Domain-Specific Query Languages

By

Yenchi Jaskolski

A thesis submitted to Macquarie University
for the degree of Master of Research
Department of Computing
December 2015

MACQUARIE
UNIVERSITY
SYDNEY ~ AUSTRALIA

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Yenchi Jaskolski

Acknowledgements

```
@prefix ack: <http://127.0.0.1/thesis/acknowledgements/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ack:1 rdf:type ack:person;
    ack:label "Steve Cassidy";
    ack:position ack:supervisor.

ack:2 rdf:type ack:person;
    ack:label "Mark Jaskolski";
    ack:position ack:partner;
    ack:position ack:best_friend.

ack:3 rdf:type ack:person;
    ack:label "Mum";
    ack:position ack:mum.

ack:4 rdf:type ack:person;
    ack:label "Myles Cover";
    ack:position ack:best_friend.
```

Abstract

Data is a valuable commodity, and while there are many methods for effectively processing data, at the core of many of these is a query language – either general purpose, such as SQL and SPARQL, or domain-specific. Examples of the latter are query languages for linguistic annotations such as LPath+ and EmuQL.

Domain-specific query languages allow greater expressiveness as they are more suited to a chosen data model. But what is the best way of creating such a query language, so as to ensure that the query system created is most suited to the task? Why do the query languages in the same domain vary so widely? Are some more expressive than others, or are they all just different surface forms of the same abstract query language?

This study analyses and compares two query languages specific to the domain of linguistic annotations as examples of domain-specific query languages. It examines the structure of the query languages, as well as their levels of expressiveness in comparison to each other, relative to the questions that are being asked. Versions of each language are implemented to work with a unified data model, and so the questions asked of each will be the same.

The end result is that whilst both can answer most questions and can be reduced to a singular query model, they may be limited by the data and interpreter used.

Contents

Acknowledgements **v**

Abstract **vii**

List of Figures **xi**

1 Introduction **1**

2 Literature Review **3**

2.1 Introduction 3
2.2 Expressiveness of Language 3
2.3 Domain-Specific Languages 4
2.4 Query Systems 6
2.4.1 Data storage 6
2.4.2 Query Languages 7
2.4.3 Query Algebra 7
2.5 RDF Query System 8
2.6 Conclusion 10

3 A Study in the Domain **11**

3.1 Introduction 11
3.2 Representing Annotations as XML 14
3.3 Representing Annotations as RDF 16
3.4 Annotation Data Model 16
3.4.1 LPath+ 16
3.4.2 EmuQL 17
3.4.3 Design Analysis 19
3.5 Questions in the domain 21
3.5.1 Basic 21
3.5.2 Sequence 21
3.5.3 Dominance 21
3.5.4 Transitive Closure 22
3.5.5 Wildcard 22
3.5.6 “Not” And “Or” 23
3.5.7 Nested Queries 24

3.5.8 Finding Siblings 24
3.6 Conclusion 25

4 Case Study: Implementation 27
4.1 Data 27
4.2 Software 28
4.3 Parsing A Query 28
4.3.1 EmuQL 29
4.3.2 LPath+ 30
4.4 Understanding A Query 32
4.4.1 Shared Internal Tree 33
4.5 Reconstructing The Query 34
4.6 Worked Example 35
4.7 Query Results 37
4.8 Conclusion 40

5 Discussion 41
5.1 Language Design and Expressiveness 41
5.1.1 EmuQL 42
5.1.2 LPath+ 43
5.2 Internal AST Representation 44
5.2.1 Timing Queries 45
5.3 Future Work 45

6 Conclusion 47

A Appendix - Language Grammars 49

References 53

List of Figures

2.1 Chomsky hierarchy 4
2.2 Example of SQL query 7
2.3 Example of RDF as a graph 9
2.4 Example of RDF in Turtle / N-triple notation 9
2.5 Example of SPARQL query 9

3.1 Example of linguistic data 12
3.2 Example of tiered data 13
3.3 Example of XML 15
3.4 Example of XQuery query 15
3.5 Example of XPath query 15
3.6 LPath+ Axes 18
3.7 Example of EmuQL query 18
3.8 Simple and complex EmuQL queries 18
3.9 EmuQL features and relations 19
3.10 Simple and complex LPath+ queries 20

4.1 Subset of EmuQL grammar 30
4.2 Subset of LPath+ grammar 31
4.3 Original LPath+ versus implemented LPath+ 31
4.4 Original LPath+ versus implemented LPath+ with wildcard 32
4.5 Shared internal AST 34
4.6 Example of SPARQL with transitive closure 38

5.1 Example of equivalent queries in LPath+ and XPath 44

A.1 Implemented Emu query language grammar 50
A.2 Implemented LPath+ grammar 51

1 Introduction

There is a lot of data in the world, and this amount is ever-increasing as people continually create and share more content. New technologies are often being introduced that encourage the retention of data rather than removal or update, which only adds to the growing mass of available data. As the amount of available data increases, it becomes apparent that an effective way of managing and processing this data is required. There are a lot of different ways to manage this data. At the core of many of these methods is a query system allowing a user to select the parts of the data that are relevant to a particular question.

Query systems are one of many tools used for data analysis. They allow users to query a database, as well as add, remove and modify the stored data [1]. They do this through the use of query languages. Different query languages are suited to different query systems and databases. The appropriate query system to use will depend on the data model being queried. The data model chosen for a data set can be abstract, generic and removed from the actual content of the data. For example, a relational data model is a general purpose data model [2] that is independent of any specific domain. On the other hand, it is possible to create a data model specific to the domain being represented, for example, an annotation data model for linguistic annotations.

A data set is not limited to just one data model. Data can be represented in a variety of ways, and so it is possible to represent the same data using a variety of different data models. The model chosen is often dependent on the purpose and focus of the database, as well as the data that is to be stored in the database. In the same way that data can be represented using a variety of data models, a data model is not restricted to being queried by just one query language. While some query languages are more suited to specific data models, e.g. SQL is specific to relational databases [2], other query languages can be used on data models that they were not designed for, e.g. LPath+ can be compiled to work on the more generic relational model [3]. Different query languages are available to different data models. Query languages not designed for specific data model can be adapted for use on them. This allows a user to mix-and-match the components of a query system – the data model and the query language – depending on their needs and what is most suitable for their data.

The range of possible query languages and data models for a query system allows for a range of different questions to be asked and for a range of different functions to be performed on a data set. Some query systems are more tailored to specific sets of data and end-user requirements than others. The appropriate language and data model will depend on the specific user requirements. These requirements – the users' questions – may be dependent on the scope of the domain. We can describe two classes of query languages: general purpose query languages, such as SQL and SPARQL, and those that are designed to meet the needs of a specific domain. Examples of the latter are query languages for linguistic annotations, such as LPath+ and EmuQL.

Query languages that are focused on specific domains are optimised for questions in that domain in comparison to the generic approach of general purpose query languages. This makes domain-specific query languages a better option for the user when asking questions in a relevant domain [4]. This is because the domain-specific query languages incorporate concepts unique to their domain, which would otherwise be difficult or cumbersome to ask in a general purpose query language. But what is the best way of creating such a query language or query system? Why – if given a standard set of questions for a domain – do the query languages in the same domain vary so widely in how they ask the questions? Are some more expressive than others in this regard, or are they all just different surface forms of the same abstract query language?

The goal of this project is to analyse and compare two query languages specific to the domain of linguistic annotations as examples of domain-specific query languages. To examine their structure and design, as well as their levels of expressiveness in comparison to each other and relative to the semantics of the questions that are being asked. Do the differences between the languages affect their ability to answer the domain-specific questions?

As part of this study, versions of each language will be implemented to work with a unified data model, and so the questions asked of each will be the same. This will allow the queries in both languages to then be converted to the general purpose query language SPARQL, and run on a data set to prove that the implementation of the languages is functional. The design of the underlying implementation will also be taken into account. Conclusions are drawn regarding their design and expressiveness, the implications regarding the similarities and differences between the languages, as well as areas for future work.

This study introduces the area and offers some preliminary work on the subject, paving the way for possible further studies on a broader analysis of domain-specific query languages, their optimal design and construction.

Chapter 2 will cover a review of the literature and background material. Chapter 3 will explain some topics in domain-specific query languages with a particular focus on linguistic annotations, along with existing work in the area and common questions in the field of linguistic annotations that a user may want to ask. Whilst this won't cover all possible questions in the field, it will cover a common subset, as well as some of the more complex queries in the field. Chapter 4 describes the case study carried out as part of this study, from the implementation to the results. Chapter 5 discusses the results of the case study presented in chapter 4, and the overall results of the study. Chapter 6 closes up this study.

2 Literature Review

2.1 Introduction

Language is used to express concepts. To convey these ideas to others, a language needs to be understood by others. It needs to be expressive. This is true not only for natural languages, but also for programming languages. The expressiveness of a language is defined by its grammar and semantics, which may be referred to overall as a language's rules. More complex rules allow for more complex languages and increased levels of expressiveness. In contrast, simpler languages are defined by simpler grammars.

This chapter provides some background on the expressiveness of languages and language grammars. It covers the usefulness of domain-specific languages, with a focus on query languages and the overall query system. This provides the background for further discussion on domain-specific query languages in chapter 3.

2.2 Expressiveness of Language

The expressiveness of a language grammar defines what a language is able to say. The level of expressiveness can be determined from the Chomsky hierarchy [5], shown in figure 2.1. This hierarchy formally classifies grammars, from which their complexity and expressiveness can be determined. The Chomsky hierarchy has four levels of increasing expressiveness with which to describe language grammars, starting from the inner-most level. This increase in expressiveness and generative power of a language, represented by the varying levels of the hierarchy, can be achieved by gradually removing constraints on how the grammar rules are allowed to be written, thus allowing for greater expressiveness [5].

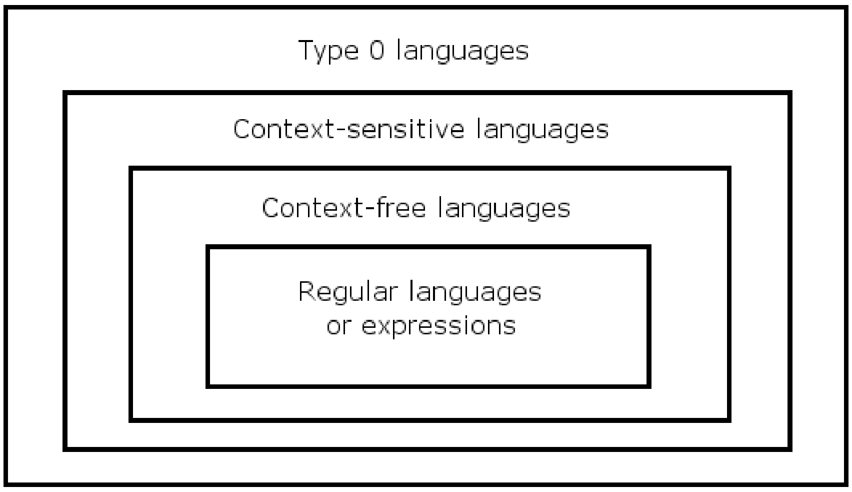

FIGURE 2.1: Venn diagram of the languages on the Chomsky hierarchy [5].

The levels of the Chomsky hierarchy from the inner-most restricted layer to the outer-most expressive layer are: regular language, context-free grammar, context-sensitive grammar and at the most expressive level is type 0, otherwise unknown as an unrestricted grammar [5]. Regular language grammars are equivalent to regular expressions and allow any single non-terminal to be rewritten as a terminal or a terminal followed by a non-terminal. Terminals refer to the words in the language, whereas non-terminals refer to the symbols that represent clusters of terminals and non-terminals. Together, they form the rules of the lexicon. Context-free rules allow any single non-terminal to be rewritten as any string of terminals and non-terminals. Context-sensitive grammars have rewrite rules where non-terminals can be rewritten as non-terminals surrounded by terminals. Finally, unrestricted grammars have no restrictions on their form other than the left-hand side cannot be an empty string.

While it may seem ideal for languages to be as expressive as possible, there is a benefit to restricting a language's expressiveness. A language's expressiveness is tied to its complexity. A complex language takes longer to compute solutions, whereas a simpler language is much easier to both understand as well as process [5]. This is due to the rules defining a language. Less complex rules result in fewer variations in the language. Regular expressions, for example, are restricted in what they can express, but are very easily processed. The grammar of a language like English, on the other hand, while expressive, would be much more difficult for a machine to process and understand [5].

2.3 Domain-Specific Languages

Domain-specific languages (DSLs) are computer programming languages of limited expressiveness focused on a particular domain [4][6]. DSLs trade the general expressiveness of general purpose languages (GPL) for more specialised expressiveness within a limited domain. A benefit of restricting the domain is that while it maintains or increases the expressiveness of a language within a domain, it does so without the increased complexity of an unrestricted language grammar [7]. While the general approach of GPLs are able to provide general solutions, they are suboptimal and not targeted. Restricting the domain to a smaller set of problems allows for a more specific solution [4]. While DSLs may fall into any of the levels described in figure 2.1, in many cases they are within the more restricted layers as their grammar is more restricted than that of GPLs. GPLs would tend to be type 0.

DSLs can be designed and implemented from scratch, or they can be implemented by extending a given base language [4], where the base language may be a GPL. It is encouraged that DSLs are based on existing languages [7]. By being based on an existing language, it brings the added benefit of reducing the training time and cost involved in learning an entirely new language, which may otherwise have a non-standard and unintuitive language design. This increases the likelihood of uptake for a new DSL. By specialising in a domain, they are optimised to better express desired concepts in specific domains in comparison to the base language. Convoluted approaches are less likely, as the language would have been designed with the domain in mind, thus enabling the intent of the program to be more clearly communicated [6]. Implementing a DSL through the use of an existing language allows the compiler to be left as is without the need for further adjustments. In this sense, the DSL can be considered as a library of the base language i.e. it implements additional functionality. This allows for a domain-specific solution without losing features of the base language due to restrictions [4]. The problem with this approach though is that the expressiveness of the DSL will be limited by the syntactic mechanisms of the base language. A DSL may need to express concepts which are not possible in the base language. Consequently, the optimal notation of the DSL would need to be compromised to fit the limitations of the base language [4].

There are a wide range of DSLs for different purposes. Many GPLs also have additional libraries or packages that can be imported to increase the functionality of the language, like a small-scale DSL within the GPL. The functionality that they add varies, but they will be for specific purposes, whether it be to handle text parsing (such as Python's Pyparsing[1]) or to add an easier-to-use API for the existing language (such as JavaScript's jQuery[2]). These allow a programmer to use pre-written modular code, so as to not need to re-write existing but specialised functionality, or learn a new language altogether. These libraries behave very much like domain-specific languages, and can be considered as such. To differentiate between them, the libraries are referred to as internal DSLs and the larger scale DSLs are referred to as external DSLs, due to them being removed from the base language [6]. Different types of DSLs include GPL libraries, markup languages and query languages. Some common examples of DSLs include SQL, HTML, MATLAB and Microsoft Excel's macro language [7]. It should be noted though that it is not always clear whether a language is defined as a GPL or as a DSL. The reason for this is that the level of domain specialisation can be considered as a scale [7] from general domain focus to a very specific focus point within the domain.

[1]`http://pyparsing.wikispaces.com/`
[2]`http://jquery.com/`

2.4 Query Systems

A query language is a common type of DSL. Query languages were designed to focus on the domain of databases, with their scope limited to accessing, modifying and querying databases [1]. Query languages are a component of a larger query system. Query systems consist of a database and a query language which is executed by a query processor. A query language may also have an associated query algebra that provides some insight into the formal properties of the language. Query systems are designed to provide a user with a way of effectively managing and accessing a data store. There are differing types of query systems. For the purpose of explaining the components of a query system, the relational query system is used as an example to illustrate the parts.

2.4.1 Data storage

A database is a collection of persistent data which may model the real world, and in many cases is also a shared resource [1]. In a relational database, this data is stored as relations. The relations are inter-connected data items. For example, a database may model a library system, where the data items are the books in the catalogue. Each data item entry may have a series of attributes about the item in question. Using the same example, each book would have a title, an author, a call number, etc. Relations are often called flat relations due to their flat tabular form, as the values cannot be nested i.e. an attribute value for a data entry cannot be a set of values. In the library example, this would mean the author attribute could not list more than one author. To implement the sort of nested hierarchy required for listing multiple authors, a nested relational model would be required. A nested relational model uses keys to handle the one-to-many mappings.

A computer system responsible for the storage and retrieval of data items from a database is called a Database Management System (DBMS). A DBMS should provide a Data Definition Language (DDL) for defining schemas, a Data Manipulation Language (DML) for querying, accessing and updating the data, as well as maintain integrity and consistency [1]. Each DBMS supports a particular data model, where the standard is the relational data model.

A relational data model is a model consisting of three parts; structural, integrity and manipulative. The structural part consists of schemas, which outline the structure of the database. The integrity part maintains consistency between the varying tables within the database, through the use of primary and foreign keys. The keys function as unique identifiers and attributes in the table entities, and allow for cross-referencing between tables. This enables a model to implement the nesting of values. The manipulative part allows for the relational algebra and calculus to perform operations on the relations in the database [1].

Other types of data models include generic data models such as XML [8] and RDF [9], as well as domain-specific models such as the annotation model. Unlike the relational data model which stores data in tables, XML and RDF store their data in trees and graphs respectively. This allows the data model to be traversed by the query engine. These data models are less rigid in their structure in comparison to the relational data model and are suitable for representing hierarchical data. The annotation data model is a data model specific to the domain of linguistic annotations. The structures in the data model correspond to elements in the data and allow a user to describe language data. These data models are further explored in chapter 3.

2.4.2 Query Languages

A query language is a domain-specific language and can be a data manipulation language. In addition to querying data in a database, it may provide a means for the user to add new data, as well as modify and delete existing data from a database [1]. As with domain-specific languages, there are a wide range of query languages, for varying databases and query systems. Some examples of query languages are SQL, XPath, XQuery, LPath+, EmuQL and SPARQL.

SQL is a general purpose query language based on relational calculus [1], as well as a DSL [7]. It is also the standard query language used across the industry for relational DBMSs. Originally named SEQUEL (Structured English Query Language) [2], SQL is short for Structured Query Language [1]. It is designed to be reasonably high-level, so as to be relatively readable and usable to users who are willing to learn, but may not be computer specialists. It relies on a relational database, as this is the simplest possible general purpose data model. This in turn allows for the maximum degree of data independence [2].

The structure of the data stored in an SQL database is defined by a schema. The schema is used to validate the data, as the data needs to be complete and properly structured in order for it to be queried. Basic SQL queries consist of multiple parts: a table name, domain, range and argument [2]. The table name defines which table in the database should be searched, the domain and argument are used as filtering conditions to narrow the search, and the range is the value being searched for. This is shown in figure 2.2. Given this structure for a basic SQL query, the user only needs to be able to specify which columns they are searching, from which table they are searching, as well as the conditions they need the results to meet, for them to be able to run a query on the database. This structure is a basic component of the language [2].

```
SELECT id, title
FROM annotation_sets
WHERE {
    type = ‘word’
}
```

Figure 2.2: SQL: Select all entries from the annotation_sets (table name) table, where the type (domain) is “word” (argument). Of the entries selected, return the ID and title (range) of each entry.

2.4.3 Query Algebra

The relational data model has only one data structure – the relation. Relations represent entities and their properties are described by a relation schema. Relational databases are sets of relations and a database schema is a set of relation schemas [1]. These relations can be manipulated, queried and updated using relational algebra and relational calculus.

Relational algebra – a procedural query language for the relational data model – is a collection of operators. It is a procedural query language in that it specifies the steps in the procedure to be executed, as opposed to a declarative query language which would specify the goal at a higher level. In relational algebra, each operator takes a relation or pair of relations, and the returned output of a query would be a single relation. For example, calculating the union of two relations. It is common to translate a query language into an algebra, as it provides the semantics for the query language, making it clear what operations are possible. In addition to this, an algebra supports query optimisation [10]. Relational calculus, on the other hand, is a declarative query language and is the theoretical basis of SQL [1]. Based on first-order predicate calculus, it is more suitable as a basis for user-orientated query languages due to being closer to natural language.

2.5 RDF Query System

Query systems are not restricted to the relational model. Another option is to use RDF's graph structure as a basis for a query system. RDF, short for Resource Description Framework [9], is a semi-structured format that stores data as a collection of triples to create a directed edge labelled graph. It is often used to represent metadata about Web resources. Each triple represents an entry in the data store, consisting of a pair of nodes and the relation linking them to each other. The nodes are referred to as the subject and the object, whilst the linking relation is the predicate[3]. RDF triples describe resources, where the object node represents the resource being described. It is represented by a URI linked by a predicate to a subject node; the predicate describes the relation between the two nodes, and the subject node may be represented by either a URI or by a literal string. Multiple triples can be linked together to form a graph, as the object node of one triple may be the subject node of another triple. Consequently, an RDF graph would be closer to the design of a nested relational model than that of a flat relational model. An example of an RDF graph is shown in figures 2.3 and 2.4.

There are a variety of query languages that a user may choose from to query an RDF data model, including SPARQL [11] and RDF-GL [12]. SPARQL [11], a recursive acronym for SPARQL Protocol and RDF Query Language, is the standard query language for RDF graph data [13]. Basic SPARQL queries look similar to SQL queries and contain a basic graph pattern. A basic graph pattern is a set of triple patterns representing RDF triples, where some elements of the triples may be variables. This basic graph pattern is used as the conditional on which the query finds matching subgraphs [11]. The query specifies the various requirements of the search in separate sections. An example of a SPARQL query is shown in figure 2.5.

RDF-GL[12] is another RDF query language. It takes a unique approach in regards to standard query formats, as the queries are formed graphically rather than via text queries,

[3]Predicates in RDF refer to relations whereas predicates in path-based query languages refer to peripheral conditional filters.

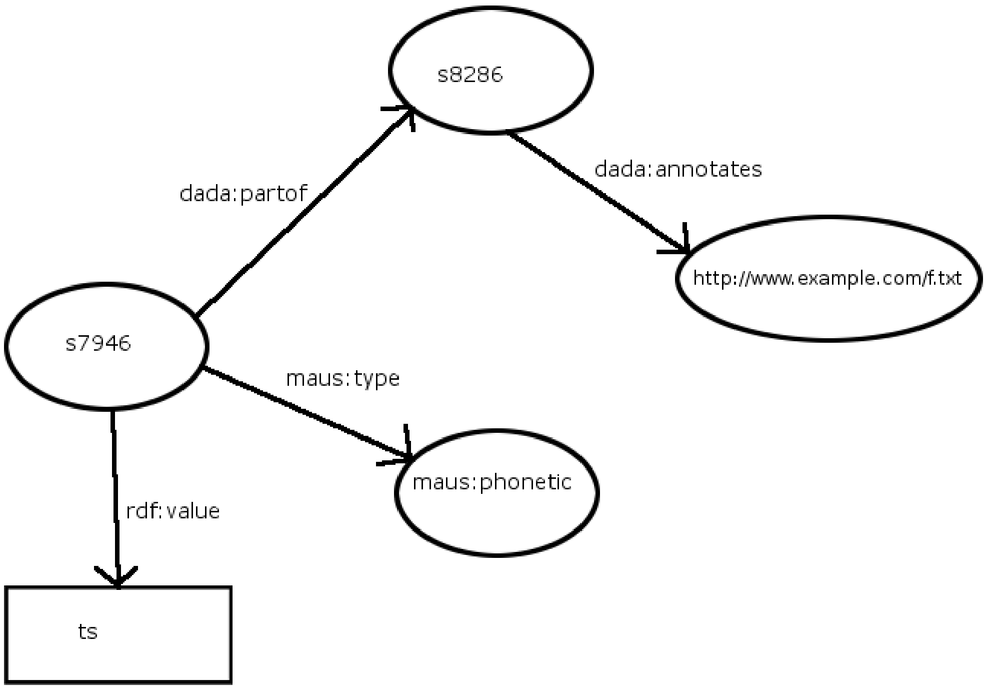

Figure 2.3: RDF graph showing nodes and relations.

```
:s8286 rdf:type dada:annotationset;
    dada:annotates <http://www.example.com/f.txt> .

:s7946 rdf:type dada:annotation;
    dada:partof :s8286;
    maus:type maus:phonetic;
    rdf:value "ts" .
```

Figure 2.4: Turtle / N-triple notation of RDF triples in figure 2.3

```
PREFIX ns1: <http://www.someaddress.com#>

SELECT ?id, ?title
WHERE {
    ?annotation ns1:id ?id.
    ?annotation ns1:title ?title.
    ?annotation ns1:child ?child.
    ?child ns1:type ‘word’.
}
```

Figure 2.5: SPARQL: Select all annotations, where the child’s type is “word”. Of the annotations selected, return the ID and title of the annotation.

though the back-end is still reliant on SPARQL. Whilst this does make it easier to use as the user does not need to learn the finer points of the language, it is also less expressive and restricts what can be asked due to its limited ability in interpreting different visualisations.

2.6 Conclusion

Language is an important tool used for communication, and query languages – and by extension, query systems – are important for use in querying data stores. As shown using the relational query system, a query system may consist of a data model e.g. the relational database, a query language with which to access and query the data model e.g. SQL, and possibly a query algebra. Query systems rely on these components to provide functionality.

While these components form the basis of a query system, a query system is not restricted to just those shown in the relational query system example. There exists a wider range of options available to users. For example, the data model may be a relational database, or it may be in RDF, XML or even a custom designed data model to suit the data being represented, such as annotations. Likewise, the algebra and query language also do not need to be relational.

While there are a range of different data models, query languages and query systems to choose from for a given data set, there are times when a user may want a new query system created for their data – a query system tailored to their specific domain. A domain-specific query system may make use of pre-existing data models and query languages, or it may use its own data model and language. Which is appropriate is often dependent on the data, domain and user. The ability to mix-and-match, as well as create new components, is a benefit of having the separate modular components. New versions could potentially be developed for different purposes. New query languages could be developed for existing data models, or existing languages could be adapted for use with new data models.

There exists a lot of data in the world, and whilst not all of it may be available to query systems, there have been efforts to include as much as possible. In addition to that, big data and linked data have also been gaining popularity as part of the larger the move away from completely structured database tables and towards semi-structured databases as a distributed global resource [14]. This results in the added importance of the need for an effective query system.

3
A Study in the Domain

3.1 Introduction

There are a wide variety of queries that a user may want to be able to ask a database. In selecting the appropriate query system, a user is presented with a wide range of choices regarding query language, data model and database format. Limiting the scope to a specific domain allows for a more focused approach in regards to data querying and query systems as a whole. Query systems and languages can be made more effective for a given data model if they are domain-specific [7].

Domain-specific query languages restrict the user to asking specific types of queries relevant to the domain they are designed for [6]. While this limits the overall possible expressiveness of a query language, it has the benefit of being specially designed for features in the target domain that more general languages may not adequately handle. For example, the data may be a graph of child relations and a user may want to query the ancestor concept. A DSL may optimally implement the ancestor concept without the relation being explicit in the data i.e. the relation may be inferred. A GPL on the other hand would not have this concept and the user would need to traverse the child relations. The ancestor concept in the DSL would be built into the language, thus possibly reducing overhead and repetitiveness in the query. This simplifying of the query increases the usage, effectiveness and possibly readability of a possible query language. In addition to this, as the system no longer needs to handle queries irrelevant to the domain, it is possible to optimise the system based on the domain and the range of possible domain questions [4].

This study will focus on the domain of linguistic annotations, as a case study to explain this concept. Linguistic annotations are the metadata about existing language data. Language data includes text, audio and video data, such as recordings of spoken data or collections of written text. This language data is often stored in corpora. Corpora are large sets of linguistic data, and are usually structured and representative of a larger set e.g. a corpus may be representative of the language used by the population. Corpora may contain text, audio, imagery and metadata [15].

Linguistic annotations are annotations on this language data. These annotations allow linguists to describe structures in the language data, as well as any other additional relevant data. Linguistic annotations may include data regarding timing, word structures, pitch, meaning, part-of-speech tagging, transcription of speech, intonation and gestures [16]. The annotations may be embedded in the language data, or they may be stored separately from the language data and refer back to the original data. They may be stored alongside the language data in the same corpus. Linguistic annotations are hierarchical, and may be represented as trees or graphs, as shown in figure 3.1.

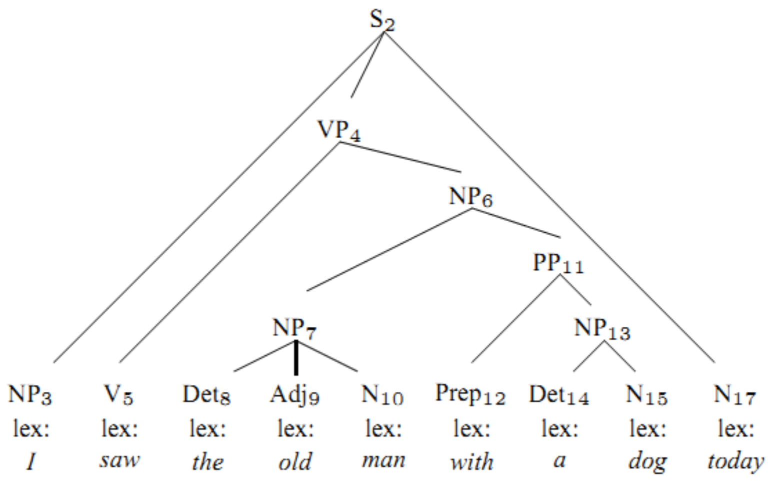

Figure 3.1: Linguistic data represented as a tree of part-of-speech tags for the sentence "I saw the old man with a dog today" [17].

Figure 3.1 shows a linguistic tree consisting of annotations describing the sentence "I saw the old man with a dog today". This sentence is broken down into part-of-speech tags e.g. verb phrases and noun phrases. The root node represents the entire sentence and is denoted by S_2. This is broken down into the noun phrase NP_3, verb phrase VP_4 and noun N_{17}. NP_3 represents the label "I", VP_4 represents "saw the old man with a dog" and N_{17} represents "today". The entire sentence is broken down in this way. Each part-of-speech tag is an annotation node. At the end of the tree, the leaf nodes also contain lexical attributes describing the relevant word in the sentence. This linguistic tree of annotations consists of one annotation type – part-of-speech, though some of the nodes may contain additional lexical attributes. Linguistic trees such as this are used by LPath+.

Additionally, due to the nature of language data, each piece of data may have multiple types of annotations linked to it, as shown in figure 3.2. This figure describes the spoken sentence "he does a lot of environmental impact stuff". The annotations on this sentence describe the time alignment on each word, tonal changes and pauses in speech. These are represented as red, yellow and green markings respectively. The data also includes the text transcription of the audio recording, as well as the audio signals. These annotations are multi-modal, as they include multiple types of annotations. These varying types of annotations may also form hierarchical tiers e.g. orthography annotations may dominate phonetic annotations. Multi-modal linguistic annotations are used by the Emu query system.

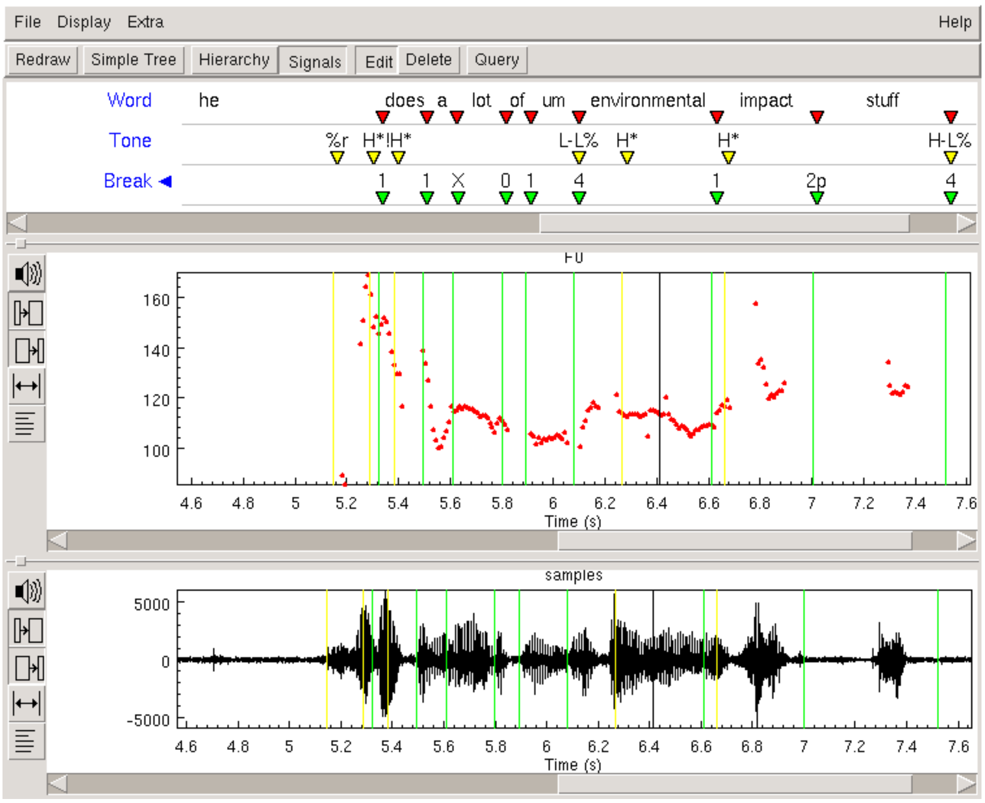

Figure 3.2: Tiered Emu speech data, showing word, tone, break and timing tiers for the spoken sentence “he does a lot of environmental impact stuff”.

Linguistic annotations are a suitable domain to use for this case study as they contain concepts unique to the domain that are not normally present in generic graph data. For example, the concepts of dominance and immediately following are specific to this domain and commonly used [17][18]. Additionally, it is suitable for representing as graph data due to its hierarchical nature. Graphs are a flexible way in which to represent data due to it enforcing little restriction on the data itself, and have long been used to describe linguistic annotations [19].

This chapter will cover some examples of query systems suited for linguistic annotations, including both the data models as well as the query languages. The different query systems each have their separate strengths and weaknesses, and the ones chosen for a particular task will often come down to the task required as well as the data being queried. This chapter will cover XML, RDF, the domain-specific annotation data model, as well as touch on a method for evaluation. As part of the focus on linguistic annotations, this study will focus on the domain-specific query languages EmuQL and LPath+, as well as the current levels of expressiveness of both languages. These two languages were chosen as examples of domain-specific query languages in the domain of linguistic annotation, and a subset of each was implemented for the case study covered by chapter 4.

3.2 Representing Annotations as XML

It is very common for linguistic annotations and text corpora to be represented as XML [15]. XML, short for Extensible Markup Language [8], is a markup language for text. The data is stored in a tree-based text format. XML provides a built-in mechanism to impose constraints on the storage layout and logical structure of the data. This ensures that the document is well-formed in accordance to the XML specifications. XML is suitable for representing linguistic annotations as its tree-based structure is similar to the hierarchy present in the data. For example, XML documents consist of a root node with possibly a series of child nodes. The child nodes themselves can have child nodes. All the nodes in an XML document may have text values, as well as additional attributes describing the nodes. This structure forms a tree. This is similar to the structure of linguistic annotations where larger structures such as a document may be the root node. The document may contain words which are child nodes, and those nodes may contain phonemes as child nodes. This could all be described in an XML document.

There are several ways to represent annotations in XML, including using embedded notation and stand-off notation [19]. The embedded format includes the annotations in the same document as the textual language data. The annotations mark up the data. For example, an audio transcription could include annotations attributing speaker turns and timestamps. The stand-off notation places the annotations in a separate document. This would leave the original document unmodified i.e. rather than marking up the original text with annotations describing words, phonemes, speaker turns, etc., the XML document would contain reference points to the text document and the annotations would be stored separately. Using stand-off notation allows for multiple types and styles of annotations, whilst keeping the original document unmodified and readable.

There are a variety of query languages designed for use with XML, such as XQuery [20] and XPath [21], as well as XSLT [22] which can be used for transforming XML documents. XQuery is an XML query language that is generalised and able to query different types of XML data sources, including both databases and documents. Like SQL, XQuery is designed to be easily understood [20]. An XQuery expression may consist of variable declarations, conditional "WHERE" clauses, as well as the format in which the resulting data should be returned. An example of XML data and an XQuery expression are shown in figures 3.3 and 3.4, where the query selects all annotations of type "word" which are a child of annotation_set, which in turn is a child of group.

XPath [21] is an XML navigational language [23] and is designed for tree-based XML data. It is also a subset of XQuery. XPath queries take the form of a path expression, very much like the address of a webpage or URI. In addition to the main path, which specifies the end target of the query, required conditionals can also be specified with use of predicates. These can be joined to form more complicated queries, along with the use of functions. XPath queries focus largely on locating nodes in the tree, whereas XQuery expressions allow for additional constraints and actions. An example of XPath with the use of a predicate can seen as follows in figure 3.5, where the query selects all annotations of type "word" which are a child of both group and annotation_set.

```
<xml>
    <group>
        <annotation_set></annotation_set>
        <annotation_set>
            <annotation>
               <type>word</type>
               <text>cat</text>
            </annotation>
            <annotation>
               <type>word</type>
               <text>dog</text>
            </annotation>
            <annotation>
               <type>phoneme</type>
               <text>d</text>
            </annotation>
        </annotation_set>
    </group>
</xml>
```

Figure 3.3: XML containing annotations

```
for $a in //group/annotation_set/annotation
where $a/type = ‘‘word’’
return $a
```

Figure 3.4: XQuery: Select all annotations which are a child of annotation_set, which in turn is a child of group. The selected annotations should be of type "word".

```
//group/annotation_set/annotation[type=‘word’]
```

Figure 3.5: XPath: Select all annotations which are a child of annotation_set, which in turn is a child of group. The selected annotations should be of type "word".

Though XPath and XQuery are suitable for querying XML, which can be used to represent linguistic annotations, they do have weaknesses regarding their expressiveness in the domain. XPath and XQuery are unable to adequately express queries requiring sequential constraints and embedded clauses [24]. Functional implementations are possible in certain queries, though highly complex [15]. They require the use of user-defined functions, which would prevent query optimisation by the query engine. These queries often rely on recursion and repetition in language. A possible solution to this problem would be to extend the query languages for annotation queries.

3.3 Representing Annotations as RDF

As an alternative to XML's tree-based structure, linguistic annotations can also be stored in RDF's graph structure. While RDF is intended for use in describing Web resources and metadata, it is possible to use it to represent annotations,. This is because RDF is a graph, and so the model is a close match for the annotation data model. Examples of annotation data represented as RDF can be found on Alveo at `http://alveo.edu.au/`. Alveo [25] is a project connecting tools and large collections of corpora for research. Some of the data used for this study was taken from Alveo.

RDF is not a perfect match for the annotation data model though, as the annotation data model contains functionality and elements specific to linguistic annotations. For example, while both RDF and language data may be represented as a directed graph, the order of the nodes in RDF is not defined by the model like in the XML data model. This detail is important in language data. Consequently, it would need to be explicitly defined in RDF, whereas it may be implicitly defined in XML. In linguistic annotation data, the ordering may be used to convey information about dominance or other relations such as following.

3.4 Annotation Data Model

XML and RDF are not the only options for representing linguistic annotations. Another option would be to create a new abstract data model specific to annotations i.e. the annotation data model. The design for this data model can be based off pre-existing data models such as RDF, but include modifications and features specific to linguistic annotations. Alternatively, it may be designed entirely from scratch and specific to a query system. For example, LPath+ uses a relational database to store linguistic annotation graphs [3], whereas the Emu query system uses its own data model [26].

The ISO standard for representing the annotation data model is LAF, and there has been some work in the area based on LAF, such as GrAF [19] as well as an RDF realisation of LAF [27]. LAF, short for Linguistic Annotation Framework, provides a general framework for representing linguistic annotations [19] and GrAF is a graph-based format for representing linguistic annotations, as an extension of LAF.

As the annotation data model is a domain-specific data model, the query languages used to query this data model would ideally also be domain-specific. By using a suitable query language, the user would be able to apply, as well as query, concepts that are unique to this domain. Domain-specific query languages for linguistic annotations include LPath+ and EmuQL.

3.4.1 LPath+

Like SQL and SPARQL, LPath+ [3] is a query language and a domain-specific language. Unlike SQL and SPARQL, it is also a domain-specific query language. It is based on XPath and has been formally designed with a defined grammar. Like XPath, LPath+ is a path-based query language. A path-based query language is one where the query denotes the path through the data that needs to be traversed, in order to find the resulting nodes. For example, consider the query:

```
//S/NP/N
```

The query finds S nodes, then follows the child nodes labelled NP and continues on to subsequent child N nodes. The query would return the N nodes. These result nodes were reached from following the path specified through the data.

LPath+ is specific to the domain of linguistic annotations and has been designed for linguistic trees, such as shown in figure 3.1. Linguistic trees can be considered to be a special subset of graphs, in that they have a singular root node and the graph is non-circular and directed. LPath+ has the benefits of being both domain-specific and being based on an existing commonly used and successful language i.e. XPath.

As a path-based query language, LPath+ is tree-based, and so will not natively handle the circular nature of graphs in the query itself. Whilst adapting a path-based query language to handle graphs is possible, such as with GXPath [23] or via generalising XPath [15], it does take a bit of work and isn't always as intuitive as the base language. Generally, the preference is for an intuitive, simple and optimised solution as opposed to a possibly convoluted workaround. Consequently it would be preferable to have a domain-specific query language rather than a general purpose query language when a clear domain is given. LPath+ implements several concepts important to linguistics and linguistic annotations, such as immediately following and ancestor relations, which may not always be present in the data. The ability to infer and query concepts not present in the data is a benefit of a domain-specific query language. An example query in LPath+ is shown as follows, where it selects a node labelled VP which immediately follows a NP, with an ancestor node labelled S.

```
//S//NP-->VP
```

Other relations handled by LPath+'s relation axes include parent, child and sibling. LPath+'s axes are shown in figure 3.6. LPath+ is not just an implementation of XPath on linguistic trees. It extends XPath in a number of ways, such as scoping [17]. Scoping allows a user to restrict part of a query to a subset of the graph. To perform a similar query in XPath, two separate queries would be required i.e. the first to obtain the subset and the second to perform the actual query.

3.4.2 EmuQL

The Emu Query Language [26] (EmuQL) is a domain-specific query language for multi-level linguistic annotations and hierarchical annotations on speech data. Designed specifically for querying speech corpora, it handles concepts such as dominance and followed-by, which can be thought of as a basic implementation of time-alignment, whilst also remaining simple enough to be used by non-programmers. An example of an EmuQL query is shown in figure 3.7.

EmuQL was designed for use within the Emu query system. It was designed informally, and as such, whilst there exists documentation on the language specifications [28] and rules,

`\`	parent	`/`	child
`\\`	ancestor	`//`	descendant
`\\*`	ancestor or self	`//*`	descendant or self
`->`	immediate following	`<-`	immediate preceding
`-->`	following	`<--`	preceding
`=>`	immediate following sibling	`<=`	immediate preceding sibling
`==>`	following sibling	`<==`	preceding sibling
`.`	self		

Figure 3.6: LPath+ axes [17].

```
[[Word=C ^ Orth=T]->#POS=NP]
```

Figure 3.7: EmuQL: Select annotations where the value of the POS tier is NP, which follow annotations with an Orth value of T, dominated by an annotation where the Word value is C.

there are no clear grammar guidelines on the language. The documentation is explained using examples. Simple queries in the language involve searching for a single token and restricting either its label or its position relative to its sibling tokens [26]. More complex queries are formed from various combinations of simple queries. The basic properties of the language involve searching for matching annotation tokens on the appropriate tiers [29].

```
maus:orthography='time'

maus:phonetic='Ae'

[maus:orthography='time'^maus:phonetic='Ae']
```

Figure 3.8: Simple EmuQL queries which combine to form a more complex EmuQL query.

EmuQL queries are structured like trees, in that the leaves are simpler queries or nodes joined by connectors. For example, consider the queries in figure 3.8. In the first query, the left leaf specifies that the type of node being queried for is orthography, whereas the right leaf specifies that the label of the node in question is "time". These leaves are joined with the "=" connector, to indicate reference to the same node. In the second query, the structure from the first query is repeated. The second query searches for a phonetic node labelled "Ae". The third query is a complex query that connects the first two queries. It is in this example that the tree structure is more evident. Like the simple queries, it consists of two leaves and a connector, but in this case the leaves are the first two queries and the connector is the dominance relation, represented by "^". The dominance relation in EmuQL is bi-directional i.e. the left leaf may dominate the right leaf, or the right leaf may dominate the left leaf. Which leaf is the dominating leaf will depend on the tier of the nodes in question. The dominating tier would be defined by Emu's schema in its database template file [26], separate from the data. In this particular instance, the orthography node would

`=`	equals	`Num()`	function for a number of annotations
`!=`	not equals	`Start()`	function for initial position
`|`	or	`End()`	function for final position
`&`	and	`Medial()`	function for medial position
`->`	sequence		
`^`	dominance		
`#`	return only this annotation		

Figure 3.9: EmuQL features and relations [29].

dominate the phonetic node.

EmuQL is able to ask a wide variety of queries relevant to the domain, as described in section 3.5. It has handling for "equals", "not equals", "dominance", "sequence", "or" as well as "and". It also supports the nesting of queries, and so more complicated queries can be created from a combination of simpler queries. In addition to this, EmuQL also has some level of function handling. The features and relations supported by EmuQL are shown in figure 3.9. While EmuQL does not have handling for wildcard usage, it is possible to express this concept in EmuQL if the user has prior knowledge of the data structure, as explained in section 3.5.5. This is considered to be a workaround rather than a design decision as part of the language and is a deficiency in the language. In the examples found [29], queries using the wildcard concept are structured to use negation and a real value. For example, to find all Orth nodes, the query could be written to find all Orth nodes that do not match "x". If "x" is not present in the data, this should match all Orth nodes. This can be seen in the following example:

```
Orth!=x
```

This usage of the language syntax relies on fore-knowledge of the data. This same structure for EmuQL is retained for this study, with the "x" replaced with "_" as the latter is less likely to be present in the data. Additionally, this would make its usage consistent with the wildcard usage in LPath+. This workaround usage in EmuQL may be considered as a weakness in the language design.

3.4.3 Design Analysis

The design and structure of an LPath+ query is different from an EmuQL query. LPath+ was designed in a much more formal and controlled way than EmuQL. This can be seen in the grammar definition of LPath+ [17] versus EmuQL's grammar which is defined through examples [26]. Despite the differences between the two languages, each are effective in their intended purpose.

While LPath+ has a defined grammar, it is not an entirely complete one [17]. Many LPath+ features are inherited from XPath, and as such, are not described in the specifications even though they are handled by the language, e.g. the use of "@" as shorthand for an attribute test, or function handling [17][21]. In addition to that, the LPath+ query language was designed for a data model with a single node type, unlike the multi-type data used for this study. This has resulted in assumptions being made by the language in regards to the data i.e. it is assumed that the data is a tree of labels of the same type, as shown in figure 3.1, which do not need to be specified. While original LPath+ queries would still be valid on the test data set due to the different labels being used for the different node types, this creates complications when a wildcard character is used in place of a label, as the labels are no longer relevant in this case. To solve this, attribute tests are used to specify the node types, as they are already part of the language through the use of the "@" character.

LPath+ differs from EmuQL in query structure. Whilst both are intended for linguistic annotation data, LPath+ is structured as a path-based navigational language. Tree-wise, it would be similar to the right-leaning tree, rather than Emu's balanced tree. Figure 3.10 shows the queries from figure 3.8 written in LPath+, in both the original intended LPath+ as well as the version used in this study with the attribute tests. The first query in each case shows the original intended LPath+ whilst the second shows the version used in this study. It can be noted that whilst EmuQL and LPath+ queries are quite different in structure, their expressiveness in regards to the sorts of queries they can ask are similar. This implies that the difference between the two languages is merely at a surface form level.

```
//time
//time[@dada:type=maus:orthography]

//Ae
//Ae[@dada:type=maus:phonetic]

//time/Ae
//time[@dada:type=maus:orthography]/Ae[@dada:type=maus:phonetic]
```

Figure 3.10: The first query is a simple LPath+ query finding an orthography node labelled "time". The second query repeats this concept for a phonetic node labelled "Ae". The third query is a more slightly complex query that finds the node labelled "Ae" that is dominated by the node labelled "time".

While it is possible to mix-and-match data models and query languages, some languages are more suited to specific data models. The levels of expressiveness presented by each language varies, as does their suitability for varying tasks. Despite these variations, it is possible for languages to have similar levels of expressiveness. This study boils down both EmuQL and LPath+ into a similar abstract language model. This model highlights the possible range of both languages and their focus on the domain questions, as well as shows that while the query structures appear different, the differences are surface form and are likely syntactic choices by their developers, as the questions they are able to ask and answer are essentially the same.

3.5 Questions in the domain

As this study involves an implementation of query languages, it is important to understand the range of queries that a user interested in this domain may want to express. These queries, or even just a subset of these queries, may form part of a series of tests on which to evaluate the implementation. They also allow the analysis and comparison of the query languages, as while the questions are relevant to the domain, they are independent of the query languages. This allows for an evaluation of the expressiveness of the query languages. Some queries that a user interested in linguistic annotations may want to express follow. These queries are used as tests for this study.

3.5.1 Basic

This test is a query in its most basic form, searching for a single node. For example, find an orthography node labelled “time”. The point of having this query would be to act as a base case, to ensure that the query languages can express the simplest of questions. Both EmuQL and LPath+ are able to express this, as shown.

```
EmuQL:
maus:orthography=‘time’

LPath+:
//time[@dada:type=maus:orthography]
```

3.5.2 Sequence

The sequence query searches for two nodes that are next to each other. Given the data used, this is represented as a sequence i.e. X which is then followed by Y sequentially in time. An example of this query in both EmuQL and LPath+ is shown. The query finds a phonetic node labelled “t” followed by a phonetic node labelled “Ae”.

```
EmuQL:
[maus:phonetic=‘t’->maus:phonetic=‘Ae’]

LPath+:
//t[@dada:type=maus:phonetic]->Ae[@dada:type=maus:phonetic]
```

3.5.3 Dominance

The dominance query searches for a node that has a particular parent or child. The “dominance” term is specific to linguistic annotations, as the nodes involved belong to different classifying tiers, which may dominate each other. Consequently the nodes are considered to dominate each other. For example, orthography nodes are considered to dominate phonetic nodes. An example of this query in both EmuQL and LPath+ is shown as follows where an orthography node labelled “time” dominates a phonetic node labelled “Ae”.

```
EmuQL:
[maus:orthography='time'^maus:phonetic='Ae']

LPath+:
//time[@dada:type=maus:orthography]/Ae[@dada:type=maus:phonetic]
```

3.5.4 Transitive Closure

The transitive closure or Kleene Star, often represented by an asterisk, is a qualifier of zero or more specified relations. To denote one or more, a plus sign is often used. For example, "X parent+ Y" would mean X is linked to Y by one or more parent relations. In this example, the concept would be considered an ancestor relation. This is a difficult concept to implement in query languages, both general purpose as well as domain-specific. It allows the traversal of a graph or tree of unknown depth, provided that the relations match.

EmuQL is unable to handle transitive closures, though it is able to handle some transitive relations i.e. following. That said, EmuQL is unable to distinguish between a general following and immediate following. It is possible that EmuQL behaves as though following is a defined relation, as opposed to a series of immediate following relations inferred from the data.

LPath+ is also unable to handle transitive closures, though like EmuQL, it is able to handle some transitive relations. Unlike EmuQL, it is able to distinguish between immediate and non-immediate following relations. The grammar also includes handling for the ancestor and descendant concepts [17], which are common in linguistic annotations as well as in graph data. This is a feature inherited from XPath, on which LPath+ was built. An example of LPath+'s descendant relation is shown as follows, where the query finds a phonetic node labelled "Ae" that is a descendant of an orthography node labelled "time".

```
//time[@dada:type=maus:orthography]//Ae[@dada:type=maus:phonetic]
```

3.5.5 Wildcard

Unlike the previously mentioned test queries, the wildcard test is not a test on the relation between nodes, but rather on the node label. That is, whether a query language can handle an unspecified variable as part of the query. In this case, whether the node can have an undefined label. Some query languages handle this by using a variable. This is how SPARQL handles wildcards.

Other languages may require the specific use of a wildcard character, such as LPath+'s "_" or the "." in regular expressions. Whilst LPath+ has native handling for the wildcard character, EmuQL does not. It is possible to work around this lack of functionality in EmuQL by taking advantage of the structure and content of the data, as seen in the following example. The query implements the wildcard concept by finding nodes that do not match a label which the user is certain is not in the data set. In the example provided, the Emu query system would find a phonetic node dominated by an orthography node labelled "time", as long as the phonetic node did not match the label "_". As "_" is an invalid label for a phonetic node, the node test will return all other phonetic nodes in the data set, provided the rest of the query conditions matched.

```
EmuQL:
[maus:orthography='time'^maus:phonetic!='_']

LPath+:
//time[@dada:type=maus:orthography]/_[@dada:type=maus:phonetic]
```

The use of wildcards are interesting as they increase the number of possible queries expressible. A user may not always know exactly what they are looking for, or they may not always have all the details on what they are searching for. Or perhaps there are parts of the query that they do not care about e.g. they may want to find X which is followed by something. It does not matter what the latter value is, so as long as X is not the last value. A wildcard would allow the user to search for it anyway.

3.5.6 "Not" And "Or"

Depending on the query language in question, the concept of negation is either fairly straight forward, or a difficult one. Most, if not all, query languages work by allowing a user to specify what it is they are searching for. The concept of negation goes against that by returning everything except what the user specified. EmuQL and LPath+ have some degree of handling for this concept, as seen in the following example.

```
EmuQL:
[#maus:orthography='time'^maus:phonetic!='r']

LPath+:
//time[@dada:type=maus:phonetic][not /r[@dada:type=maus:phonetic]]
```

The difference in how each language handles negation can be seen in how the query is expressed. The EmuQL query finds an orthography node labelled "time" which dominates a phonetic node that is not labelled "r". The LPath+ query, on the other hand, finds an orthography node that does not dominate a phonetic node labelled "r". Note that the example of the LPath+ given here is based on our understanding of the grammar. Whilst there is mention of handling for this concept in the specifications, we did not come across examples, and so this is what we have interpreted the grammar to mean.

In a similar manner as negation, the "or" concept is a boolean concept that is more easily handled in some languages than others. Both EmuQL and LPath+ are able to perform "or" queries, as seen in the following example querying for an orthography node labelled "time" that dominates a phonetic node labelled either "t" or "Ae". Like in the previous case, the LPath+ example is our interpretation of the grammar as we did not come across examples.

```
EmuQL:
[#maus:orthography='time'^maus:phonetic='t'|'Ae']
```

```
LPath+:
//time[@dada:type=maus:phonetic][/t[@dada:type=maus:phonetic] or
        /Ae[@dada:type=maus:phonetic]]
```

3.5.7 Nested Queries

Nesting is included as a test to ensure that EmuQL and LPath+ are able to handle sufficient complexity in the domain. It allows the ability to stack multiple filter conditions which a user may want to express. For example, find W which is a child of X that is followed by Y which is a parent of Z. The following is an example of a nested query in EmuQL and LPath+ searching for a phonetic node which follows a phonetic node labelled "m", which in turn is dominated by an orthography node labelled "time".

```
EmuQL:
[[maus:phonetic='m'^maus:orthography='time']->#maus:phonetic!='_']

LPath+:
//time[@dada:type=maus:orthography]/m[@dada:type=maus:phonetic]
        ->_[@dada:type=maus:phonetic]
```

3.5.8 Finding Siblings

The sibling concept is a combination of the dominance and sequence queries. For nodes to be considered siblings, in addition to occurring sequentially i.e. following each other, they must also share the same parent. For example, an orthography node labelled "time" may have child nodes which are phonetic and labelled "t", "Ae" and "m". All these phonetic nodes would be siblings, and the pairs "t" and "Ae", as well as "Ae" and "m" would be immediate siblings. The non-immediate sibling concept would also require the use of a transitive closure, since it is a sibling concept of undefined depth, or breadth in this case. The following shows an example of this query in EmuQL and LPath+ in the case where an orthography node labelled "time" dominates phonetic nodes labelled "t" and "Ae", and the phonetic nodes are immediate siblings..

```
EmuQL:
[maus:orthography='time'^[maus:phonetic='t'->maus:phonetic='Ae']]

LPath+:
//time[@dada:type=maus:orthography]/t[@dada:type=maus:phonetic]
        =>Ae[@dada:type=maus:phonetic]
```

3.6 Conclusion

In the end, there's a wide variety of query systems and languages. While some are general purpose, like SQL, others are domain-specific like LPath+. Domain-specificity improves the effectiveness of a language against the data set for the target domain, allowing for greater expressiveness in the queries being asked. That given, there's still a wide range of query languages for a variety of domains. Some of these domain-specific query languages are more expressive than others in the scope of what they are capable of asking. This may often come down to the design of the language. Overall for a given domain, the query languages in that domain will target a relatively restricted group of questions. This may imply that the differences between domain-specific query languages in the same domain are merely syntactic.

While there is a general method through which many languages are created and tested, there is no standard method guiding the creation of domain-specific query languages in general. How they are designed often depends on the data model. So then, what is the best way of designing a domain-specific query language, so as to improve the levels of expressiveness for the queries being asked? Are the varying languages really all that different? This is a question yet to be answered.

4 Case Study: Implementation

As part of this study, subsets of EmuQL and LPath+ were implemented. LPath+ contrasts EmuQL as it is a path-based query language, whereas EmuQL is expression-based. This difference in how the queries are formed makes the languages interesting for comparison. If similarities arise from the analysis, it would indicate that despite their differences, they are similar underneath their syntax. Subsets of both query languages were implemented as this case study was largely concerned with the similarities and overlap between the languages, though some additional differentiating features were implemented where possible. Enough of both query languages were implemented to cover the selection of sample queries used for evaluation. This case study is used for the analysis of the query languages in question, as well as considering domain-specific query languages as a whole. This chapter will discuss the implemented case study, the components involved and the test results.

4.1 Data

To implement and test a query system, there first needed to be a data set to query. Most of the data used for this test study came from Alveo[1], primarily the Mitchell-Delbridge [30] data set. This data set was used as a sample of linguistic annotations which contains tiered data. Alveo is a large scale repository containing language data and the relevant tools for research. In addition to storing large collections of language data, Alveo also included the metadata for the various data stores, annotations, and a SPARQL end-point for querying the data. The data in Alveo is stored in RDF. RDF was used as the underlying data format in this study on which to build the annotation data model as it was able to represent multi-tier graph data and had a developed and standardised query language i.e. SPARQL.

The Mitchell-Delbridge data set was chosen for this study as it is one of the few collections

[1]`http://alveo.edu.au/`

in Alveo to have multi-level hierarchical annotations associated with most of the data. The data consisted of audio recordings of speech. The annotations on the data included timestamps, hierarchy tiers, words, phonemes and phrases in the speech signal. The annotations are stored in the Alveo system as part of an RDF graph that also included the metadata about each individual recording. In the annotations, the words and phonemes were linked to time nodes which described the start and end times. No explicit hierarchical relations were stored between the words and phonemes. As the words and phonemes were not linked to each other, their relation had to be inferred based on the timestamps provided by the start and end times of each annotation.

4.2 Software

The software implementation for this study was done using Python 2.7, in addition to third-party libraries. Python was used due to its modular nature as well as its extensive collection of community-made libraries. It is also already used in other parts of the Alveo system. As a common language with an established user-base, the use of Python for this project also increased the likelihood of continual use and maintenance by the community.

The libraries used in this project, not including the dependency libraries this list requires, are as follows.

- PyParsing - a Python parsing library. This library was used to create the grammar to parse the tested query languages. This library was chosen due to it being an established and popular choice. This would ensure that it would remain relatively up-to-date and maintained.
- RDFlib - a library for handling and processing RDF files. It provided an API for both reading and writing RDF files. Like PyParsing, RDFlib is a popular library and a standard choice for this purpose.
- SPARQLwrapper - a Python library used for SPARQL endpoint interfacing. This library was used as it is both popular and powerful. Popularity is often a deciding factor as it meant a library would be more commonly used and have a larger support and user base. It also increased the likelihood of the library receiving updates and developer support.
- PyAlveo[2] - a Python library used for interfacing with the Alveo API and data store. Alveo was used as the data store on which the parsed and converted queries were initially tested. Alveo contains a large collection of linguistic annotations, and was thus suitable for testing the domain-specific query languages EmuQL and LPath+.

4.3 Parsing A Query

The structure of a query greatly affects what a user is able to ask of a data store. The design of the language may restrict the expressiveness capability of a language, or improve

[2]`https://github.com/Alveo/pyalveo`

it. For a query language to be effective, the system needs to be able to understand what the user is asking. For the purpose of comparison, one way of evaluating and analysing differing query languages would be to convert them into the same format, if possible, and compare the end results. Converting the query languages into the same format makes it clear what each language is capable of expressing, as well as whether or not the compared languages are really asking the same question. For this, the query languages need to be parsed.

An additional benefit of using a domain-specific query language, aside from the expressive power available, would be that due to the restricted types of questions a user may ask, there are fewer possible variations in what the language needs to handle. This makes for lighter work in regards to parsing a variety of queries and making a parser for a query language.

The easiest way to create a parser for a query language would be to find the grammar specifications for a language. While complete published grammars are available for some languages, this wasn't the case for EmuQL and LPath+. Though both languages had some specifications, EmuQL was defined through examples, whereas the LPath+ grammar was incomplete. The grammars used for this study were consequently defined based on what specifications and examples were available. This resulted in parsers for variants of the query languages.

A subset variant of both EmuQL and LPath+ grammars were implemented and examined. Features shared by both query languages were focused on for consistency, along with features present in one but not the other. This highlighted the expressiveness of the languages and what is possible within the domain. This subset variant will act as a representative example of both query languages, leading into an analysis of the query languages as a whole, including the unimplemented portions.

4.3.1 EmuQL

The Emu query language was one of the query languages chosen for this case study. As a domain-specific query language for linguistic annotations, EmuQL was able to handle some concepts that are common in linguistics that are not as prevalent in other query languages, such as sequence. The Emu query language also had the benefit of being a relatively simple query language, compared with some of the other available choices, making it more suitable for short analysis.

A subset and variant of EmuQL was implemented for the case study tests, based on the available documentation [28]. Rather than implement the language in its entirety, the point of the case study was to compare two query languages, so enough was implemented to allow this.

The Emu query system handles the ordering of tiers in linguistics via its schema e.g. the language is able to determine that the orthography tier dominates the phonetic tier without this being explicit in the data. This schema is independent of the data, and whilst it was known, it was not a part of the available data or metadata and hence not part of the implementation. This resulted in a variant of the language being implemented, as the original specifications for EmuQL take advantage of this feature by allowing the specifying of nodes to be independent of ordering i.e. "[X^Y]" can mean either X dominates Y or Y dominates X, depending on what the values and tiers of X and Y are, whereas the implementation used

```
<exp> ::= <simpleq> | <complexq>

<simpleq> ::= <exp_basic> | <exp_basic> "&" <simpleq>
<complexq> ::= "[" <exp_basic> <operator> <exp_basic> "]"

<operator> ::=   "^"  |  "->"
<exp_basic> ::= "#" <g_string> <g_equ> <g_vals> |  <g_string> <g_equ> <g_vals>

<g_string> ::= <g_text> | "‘" <g_text> "’"
<g_text> ::= <token matching regular expression>

<g_equ> ::=  "=" |  "!="

<g_vals> ::= <g_string> |  <g_string> "|" <g_vals>
```

FIGURE 4.1: The grammar that was implemented for a subset of EmuQL.

for the case study is order-dependent and in this example would mean X dominates Y.

The grammar that was implemented for EmuQL is shown in figure 4.1. It generates the abstract syntax tree (AST) by first traversing down the query to find the simplest parts of the query i.e. the basic query mentioned in section 3.5.1. From there, the parser works its way back out of the query, handling each part as if it were a basic triple of the form node-relation-node, even if one of the node parts is another query in its entirety. The query is parsed and deconstructed in reverse of the pattern shown in figure 3.8. The AST result of the parser is described in section 4.4.1.

4.3.2 LPath+

LPath+ was the other query language chosen for this case study. LPath+ is a query language designed specifically for linguistic trees, and is thus suitable for use on linguistic annotations. Like EmuQL, a subset of LPath+ was implemented for the case study. A subset of the language was implemented as the study was largely concerned with the overlap of features between EmuQL and LPath+. The basic structures of LPath+ were implemented, and while the LPath+ parser will handle a large proportion of the queries, it isn't able to handle more complex concepts that exist within the predicates, which are represented with square brackets in LPath+, or with scoping. In this study, there is no compiler implementation for some of the axes handled by the parser, as there were either no EmuQL or SPARQL equivalents e.g. EmuQL and the implementation of the SPARQL generator did not have handling for the descendant concept.

The grammar that was implemented for LPath+ is shown in figure 4.2. As LPath+ is a path-based query language, the parser handles the query from left to right, keeping track of variables where required due to predicate usage. The AST result of the parser is described in section 4.4.1.

```
<fexpr> ::= <locpath>

<locpath> ::= <axis> <steps>

<axis> ::= "\\*" | "\\" | "\" | "." | "//*" | "//" | "/" | "-->" |
           "<--" | "->" | "<-" | "==>" | "<==" | "=>" | "<="

<steps> ::= <nodetest> | <nodetest> <axis> <steps>

<nodetest> ::= <test> | <test> "[" <pred_opt> ]" |
               <test> "[" <pred_opt> ]" <closure>

<test> ::= <p> | "^" <p> | <p> "$"
<pred_opt> ::= <fexpr> | <attr_test>
<attr_test> ::= "@" <p> "=" <p>

<p> ::= <token matching regular expression>
<closure> ::= "?" | "*" |"+"
```

Figure 4.2: The grammar that was implemented for a subset of LPath+.

Due to being designed for relatively simple linguistic data, the usage of the query language had to be adapted slightly for use with this study's different data model. The original data model used by LPath+ was a linguistic tree that did not specify the type of each node. As the linguistic annotation data used for this study is a multi-type hierarchical graph, the type of each node in the query needed to be specified through the use of attribute tests in predicates, as they were not all of the same type as had been assumed as part of the original language. This is still in line with the specifications of the original language though; just not necessarily the intended use. An example of the equivalent query both with and without these attribute tests is shown in figure 4.3.

```
Original LPath+:
//time/Ae

This study's LPath+:
//time[@dada:type=maus:orthography]/Ae[@dada:type=maus:phonetic]
```

Figure 4.3: Original LPath+ did not specify node types. This study works around this by using attribute tests.

This specification of node type can be omitted and the query result would still be valid, but the query would then be much more dependent on the data labels being unique to each tier, and the results would be incorrect if a wildcard character were used in place of a node label. For example, in figure 4.3 the label "time" would only appear on orthography nodes, and "Ae" would only label phonetic nodes. In this instance, omitting the attribute tests would not affect the query. Contrastingly, in figure 4.4 without the attribute test, the query cannot ensure that the parent node is on the orthography tier, as there are no restrictions on what the node can be labelled.

```
Original LPath+:
//_/Ae

This study's LPath+:
//_[@dada:type=maus:orthography]/Ae[@dada:type=maus:phonetic]
```

Figure 4.4: In cases using the wildcard character, the node type cannot be ensured without an attribute test.

4.4 Understanding A Query

Once a query had been parsed, it was converted to an internal format. Whilst it was possible to go directly from parsing a query to interacting with a data store, converting the query to an intermediate format made query execution simpler. This was because it allowed both checking to ensure that the query had been parsed and understood correctly, as well as allowed conversion to a format that was more easily handled. In this study, the queries were eventually converted into SPARQL, as the data was in RDF.

An advantage of using an internal representation was that by converting all the query languages analysed into one format, it created a consistent standard that made analysis and comparison easier. It allowed a singular benchmark to be used in comparing the languages and evaluation. For example, the internal representation may allow a query language to specify optional peripheral parameters. This option would persist in the internal format, and whether a query language made use of it would highlight the extent of a query language's expressive ability. In addition to that, it also allowed for later conversion from the internal representation to some other query language i.e. SPARQL. In essence, by acting as an intermediate point, an internal format created a bridge between multiple query languages. This assisted in converting between query languages.

The design of the internal format is important as it is necessary that it is as expressive, if not more, than the original query languages being parsed and converted. This ensures that no data is lost and the end format is able to express the full potential of the original queries.

A common internal or intermediate format is the abstract syntax tree (AST). There is no standard AST, but rather the structure of an AST is based on the original query language from which it was generated. As more than one query language was involved in this study, rather than try to develop a suitable AST from scratch, an easier technique was to model the AST after one of the languages, capturing all the expressiveness of the language, then do the same to the other query language. Once both ASTs were obtained, both trees were compared to each other for similarities and differences. In the end, the trees were combined, incorporating the differences between the languages into the same tree. With this end AST in mind, when creating the ASTs for each language, the design was created to be as generic as possible. While this combined AST may result in certain aspects of the tree not being used in one of the languages, it does capture all aspects of both languages. By being able to convert to a singular AST, this task also highlights the similarities between the query languages, as well as the possibility that all query languages in the domain are based on the same underlying query structure.

4.4.1 Shared Internal Tree

The abstract syntax tree used by both the EmuQL and LPath+ parsers is shown in figure 4.5. The tree is recursive, and so is able to re-use concepts. To explain this, the various parts have been separated so that it is clearer to visualise the components. An example showing how EmuQL and LPath+ use this tree is shown in section 4.6.

The AST consists of an expression "exp", as seen in figure 4.5, which generally contains a connector, left node and right node. Note that not all parts of the tree are required to be filled in, as one query language may handle features that another may not. The connector is also known as an axis, and is the relation that connects nodes e.g. the "sequence" relation which is represented as "->".

The left and right nodes may be leaf nodes, or more complicated expressions. Whilst for the most part the left and right nodes follow the same structure, a difference between them is that the left node can also use a "hash", whereas the right node has the option of an "or_group". The "hash" is used by EmuQL to mark which node the user is most interested in whereas the "or_group" is used recursively to represent a list of options e.g. "maus:orthography='cat'|'dog' ".

Once the AST reaches a leaf node, the leaf node may contain the text value, a predicate and/or attribute test. Predicates allow for optional peripheral expressions on a node, where the end goal node is not a part of the peripheral expression. For example, finding a node labelled "t" which is a child of an orthography node. The parent orthography node may have other specifications, such as a label. These details, while important, do not need to be traced to get to the end child node, and as such would be specified within a predicate.

As both the EmuQL and LPath+ parsers are able to use the same AST to an extent, it highlights similarities between how the two languages handle queries, as well as the scope of what one is able to express. So while both query languages appear to be different in form and structure, when stripped of syntax and reduced down to an AST, both query languages are fairly similar in their capabilities and how they function.

From the AST, the differences between the two can be seen. For example, EmuQL is able to return more than one result node and has some handling for scoping. LPath+, on the other hand, permits the usage of predicates and nested predicates. That being said, these differences are a result of the implementation and how a query may be structured. For functionality similar to LPath+'s predicates, a user could use the "&" connector in EmuQL. Similarly, the LPath+ specifications do include handling for scoping through the use of braces i.e. "{ }" [17]. An example of a query that is expressible in one language and not the other would be LPath+'s ability to query for an ancestor. Once highlighted by the

```
exp:
        connector:‘’
        right:
                or_group:
                node:
                exp:
        left:
                hash:‘’
                node:
                exp:

node:
        text:‘’
        attr_test:
        predicate:

predicate:
        exp:

attr_test:
        attr_val:‘’
        attr:‘’

or_group:
        text:‘’
        or_group:
```

Figure 4.5: The shared internal AST used by both the EmuQL and LPath+ parsers. This is the internal representation of query languages used by the compiler before the query is converted into SPARQL.

combined AST, such differences could then possibly be considered and incorporated into an improved query language for the domain and data.

4.5 Reconstructing The Query

Once the queries had been parsed and converted to an internal format representing a unified AST, there were two possible implementation choices in order to retrieve the appropriate results. One possible option was to compile the AST into an existing query language and then execute the generated query. The other option was to implement a full query engine with direct access to the data store. Each option had its own benefits. Implementing an engine and running a query directly on the data store may offer greater flexibility due to not being restricted by constraints introduced by other languages. On the other hand, converting from the internal format to another query language was simpler implementation-wise depending on the target query language chosen. This was because the query language may have had an existing implementation and accompanying query system, and by using a pre-existing system, the implementation could take advantage of any query optimisation already implemented as part of the query system. This was the case with SPARQL, which was used for this study.

Converting the internal format to another query language removed the complications of handling the end query, as the implementation no longer needed to directly interface with the data store. This was handled by the query language interface. Another benefit of converting to an intermediate query language which would then run on the data store was that, like the conversion from a query language to a unified AST, the conversion from the AST back into a query language emphasised the similarities between all query languages used for the shared domain. It underlies the notion that there existed a query model that all query languages in the same domain could be mapped to, which expressed the queries fully.

Being able to view the generated query language also made verifying the AST conversion simpler, presuming that the final query language implementation was a correctly functional one. This could be ensured by using a standard and common implementation. The verification and checking was made simpler due to it being possible to trace the query to make sure that the compiler generated the AST and resulting query language correctly and as expected.

The data queried in this study was in RDF and the standard query language for RDF is SPARQL. This study used SPARQL as the target query language that the other query languages were eventually converted into after the conversion to the internal AST. It was this final SPARQL that was run on the data store via a SPARQL end-point interface.

As each of the parsers and ASTs had been developed independently of each other, with the idea being that they would be merged in the end, each also had their own implementation of the SPARQL generator. As a result of this, whilst the SPARQL statements were essentially the same, there were variations in regards to the ordering of the WHERE clauses in the SPARQL. For example, one version of the generated SPARQL might have the clause with the string value at the start, whereas the other may have the clause at the end of the query. This can be seen in section 4.6. This had an unexpected effect on the running time of the queries. The variations between how the queries from each language are converted into and use the same AST, and the normalisation of this conversion process is an area for possible future work, where potentially they could be converted to use the tree in the same way. This would allow for a single implementation of the SPARQL generator, which would remove these variations.

4.6 Worked Example

To demonstrate these components, consider a query searching for a phonetic node labelled "Ae" which follows a phonetic node labelled "t". In EmuQL, this is written as:

```
[maus:phonetic='t'->#maus:phonetic='Ae']
```

This query is parsed by the parser and re-written using the internal AST representation. The AST representation for this query is as follows.

```
exp:
        connector:->
        right:
                connector:=
                right:
                        text:Ae
                left:
                        text:maus:phonetic
                        hash:#
        left:
                connector:=
                right:
                        text:t
                left:
                        text:maus:phonetic
```

Once the internal AST is generated, the compiler generates the corresponding SPARQL. In this case, the SPARQL generated for the EmuQL query is:

```
select ?var2
where {
        ?var1 dada:type maus:phonetic.
        ?var1 dada:label 't'.
        ?var2 dada:type maus:phonetic.
        ?var2 dada:label 'Ae'.
        ?var1 dada:partof ?var3.
        ?var2 dada:partof ?var3.
        ?var1 dada:targets ?var4.
        ?var2 dada:targets ?var5.
        ?var4 dada:end ?var6.
        ?var5 dada:start ?var7.
        filter( ?var6 = ?var7).
}
```

Using the same example, the LPath+ query is written as:

```
//t[@dada:type=maus:phonetic]->Ae[@dada:type=maus:phonetic]
```

The structure of an EmuQL query is different from that of an LPath+ query. This can be seen from the previous balanced AST representation of the EmuQL, versus the following right-leaning AST representation of the equivalent LPath+ query.

```
exp:
        connector://
        right:
                connector:->
                right:
                        text:Ae
                        attr_test:
                                attr_val:maus:phonetic
                                attr:dada:type
                left:
                        text:t
                        attr_test:
                                attr_val:maus:phonetic
                                attr:dada:type
```

The SPARQL generators for EmuQL and LPath+ were written independently of each other. Each used their relevant AST representations to generate SPARQL. A consequence of this is variations in the ordering of the WHERE clauses. This can be seen in the following SPARQL generated for the LPath+ query.

```
select ?var2
where {
        ?var1 dada:label ‘t’.
        ?var1 dada:type maus:phonetic.
        ?var1 dada:partof ?var3.
        ?var2 dada:partof ?var3.
        ?var1 dada:targets ?var4.
        ?var2 dada:targets ?var5.
        ?var4 dada:end ?var6.
        ?var5 dada:start ?var7.
        ?var2 dada:label ‘Ae’.
        ?var2 dada:type maus:phonetic.
        filter( ?var6 = ?var7).
}
```

4.7 Query Results

Unit tests were constructed to test SPARQL’s ability to answer the questions being asked in section 3.5, as well as to ensure that the parsers and converters behaved as expected. Once this was certain, these queries were used to evaluate the expressiveness of EmuQL and LPath+.

Different variations of the queries in section 3.5 were tested in both EmuQL and LPath+, and their generated SPARQL queries and query results were compared. The results returned from the equivalent EmuQL and LPath+ queries matched, showing that they were able to

```
SELECT ?var2
WHERE {
    ?var1 dada:type maus:orthography.
    ?var1 dada:label ‘time’.
    ?var1 dada:hasChild+ ?var2.
}
```

Figure 4.6: SPARQL: Return all descendant nodes of the orthography node labelled “time”.

express the same queries, and that their generated AST and SPARQL queries functioned as expected.

The equivalent EmuQL and LPath+ queries did result in running time variations. Additional tests were run to trace the generated SPARQL. It was understood that the variations were a result of differences in the ordering of the WHERE clauses in SPARQL.

A summary of the query results for both EmuQL and LPath+ follow.

- Basic - Both EmuQL and LPath+ successfully ran basic queries. These queries asked for the minimal requirements of both languages. An example of this query asks for a phonetic node labelled “t”.

- Sequence - Both EmuQL and LPath+ successfully ran sequence queries. As explained in section 3.5.2, these queries were formed from a combination of basic queries. An example of this query finds a phonetic node labelled “Ae” which follows a phonetic node labelled “t”.

- Dominance - These queries were interesting as the data did not originally contain the “hasChild” relation. The queries needed to infer this information based on the tiers in the data as well as shared and overlapping time nodes. A dominance query may find a phonetic node labelled “Ae” that is dominated by an orthography node labelled “time”. This query type could be further explored by adding in the “hasChild” relation to the existing data.

- Transitive closure - While the transitive closure queries were interesting, the version of SPARQL used as part of the implementation for this study was unable to handle transitive closures. Consequently the full implementation of these queries were not tested, though the LPath+ parser was able to successfully parse the queries. EmuQL was unable to express transitive closures, as explained in section 3.5.4. It should be noted though that other implementations of SPARQL are able to handle transitive closures. An example of SPARQL with a transitive closure is shown in figure 4.6. Though this example assumes the presence of the “dada:hasChild” predicate, it does show how the descendant concept can be queried for using a transitive closure.

- Wildcard - These queries asked questions that may fall into one of the other categories, with the exception that at least one of the variables were left open-ended. This increased the amount of matching results. An example of a wildcard query would be searching for any phonetic node that is followed by a phonetic node labelled “r”. Whilst both EmuQL and LPath+ had an implementation for the wildcard concept, only LPath+ had the character built-in for that purpose. EmuQL required the query language to be used in a way that wasn’t intended, through the use of negation and prior knowledge of the contents of the database.

- “Not” and “Or” - These queries ask special boolean variants of the basic query. The “not” query is a negated equivalent of the basic query e.g. find an orthography node that isn’t labelled “time”. The “or” query is a variant of the basic query where multiple literals could match the node e.g. find a phonetic node that is either labelled “Ae” or “m”. Whilst both query languages were able to express the “not” concept as well as the “or” concept, these concepts were not implemented for the LPath+ parser. This is because in LPath+, these concepts are more expressive in what they are able to ask and how, thus making it more complicated to implement. Additionally, they exist in LPath+ as an extension of XPath, and while there was mention of handling for these concepts in the specifications, we couldn’t work out from the published description how this worked in LPath+. The EmuQL parser and SPARQL generator were able to successfully handle these queries. As these queries were more complex, the running time was longer than the equivalent basic query i.e. finding an orthography node that isn’t labelled “time” took longer than finding one that is labelled “time”.

- Nested queries - Nested queries were useful for testing complex query expressiveness, as they are formed from stacked simpler queries. For example, such a query may find any phonetic node that follows another phonetic node, where the node being followed is a child of an orthography node labelled “time”. These queries were successful in both the EmuQL and LPath+ implementations.

- Siblings - These queries increased the complexity from that of the nested queries. In addition to nesting and possibly sequence tests, there was an additional dominance test. For example, a sibling query may involve finding two phonetic nodes, which are the children of an orthography node labelled “time”. The difference between these queries and the nested queries is that in these queries, both phonetic nodes are children of the orthography node labelled “time” i.e. the nodes are siblings under the same parent node. In the nested query example, the dominance relation of only one of the nodes was important. These queries highlight the usage of scoping. The parsers for both EmuQL and LPath+ were able to successfully handle these queries. This concept could also potentially be formed from a combination of domain-specific concepts i.e. the “hasChild” and “followedby” relations.

- Empty - Both EmuQL and LPath+ successfully ran empty queries. These queries intentionally searched for non-existent data, as an edge case. Such a query may ask to find a phonetic node labelled “r” which is dominated by an orthography node labelled “time”. As the sound “r” does not occur in the word “time”, the result set should be empty.

Overall, for the most part, EmuQL and LPath+ had similar levels of expressiveness. In the implementation used for the case study, the Emu query language was more expressive, as it was able to use boolean concepts, such as “not” and “or”. If taking into account the language specifications, LPath+ would be more expressive as it also handles transitive closures, as well as more complex scoping structures and a wider range of functions.

4.8 Conclusion

Overall, the expressive level of both EmuQL and LPath+ are similar, based on the implemented case study. While there were variations, these were related to the implementation. Despite the differing syntactic structure of both query languages, they could both express similar questions. Both languages could be reduced to a single combined AST representation, though both languages didn’t use all the features of the internal representation. Additionally, there were slight variations in how each language used the AST. Once this has been shown by the combined AST, such differences could then possibly be considered, normalised and incorporated into an improved query language for the domain and data, as part of future work.

5 Discussion

Overall, query languages allow a user to ask specific things that they may be interested in from a data set. They provide a relative degree of flexibility and expressiveness in what they can and can't ask. As the degree of expressiveness increases, so too does the computational complexity required to perform such a request. This level of expressiveness often depends on a variety of factors, such as the data model and the language design.

Having performed the case study in chapter 4, this chapter will discuss the results in detail, both of the implementation as well as of the analysis of EmuQL and LPath+ overall. It will cover the implemented and non-implemented expressiveness of the query languages, the similarities in their underlying structure, as well as the effect of the implementation on running time. This will highlight possible areas for future work.

5.1 Language Design and Expressiveness

This study focused on EmuQL and LPath+, eventually converting queries from both languages into SPARQL. The SPARQL queries were then evaluated against the data store. As SPARQL was used as the target language for executing queries, it was important that it was expressive enough so as to not interfere with the expressiveness of both EmuQL and LPath+. SPARQL was chosen as it was the standard query language for RDF, which was used to store the data.

SPARQL is able to express a lot of the concepts that a linguist may want to ask of the data, such as "equals", "not equals", "dominance", "sequence", and therefore by inference the "parent-child" and "sibling" relations. It is also able to return more than one node as part of the result set. While SPARQL does have handling for transitive closures, as mentioned in section 4.7, the SPARQL generator used for this study was not able to express this concept.

5.1.1 EmuQL

EmuQL is an example of an informally defined language. Whilst there are rules in regards to what constitutes a valid and invalid query, these are in plain text and provided as a series of examples as opposed to in the form of a formal grammar. As such, the specifications are open to interpretation. This is not ideal for the language, though the language is relatively limited in scope and expressiveness, in comparison to LPath+. The Emu query language and query system target a specific user-base, and while restrictive in what it is able to express, it is effective at its goal due to focusing on a specific domain. It does not matter that it has a limited expressiveness overall as long as it can express what is required within its domain.

Whilst EmuQL is able to express many of the same questions as SPARQL, EmuQL cannot ask all of the queries that are possible in SPARQL. One of the limitations of EmuQL is that it can either return one node, or all of them. This is not a problem with SPARQL as the user can define which nodes they want as part of the result set. Due to this limitation, EmuQL is unable to return pairs that would otherwise have been possible in SPARQL, such as sibling pairs, or node and label pairs. Whether or not this is something a linguist may want to ask is open for further study.

EmuQL also does not have extensive handling for scoping i.e. look for a subset of the data, and perform the rest of the query using that subset. To a degree, there is some handling for this, depending on how a user structures the query. EmuQL's handling for scoping is a workaround based on the language, similar to its handling of the wildcard concept. It is dependent on a combination of ordering, the use of "#", the relations used, amongst other things. A more general approach to the problem of scoping would be to run the first part of the query on the entire data set, then run a second query on the result set of the first. Essentially, two separate queries need to be executed. This could also be treated as a nested query, and could potentially enable a user to perform more complicated set queries such as unions. While the Emu query language does not have handling for this, it is implemented as part of the Emu query system and is called "requery". This solution is general in that the approach can be done in most, if not all, query languages.

For this study, a subset of the Emu query language was implemented. The implemented parser and converter were able to handle most of the EmuQL query concepts. Whilst there was some handling for nesting in place, it was not the optimal implementation of it, and could not handle complicated nested queries. A part of this problem was due to the study's implementation of EmuQL being order-dependent, whereas EmuQL's original specifications allow order-independence due to use of a data schema. This data schema was not available as part of the data. A consequence of this implementation variation was that it created complications regarding ordering.

Features that are part of the Emu query language that have not been implemented include functions, proper handling for more complicated nested queries and domain-specific data model schemas. Functions are a feature that exist in the query language, as a sort of shorthand for more complicated concepts. As part of the specifications of the Emu query language [29], it is supposed to be able to handle the function concepts of "Start", "End" and "Num". The functions return a boolean true or false, depending on if the data matches the query part. As SPARQL does have some function handling, the possibility of implementing EmuQL's function handling through the use of SPARQL's function handling becomes an available option for future implementation.

EmuQL works by finding triples that return true for query parts. This is more evident in the way the functions operate. That said, because this case study is not implemented the same way as the original EmuQL compiler, it does not do this. The implemented case study uses an EmuQL parser, converts the parsed EmuQL into an internal format, which then gets passed into a SPARQL generator. The result is a SPARQL version of the EmuQL query.

5.1.2 LPath+

LPath+ is a formally defined language with a specified grammar. The grammar rules define what is and isn't valid in the language, differentiating it from less defined languages such as EmuQL. The formal definition theoretically makes it easier for a user to form a valid query, as they would not need to base their queries on examples, but rather by the specifications.

LPath+ is based on and inherits from XPath. Many features from XPath – whilst valid in LPath+ – are not addressed in the grammar, such as function use, attribute referencing and index selection from a collection of result nodes. They are still employed though in the accompanying example queries. This creates an incomplete grammar, where cross-referencing with the XPath specifications is required.

Additionally, there is a flaw in the grammar given for LPath+. Whilst valid LPath+ queries may be parsed with the specified grammar, nonsensical queries are also valid according to the grammar [17]. For example, the "–>" axis is used to represent the concept of "followed by". This concept requires two entity nodes as it represents a sequence. The grammar though states that all queries begin with an axis, without limitation to what this axis may be. This therefore means that a query may start with "–>". To correct this, the axes should be separated into two groups, differentiated by those which can and those which can't start a query. They would still be used as they currently are in the rest of the query.

Whilst LPath+ is an expressive language, it differs from SPARQL in that it is unable to return more than one result node. This is a consequence of LPath+ being a path-based language, as paths end on a singular node. On the other hand, it is able to express some transitive closure queries which are built into the language, such as ancestor queries. Overall though, the expressiveness of LPath+ is relatively similar to SPARQL. Like both SPARQL and EmuQL, LPath+ has some degree of function handling, though between the three languages, LPath+ has a wider selection of functions, due to having inherited the functionality from XPath. For the most part, LPath+ has similar levels of expressiveness as XPath. A difference between them being that LPath+ implements concepts which are unique to linguistic annotations, but this is merely on the surface level. A syntactic sugar, if you will. The implementation of additional axes unique to LPath+ can be executed in XPath by specifying the sort of axis to be used, as shown in figure 5.1 where the "immediate following" axis represented by "->" can be expanded in XPath. LPath+'s implementation of these concepts are not to increase levels of expressiveness, but rather to improve on usability.

Unlike this study's implementation of the Emu query language, its implementation of LPath+ is less featureful. Whilst most of the basic functionality is handled, the more complicated concepts such as predicates are only handled to a limited extent. Importantly, the use of more than one predicate filter per node is unavailable, which makes certain queries

```
LPath+:
//N->VP

XPath:
//N/immediateFollowing::VP
```

Figure 5.1: LPath+'s axes can be expanded in XPath.

not possible, as it requires the user to be able to specify multiple features about a single node. Additionally, boolean functionality and functions in general are not handled.

Whilst the EmuQL implementation in this study is more powerful and expressive than the LPath+ implementation, LPath+ as a query language, based on the specifications is more powerful and expressive. This in itself is part of the reason for it not being implemented i.e. the expressiveness and complexity involved in doing the conversion, given the time constraints on the project, would have been difficult.

5.2 Internal AST Representation

Both EmuQL and LPath+ were parsed into an internal AST representation before being converted into SPARQL. This internal representation is described in section 4.4.1. By converting both languages into a similar internal representation, it made it easier to check the parsers, as well as ensure that the queries were asking the same questions. The internal AST representation also provided another level in checking as it became possible to the trace the conversion into SPARQL.

While the AST representation of both EmuQL and LPath+ were developed independently of each other, they were designed with the end goal of a singular representation in mind. Consequently, both were designed to be as generic as possible while still capturing all the features of both query languages. The result of this is that it was then possible to combine both ASTs into a single AST representation that was able to handle the parsed versions of both query languages. Additionally, it was possible to convert this AST into SPARQL.

The ability to seamlessly convert between the tested query languages and this internal AST representation of the queries emphasises the similarities between the query languages in this domain. It implies that the underlying query structure between the tested query languages may be the same for this domain, and that the differences between the languages is syntactic and surface form. So while the query languages chosen appear to be different, in their shared domain of linguistic annotations, the queries aren't that much different from each other and are designed around and limited by the questions in the domain.

It should be noted that while EmuQL and LPath+ queries can be converted to the internal AST representation, how each language uses the AST differs slightly. This is a consequence of the ASTs being developed independently of each other, as well as due to the structure of the query languages. LPath+ is a path-based query language, and when its queries are parsed, the resulting AST is a right-leaning tree. The ASTs generated by EmuQL queries on the other hand are balanced trees. Due to the different ways in which the query languages use the AST, the generated SPARQL also varies slightly. This variation in the generated SPARQL resulted in differences in some of the running times of what should have been equivalent queries in both languages.

5.2.1 Timing Queries

In addition to the tests on query expressiveness between the query languages, extra running time tests were conducted for some of the queries. This was a result of some of the generated queries running notably faster or slower than the equivalent generated queries by the other language. For example, some of the SPARQL queries generated by the LPath+ compiler ran faster than the queries generated by the EmuQL compiler. This was of interest as both compilers converted the queries into SPARQL, and so in theory, they should have had similar running times. After some investigation and testing, it was understood that the difference in running time was a result of the ordering of the generated SPARQL statements. Depending on the order of the WHERE clauses in the generated SPARQL queries, the running time was faster or slower. The ordering of the WHERE statements was different between the EmuQL generated SPARQL queries and the LPath+ generated SPARQL queries as the compilers for each were built separately, even though they both relied on the same AST representation. This optimisation of the generated SPARQL is a possible area for future work on the study.

In order to carry out the timing tests, variants of the generated SPARQL queries were compared with the equivalent handwritten SPARQL queries. These variations included choices regarding order, the amount of variables to use, as well as whether filters were used to equate variables as opposed to reusing variables.

Additional domain-specific relations could also potentially be added to the data. This would allow comparing the effects of domain-specific concepts being present in the data versus them being implemented as part of the query languages. While doing so would involve extra data pre-processing, it could potentially improve query running time due to reduced query complexity. Essentially some of the work carried out by the query engine could be done by data pre-processing. Query optimisation as part of domain-specific data inference is an area open for further investigation.

5.3 Future Work

There is still more work to be done in regards to the implementation of both the EmuQL and LPath+ parsers and compilers. The implementation for both query languages can be improved to cover more query cases and features, so as to more closely resemble the original language specifications.

Regarding the EmuQL implementation, the handling for complicated nested queries needs more work. Part of the reason why it handles the queries incorrectly may be down to the way in which a query is parsed and traversed by the compiler. The query is handled left to right, and so when used in conjunction with complicated nesting, the correct node may not always be selected. A possible solution would be for the parser to parse the query and generate the resulting AST from the simplest component and work its way out. Other areas for future work in regards to the EmuQL parser and compiler are the addition of function handling, as well as allowing the use of a data model schema to specify hierarchy, as with the original version of the Emu query system. The latter would allow the nodes in the language to be order-independent, which would assist with the expressiveness of the language.

In respect to the LPath+ implementation, the parser still needs to be able to handle multiple predicates per node, as well as the boolean concepts and functions. Scoping is an important feature that also needs to be handled. Whilst the ancestor and descendant concepts are important in respect to the expressiveness of the language, the implementation of the SPARQL generator used as part of the compiler would first need to be updated before these concepts can be used. The parser is already able to handle these concepts.

Both the EmuQL and LPath+ implementations of the compilers do not handle the "sibling" concept, even though the language specifications for both allow it. Due to the nature of the data model, this would take time to implement. The amount of work required to add this concept could be reduced through the use of the additional domain-specific relations "dada:hasChild" and "dada:followedby". By adding the relations to the data rather than using the current implementation, it would be possible to allow the enforcing of scoping, which the current version does not allow for. It would also result in running time improvements over the current implementation.

Adding new relations to the data via pre-processing would reduce the complexity of a potential query. For example, the concept of "followed by", such as the phonetic sound "t" being followed by the phonetic sound "Ae", may be implemented directly in the data. This pre-processing would save the query engine from needing to infer this relation through a series of timestamps each time the query is executed, thus reducing running time. This is another possible area for further investigation.

The compilers for both EmuQL and LPath+ generate an AST based on the queries. While the structure of the AST for both query languages is shared, their use between the languages differ. An area of further work for the AST design would be to parse both languages to use the AST in the same way i.e. the equivalent queries in EmuQL and LPath+ would generate the same AST, or trees that were as similar as possible, given the variations in each language. By producing the same, or similar AST, this would potentially eliminate the variations caused by the SPARQL generator. Consequently, queries in each language could be considered equivalent. This would further emphasise the shared abstract structure of domain-specific query languages in linguistic annotations, and could potentially lead to the development of an improved query language for the domain and data.

Additionally, the SPARQL generated from the ASTs could be further optimised for faster results, based on the examples and preliminary testing seen in section 4.7. For example, the ordering of the WHERE clauses could be optimised to ensure faster running times. Determining which methods would optimise the SPARQL queries would require further study.

The point of improving these implementations would be to enable a closer examination of the design and construction of a sample of domain-specific query languages, to draw conclusions on domain-specific query languages as a whole.

6 Conclusion

Both EmuQL and LPath+ are expressive in the domain of linguistic annotations. Whilst both are effective in their own right as query languages, in the implementation used for this study, neither language cover all cases that a user may want to express. Part of the reason for this limitation lies in the design of the query languages, whilst another in the structure of the data model being used. Both languages were designed for linguistics, though neither were designed for the specific data model used in the study. They were adapted to fit the data model and the queries were converted to SPARQL before they are run on the data store. In certain cases, the limitation lay with the expressiveness of the implementation of the SPARQL generator. Given more time, the compiler could be improved to be able to express the required queries, such as transitive closures.

As part of implementing and understanding the structure of the query languages, EmuQL and LPath+ were analysed and used to generate an internal AST representation of the query languages. This was done in respect to the queries used for evaluation. The ASTs of both languages were then combined to form a singular AST whilst retaining as much detail from both languages as possible. The AST generated from this task was able to represent both query languages, indicating that despite their differing surface forms, their underlying structures were similar. This was further emphasised when the AST was used to generate SPARQL queries without much difficulty. It is understood that for the query languages used as part of this study, as different as they were, they had a similar, if not the same, query structure.

The internal AST representation of query languages in linguistic annotations is a generic model, in that it isn't specific to one query language. Given the scope of this study, it would be possible to extend the project by incorporating other query languages in linguistic annotations into the design in both directions i.e. using the query languages to generate the AST as well as using the AST to generate the queries. The study would then not be limited by the specific query languages chosen for this study.

In the end, there is still room for further work and improvement in the area.

A
Appendix - Language Grammars

This appendix contains a listing of the grammars written for the two language implementations used in this study. These grammars are written using the Python PyParsing library. Simplified versions of these grammars are shown respectively in figures 4.1 and 4.2 in the main text. This code is included here for completeness and as a reference.

```
g_quote = Literal("'").suppress()
g_text = Regex("[\w\s\:\#\.]+").setResultsName("text")
g_string = Optional(g_quote) + g_text + Optional(g_quote)
g_equ =  Literal("!=").setResultsName("connector") |
        Literal("=").setResultsName("connector")
g_amp = Literal("&").setResultsName("connector")
g_hat = Literal("^").setResultsName("connector")
g_or = Literal("|").suppress()
g_seq = Literal("->").setResultsName("connector")
g_hash = Literal("#").setResultsName("hash")
g_left_brack = Literal("[").suppress()
g_right_brack = Literal("]").suppress()

g_vals = Forward()
g_vals << g_string +
        ZeroOrMore(Group(g_or + g_vals).setResultsName("or_group"))

simpleq = Forward()
complexq = Forward()

exp = (simpleq | complexq).setResultsName("exp")
exp_basic = Group(Group(Optional(g_hash) +
        g_string).setResultsName("left") +
        g_equ + Group(g_vals).setResultsName("right"))
simpleq << (Group(exp_basic.setResultsName("left") + g_amp +
        simpleq.setResultsName("right")) | exp_basic)
complexq << (Group(g_left_brack + exp.setResultsName("left") + g_hat +
        exp.setResultsName("right") + g_right_brack) |
        Group(g_left_brack + exp.setResultsName("left") + g_seq +
        exp.setResultsName("right") + g_right_brack))
```

Figure A.1: The grammar that was implemented for a subset of the Emu query language, in Python using the PyParsing library.

```
p = Regex("[\w:]+").setResultsName("text")
attribute = Literal("@").suppress()
eq = Literal("=").suppress()
closure = (Literal("?") | Literal("*") |
        Literal("+")).setResultsName("closure")
g_left_brack = Literal("[").suppress()
g_right_brack = Literal("]").suppress()

test = Literal("^").setResultsName("modifier") + p |
        p + Literal("$").setResultsName("modifier") | p
axis = (Literal("\\\\*") | Literal("\\\\") |
        Literal("\\") | Literal(".") |
        Literal("//*") | Literal("//") |
        Literal("/") |
        Literal("-->") | Literal("<--") |
        Literal("->") | Literal("<-") |
        Literal("==>") | Literal("<==") |
        Literal("=>") | Literal("<=")).setResultsName("connector")

locpath = Forward()
steps = Forward()

fexpr = locpath.setResultsName("exp")

attr_test = Group(attribute + p.setResultsName("attr") + eq +
        p.setResultsName("attr_val"))
pred_opt = (fexpr.setResultsName("predicate") |
        attr_test.setResultsName("attr_test"))

nodetest = Group(test + Optional(g_left_brack + pred_opt +
        g_right_brack + Optional(closure)))
steps << ( Group(nodetest("left") + axis + steps("right")) |
        Group(test + Optional(g_left_brack + pred_opt +
        g_right_brack + Optional(closure))))
locpath << Group(axis + steps.setResultsName("right"))
```

FIGURE A.2: The grammar that was implemented for a subset of LPath+, in Python using the PyParsing library.

References

[1] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond* (Springer, 1999).

[2] D. D. Chamberlin and R. F. Boyce. *SEQUEL: A Structured English Query Language.* In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pp. 249–264 (ACM, New York, NY, USA, 1974). URL `http://doi.acm.org/10.1145/800296.811515`.

[3] C. Lai and S. Bird. *Querying linguistic trees.* Journal of Logic, Language and Information **19**(1), 53 (2010). URL `http://dx.doi.org/10.1007/s10849-009-9086-9`.

[4] A. van Deursen, P. Klint, and J. Visser. *Domain-specific languages: An annotated bibliography.* SIGPLAN Not. **35**(6), 26 (2000). URL `http://doi.acm.org/10.1145/352029.352035`.

[5] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (Prentice-Hall, 2000).

[6] M. Fowler and R. Parsons. *Domain-Specific Languages* (Addison-Wesley, 2011).

[7] M. Mernik, J. Heering, and A. M. Sloane. *When and how to develop domain-specific languages.* ACM Comput. Surv. **37**(4), 316 (2005). URL `http://doi.acm.org/10.1145/1118890.1118892`.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* `http://www.w3.org/TR/2008/REC-xml-20081126/` (2008). [Online; accessed 21-June-2015].

[9] F. Manola and E. Miller. *RDF Primer.* `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/` (2004). [Online; accessed 28-May-2014].

[10] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. *The XML Query Algebra.* `http://www.w3.org/TR/2001/WD-query-algebra-20010215/` (2001). [Online; accessed 17-May-2015].

[11] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language.* `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/` (2013). [Online; accessed 24-May-2014].

[12] F. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. *Rdf-gl: A sparql-based graphical query language for rdf.* In R. Chbeir, Y. Badr, A. Abraham, and A.-E. Hassanien, eds., *Emergent Web Intelligence: Advanced Information Retrieval*, Advanced Information and Knowledge Processing, pp. 87–116 (Springer London, 2010). URL `http://dx.doi.org/10.1007/978-1-84996-074-8_4`.

[13] K. Losemann and W. Martens. *The Complexity of Regular Expressions and Property Paths in SPARQL.* ACM Trans. Database Syst. **38**(4), 24:1 (2013). URL `http://doi.acm.org/10.1145/2494529`.

[14] C. Chiarcos, J. McCrae, P. Cimiano, and C. Fellbaum. *Towards open data for linguistics: Linguistic linked data.* In A. Oltramari, P. Vossen, L. Qin, and E. Hovy, eds., *New Trends of Research in Ontologies and Lexical Resources*, Theory and Applications of Natural Language Processing, pp. 7–25 (Springer Berlin Heidelberg, 2013). URL `http://dx.doi.org/10.1007/978-3-642-31782-8_2`.

[15] S. Cassidy. *Generalising XPath for Directed Graphs.* Presented at Extreme Markup Languages, Montreal, Canada (2003).

[16] S. Bird and M. Liberman. *A formal framework for linguistic annotation.* Speech Commun. **33**(1-2), 23 (2001). URL `http://dx.doi.org/10.1016/S0167-6393(00)00068-6`.

[17] C. Lai and S. Bird. *LPath+: A First-Order Complete Language for Linguistic Tree Query.* Proceedings of the 19th Pacific Asia Conference on Language, Information and Computation, pp. 1–12 (Institute of Linguistics, Academia Sinica, 2005). URL `http://aclweb.org/anthology/Y05-1001`.

[18] S. Bird, P. Buneman, and W. C. Tan. *Towards a query language for annotation graphs.* CoRR **cs.CL/0007023** (2000). URL `http://arxiv.org/abs/cs.CL/0007023`.

[19] N. Ide and K. Suderman. *Graf: A graph-based format for linguistic annotations.* In *Proceedings of the Linguistic Annotation Workshop*, LAW '07, pp. 1–8 (Association for Computational Linguistics, Stroudsburg, PA, USA, 2007). URL `http://dl.acm.org/citation.cfm?id=1642059.1642060`.

[20] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. *XQuery 3.0: An XML Query Language.* `http://www.w3.org/TR/2014/REC-xquery-30-20140408/` (2014). [Online; accessed 22-June-2015].

[21] A. Berglund, S. Boag, D. Chamberlin, M. F., M. Kay, J. Robie, and J. Siméon. *XML Path Language (XPath) 2.0 (Second Edition).* `http://www.w3.org/TR/2010/REC-xpath20-20101214/` (2010). [Online; accessed 24-May-2014].

[22] M. Kay. *XSL Transformations (XSLT) Version 2.0.* `http://www.w3.org/TR/2007/REC-xslt20-20070123/` (2007). [Online; accessed 23-June-2015].

[23] L. Libkin, W. Martens, and D. Vrgoč. *Querying graph databases with xpath.* In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pp. 129–140 (ACM, New York, NY, USA, 2013). URL `http://doi.acm.org/10.1145/2448496.2448513`.

[24] S. Cassidy. *XQuery as an Annotation Query Language: a Use Case Analysis.* In *Proceedings of the Third International Conference on Language Resources and Evaluation, Canary Islands* (2002).

[25] S. Cassidy, D. Estival, T. Jones, D. Burnham, and J. Burghold. *The alveo virtual laboratory: A web based repository api.* In N. C. C. Chair), K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, and S. Piperidis, eds., *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)* (European Language Resources Association (ELRA), Reykjavik, Iceland, 2014).

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

[27] S. Cassidy. *An RDF Realisation of LAF in the DADA Annotation Server.* Proceedings of ISA-5 (2010).

[28] S. Cassidy. *The Emu Speech Database System.* `http://emu.sourceforge.net/new_manual/index.html` (2004). [Online; accessed 1-September-2014].

[29] J. Harrington. *Phonetic Analysis of Speech Corpora* (Wiley-Blackwell, 2010).

[30] A. G. Mitchell and A. Delbridge. *The pronunciation of English in Australia / by A.G. Mitchell and A.Delbridge A. & R.* (1965). Syd.