Poznań. Fundacja Uniwersytetu im. A. Mickiewicza.

Bański, P., Diewald, N., Hanl, M., Kupietz, M., and Witt, A. (2014). Access Control by Query Rewriting: the Case of KorAP. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

Bański, P., Fischer, P. M., Frick, E., Ketzan, E., Kupietz, M., Schnober, C., Schonefeld, O., and Witt, A. (2012). The new IDS corpus analysis platform: Challenges and prospects. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*, pages 2905–2911.

Belica, C., Herberger, A., and al Wadi, D. (1992). COSMAS. Linguistische Datenverarbeitung. *Institut für deutsche Sprache, Mannheim*.

Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The Semantic Web. *Scientific American*, 284(5):28–37.

Bird, S., Buneman, P., and Tan, W. C. (2000). Towards a query language for annotation graphs. *CoRR*, cs.CL/0007023.

Bird, S. and Liberman, M. (1999). Annotation graphs as a framework for multidimensional linguistic data analysis. *CoRR*, cs.CL/9907003.

Bodmer, F. (1996). Aspekte der Abfragekomponente von COSMAS II. *LDV-INFO*, 8:142–155.

Bodmer, F. (2005). COSMAS II. *Recherchieren in den Korpora des IDS. Sprachreport*, 3(2005):2–5.

Bosch, S., Key-Sun, C., De La Clergerie, E., Chengyu Fang, A., Faass, G., Lee, K., Pareja-Lora, A., Romary, L., Witt, A., Zeldes, A., and Zipser, F. (2012). <tiger2/> as a standardized serialisation for ISO 24615 – SynAF. In *Proceedings of the 11th*

International Workshop on Treebanks and Linguistic Theories (TLT11), pages 37–60, Lisbon, Portugal.

Cassidy, S. and Harrington, J. (1996). Emu: An enhanced hierarchical speech data management system. In *Proceedings of the Sixth Australian International Conference on Speech Science and Technology*, pages 361–366.

Cassidy, S. and Harrington, J. (2001). Multi-level annotation in the emu speech database management system. *Speech Communication*, 33(1):61–77.

Christ, O., Schulze, B. M., Hofmann, A., and König, E. (1999). *The IMS Corpus Workbench: Corpus Query Processor (CQP): User's Manual*. IMS, Stuttgart University.

Dekhtyar, A. and Iacob, I. E. (2005). A framework for management of concurrent xml markup. *Data & Knowledge Engineering*, 52(2):185–208.

Durusau, P. and O'Donnell, M. B. (2002). Concurrent markup for XML documents. In *Proc. XML Europe*.

ECMA-404 (2013). The JSON Data Interchange Format. Standard ECMA-404. `http://www.ecma-international.org/publications/standards/Ecma-404.htm`.

Evert, S. (2005). *The CQP Query Language Tutorial*. IMS, Stuttgart University.

Fiehler, R., Wagener, P., and Schröder, P. (2007). Analyse und Dokumentation gesprochener Sprache am IDS. *Kämper, Heidrun/Eichinger, Ludwig M.(Hg.): Sprach-Perspektiven. Germanistische Linguistik und das Institut für Deutsche Sprache. Tübingen: Narr. S*, pages 331–365.

Frick, E., Schnober, C., and Bański, P. (2012). Evaluating query languages for a corpus processing system. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*, pages 2286–2294.

Harrison, M. A. (1978). *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc.

Heid, U. and Mengel, A. (1999). Query Language for Research in Phonetics. In *International Congress of Phonetic Sciences (ICPhS 99)*, pages 1225–1228, San Francisco.

Ide, N. (1998). Corpus Encoding Standard: SGML guidelines for encoding linguistic corpora. In *Proceedings of the First International Language Resources and Evaluation Conference*, pages 463–70. Citeseer.

Ide, N. (2013). Web Service Exchange Protocol: Preliminary Proposal. Slides presented at ISO TC37 SC4 WG1 meeting, 2nd September 2013. `http://www.anc.org/LAPPS/EP/Meeting-2013-09-26-Pisa/overview-ep-2013-09-26-pisa.pdf`.

Ide, N., Bonhomme, P., and Romary, L. (2000). An XML-based Encoding Standard for Linguistic Corpora. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 825–830.

Ide, N. and Suderman, K. (2007). Graf: A graph-based format for linguistic annotations. In *Proceedings of the Linguistic Annotation Workshop*, pages 1–8. Association for Computational Linguistics.

ISO 24611:2010 (2010). Language resource management – Morpho-syntactic annotation framework (MAF).

ISO 24612:2012 (2012). Language resource management – Linguistic annotation framework (LAF).

ISO 24615:2010 (2010). Language resource management – Syntactic annotation framework (SynAF).

Jespersen, O. (1930). A new science: Interlinguistics. *Psyche*, 11 [3]:57–67.

Kupietz, M., Belica, C., Keibel, H., and Witt, A. (2010). The German Reference Corpus DeReKo: A Primordial Sample for Linguistic Research. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC 2010)*. European Language Resources Association (ELRA).

Kupietz, M. and Lüngen, H. (2014). Recent Developments in DeReKo. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

König, E. and Lezius, W. (2003). The TIGER language – A Description Language for Syntax Graphs. Formal Definition. Technical report, IMS, Stuttgart University.

Lüdeling, A. and Kytö, M. (2008). Corpus linguistics: An international handbook. *Handbücher zur Sprach- und Kommunikationswissenschaft*.

Mann, W. and Thompson, S. (1988). Rhetorical structure theory: Towards a functional theory of text organization. *Text*, 8(3):243–281.

Mueller, M. (2010). Towards a digital carrel: A report about corpus query tools. `http://hdl.handle.net/10932/00-01FE-EB35-D6AD-C201-6`.

OASIS (2013). searchRetrieve: Part 5. CQL: The Contextual Query Language Version 1.0. `http://www.loc.gov/standards/sru/cql/spec.html`.

Parr, T. J. and Quong, R. W. (1995). ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Przepiórkowski, A., Krynicki, Z., Debowski, L., Wolinski, M., Janus, D., and Banski, P. (2004). A search tool for corpora with positional tagsets and ambiguities. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 1235–1238. European Language Resources Association (ELRA).

Rosenfeld, V. (2010). An implementation of the Annis 2 query language. Technical report, Humboldt-Universität zu Berlin.

Sperberg-McQueen, C. and Huitfeldt, C. (2008). Markup discontinued: Discontinuity in texmecs, goddag structures, and rabbit/duck grammars. In *Proceedings of Balisage: The Markup Conference*, volume 1.

Sporny, M., Lognley, D., Kellogg, G., Lanthaler, M., and Lindström, N. (2014). JSON-LD 1.0. A JSON-based Serialization for Linked Data. Technical report, W3C.

Stührenberg, M. and Goecke, D. (2008). SGF – an integrated model for multiple annotations and its application in a linguistic domain. In *Proceedings of Balisage: The Markup Conference*, volume 1.

Zipser, F. (2009). Entwicklung eines Konverterframeworks für linguistisch annotierte Daten auf Basis eines gemeinsamen (Meta-)modells. Master's thesis, Humboldt-Universität Berlin.

Zipser, F. and Romary, L. (2010). A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010.*