A Formal Framework for Linguistic Tree Query

Catherine Lai

March 2006

Submitted in total fulfilment of the requirements
of the degree of Master of Science by Research

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia

Abstract

The analysis of human communication, in all its forms, increasingly depends on large collections of texts and transcribed recordings. These collections, or corpora, are often richly annotated with structural information. These data sets are extremely large so manual analysis is only successful up to a point. As such, significant effort has recently been invested in automatic techniques for extracting and analyzing these massive data sets. However, further progress on analytical tools is confronted by three major challenges. First, we need the right data model. Second, we need to understand the theoretical foundations of query languages on that data model. Finally, we need to know the expressive requirements for general purpose query language with respect to linguistics. This thesis has addressed all three of these issues.

Specifically, this thesis studies formalisms used by linguists and database theorists to describe tree structured data. Specifically, Propositional dynamic logic and monadic second-order logic. These formalisms have been used to reason about a number of tree querying languages and their applicability to the linguistic tree query problem. We identify a comprehensive set of linguistic tree query requirements and the level of expressiveness needed to implement them. The main result of this study is that the required level of expressiveness of linguistic tree query is that of the first-order predicate calculus over trees.

This formal approach has resulted in a convergence between two seemingly disparate fields of study. Further work in the intersection of linguistics and database theory should also pave the way for theoretically well-founded future work in this area. This, in turn, will lead to better tools for linguistic analysis and data management, and more comprehensive theories of of human language

Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the Masters except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is approximately 30,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed,

Catherine Lai
4th March 2006

Acknowledgements

I was very lucky to have Steven Bird as my research supervisor, as he introduced me to this fascinating field, and never lost his enthusiasm — even after scribbling on many pages of drafts. Marcus Kracht introduced me to the elegance of modal logic, and I have not looked back since! I would also like to thank Lesley Stirling for guiding me towards many of the deeper linguistic issues. Dave Penton was always ready to give his insightful views from across the cube, and even write horrendous XPath queries in the name of science.

As always, Andrew Clausen has been my technical and emotional support crew. The care I have had from the Clausen family has been incredible. Finally, I'd to thank my parents who fostered my thirst for knowledge and hunger for Tasmanian-Malaysian-Chinese home cooking.

Contents

List of Figures

Chapter 1

Introduction

Language often seems like a one-dimensional stream. Whether it is the written or the spoken word, we still need to deal with one word after the other. However, there are patterns in language that are best described in two (or more) dimensions. One of the basic representations of language structure is the syntax tree. The appearance of such constituent structure is not limited to syntax; phonology and semantics also depend on the way linguistic units combine.

Much of the effort of linguistic inquiry is spent developing theories on how language works. At its core, this means accounting for this apparent structure in language. These theories should be subject to empirical testing. The source of data used in these tests is often a corpus: a collection of natural language data as it is actually used. The content of linguistic corpora is highly diverse. For example, they may consist of published literature (eg. the Brown corpus) or specially elicited and transcribed speech (eg. the TIMIT corpus (Garofolo et al., 1986)). In many cases structural features are explicitly annotated on the underlying language data (e.g. the Penn Treebank). In recent times, corpora have found increasing acceptance as linguistic evidence. A large part of the reason this acceptance has not been universal is the lack of effective tools for accessing massive annotated corpora.

The strength of corpus based results depends on a number of factors. First, we must consider how well the corpus captures the linguistic phenomena being modelled; second, how accurate annotations are within a given annotation framework. Finally, how can occurrences of linguistic phenomena be accurately and efficiently extracted from such massive data sets. All of these issues are especially complicated when dealing with structurally annotated corpora. However, to a great extent solutions to the first two problems depend on a solution to the third problem. Unless we have an effective way of querying the contents of such massive data sets, there is practically no way to check the accuracy of annotation. Nor is it practical to determine where or how often linguistic phenomena without such a query tool.

This thesis investigates the problem of querying linguistically structured data as the most pressing problem facing corpus linguistics. Specifically, querying tree structured linguistic data (treebanks). The goal is to provide an intuitive query language that is a theoretically principled and (because linguists can not wait forever) efficient. Such a language also needs to be scalable to deal with the continually increasing size of treebanks. A solution to this problem needs an understanding of the linguistic requirements and current database theory. Both of these areas in turn draw on predicate and modal logics in considerations of query expressiveness and tractability. It is in this very formal arena that our story is set. However, before delving into possible solutions, we must understand the problem and its motivation in greater detail.

1.1 Corpora as Linguistic Evidence

Corpora form the backbone of statistical natural language processing. However, many arguments have been made against the use of corpus data as a replacement for traditional linguistic fact. Instead, native speaker intuitions have been the preferred form of evidence for linguistic theories.

The divide between these two sets of evidence has been driven by an ideological split over the goals of linguistics.[1] It is widely accepted that language use (*performance*) does not accurately reflect our internal knowledge of language (*competence*). Native speaker intuitions, supposedly unpolluted by performance factors, are considered to model competence. Corpus derived data clearly reflects performance. Derwing (1979) is one of many who have suggested that a greater focus on performance is necessary for linguistics to move forward as 'it is the language process - the language user's competence to perform - which is the object of ultimate interest in language study.' However, the debate is still open. Even if we accept that it is reasonable for linguists to consider performance data, there are still a number of issues to be addressed.

From a methodological point of view, a main advantage of corpora has been the repeatability of experiments and reproducibility of results. Even taking into account the presence of dynamic corpora (e.g. the world wide web), it is easy to make corpus based data into something static. For example, it is not difficult to take a electronic snapshot of portions of the web. This reproducibility is very hard to achieve when using intuitions (Cowart, 1997). In theory, this also means that different approaches should be should be comparable. In fact, this is the main focus of task-based conferences such as CoNLL.

Corpora have provided much evidence to contradict linguists' intuitions (Manning, 2003; Sampson, 1996). However, there are still cases where corpora provide no direct evidence at all. A finite sized corpus cannot capture every possible utterance of a language. On the one hand, this makes data sparsity a very real problem. On the other hand, the nature of corpora has changed dramatically in the last fifty years. Machine readability, larger data collections and the development of better statistical and computational tools have allowed corpora to counter the problem of sparsity to some degree.

The increase in the size and number of corpora has not necessarily meant increased usability. The most sophisticated statistical techniques are still based on frequency counts. These in turn depend on an ability to locate specific occurrences of a phenomena. The problem of searching within a corpus has become much more tractable as corpora have become machine readable. However, tools for searching electronic documents (e.g. unix `grep`) are not well geared toward linguistic inquiry. For example, many linguistic queries are to do with the overriding structure of language. This sort of structure cannot always be derived from only the underlying text at run-time. As a result, considerable time and effort is spent creating structurally annotated corpora.

1.2 Structural Annotation

Structural annotations appear in many different forms across corpora. Even annotations that purport to record similar phenomena can have very different low-level representations.

The Penn Treebank provides skeletal syntactic annotation to the Wall Street Journal (Marcus et al., 1994). Here, sentence constituents are marked with brackets. Predicate-argument structure is also annotated to some extent. This manifests as co-indexing, traces and empty

[1] Though this has mostly been a battle between syntacticians

```
(SBARQ (WHNP-1 What)
       (SQ is
           (NP-SBJ Tim)
           (VP eating
                (NP *T*-1)))
       ?)
```

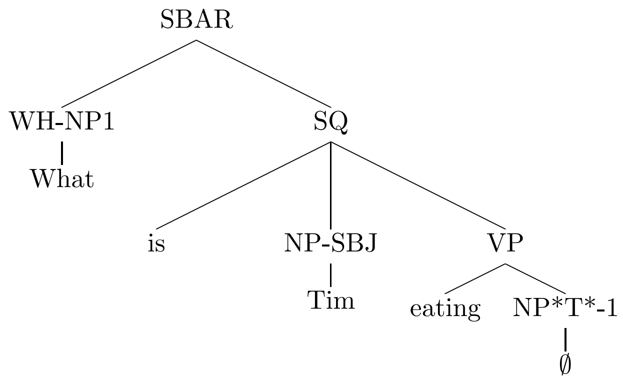


Figure 1.1: Penn Treebank: flat file format and tree structure

constituents. These annotations can be abstractly represented as an ordered tree with the underlying text at the leaf level (cf. Figure 1.1).

The TIGER corpus contains syntactically annotated text of the German newspaper Frankfurter Rundschau (Brants et al., 2002). Its annotation scheme is an extension of that used in the NEGRA corpus. Instead of bracketing, constituent structure is represented in a much more relational way (cf. Figure 1.2). This more relational structure allows the TIGER corpus to express non-tree properties. For example, crossing branches are permitted in order to deal with free-word order (scrambling) in some German sentences.

These corpora are not comparable when only considering flat file formats. However, they both clearly have tree-like structure as a result of their syntactic focus. But, not all annotations are best represented by trees. The TIMIT corpus (Garofolo et al., 1986) provides phonological annotations for spoken word samples. This is presented as annotated time series data. A tree representation of the data can be derived form the corpus directory structure. However, the sequentially and temporally dependent data finds a much more natural representation in annotation graphs (Bird and Liberman, 2001) (cf. Figure 1.3).

These graph representations give some idea of how such diverse structurally annotated corpora might be queried. They provide the logical level in the classic three level architecture for database systems. They also hint at some major problems for query language development. How can annotated corpora from different linguistic subfields can be used together? For example, linguists may wish to extract data relating syntactic structure with phonological structure (a la TIMIT). This could be represented as a set of intersecting trees (cf. Figure 1.4).

More generally, we need to be able describe the sequential and hierarchical nature of language in a concise, constrained way. The recursive nature of linguistic description, together with the temporal nature of linguistic events suggest ordered trees as a suitable representation. Trees represent hierarchal structure clearly while sequential structure is implicit. As the dominant structure in corpus linguistics, trees provide a baseline for expressiveness. Moreover, there is already a large existing literature on the formal properties of trees. This literature has been well motivated by the theoretical linguistics. In fact, a number of query languages for treebanks have already been proposed and implemented.

1.3 Tree Query Languages

Existing treebank query languages suffer from two major problems. First, these query languages are generally geared towards a particular corpus. This has led to ad hoc search strategies that are

```
<word>          <POS>   <morph>         <edge>   <parent>
...
kein            PIAT    Masc.Nom.Sg.*    NK       501
Arzt            NN      Masc.Nom.Sg.*    NK       501
anwesend        ADJD    Pos              PD       502
...
#501            NP      Masc.Nom.Sg.*    SB       502
...
```

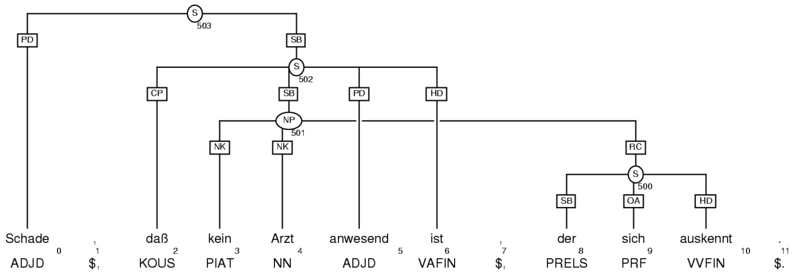


Figure 1.2: NEGRA file format and tree form

highly dependent on flat file representations. A variety of mutually-inconsistent approaches have resulted which are inapplicable outside a particular data set. Second, levels of expressiveness vary greatly. However, it is not clear that any of the current query languages have sufficient expressiveness to be a useful general purpose linguistic query tool. Finally, their relationship to existing database query languages is generally poorly understood. This makes it difficult to apply standard database indexing and query optimization techniques. As consequence they do not scale well.

However, there is increasing interest in the tree query problem in the database community. This has largely been the result of the proliferation of semistructured data on the world-wide web. XML has become an unofficial standard for storing structured data in a variety of fields, including linguistics. XML also has a very natural tree abstraction. As a result, querying tree structured data is currently an area of intense research. The focus of much of this research is the XPath query language. Understanding of the formal properties of XPath has allowed great improvements for efficient implementation of XML query tools.

Although XPath is a hot topic in tree query research, that does not necessarily make it a good choice for a linguistic tree query language. Linguistic data has different constraints than, say, biological data. However, it is clear that the linguistic query problem could benefit greatly from the advances made in understanding XPath's formal properties. Moreover, it is now clear that the study of trees in mathematical linguistics has something to offer both problems.

Model-theoretic syntax tackles the problem of describing linguistic structure. The aim of this program is to understand the structures licensed by a syntactic theory rather than the mechanisms generating them. This declarative approach to describing structure is exactly what is required of our query language. As we shall see, understanding the logics used in this field give us a good indication of the expressive requirements for a linguistic query language. Moreover, this work on linguistic trees has recently been incorporated into XML tree query research. This intersection point is fertile ground for developing an expressive and tractable linguistic tree query language.

```
train/dr1/fjsp0/sa1.wrd:
2360 5200 she
5200 9680 had
9680 11077 your
11077 16626 dark
16626 22179 suit
22179 24400 in
24400 30161 greasy
30161 36150 wash
36720 41839 water
41839 44680 all
44680 49066 year
```

```
train/dr1/fjsp0/sa1.phn:
0 2360 h#
2360 3720 sh
3720 5200 iy
5200 6160 hv
6160 8720 ae
8720 9680 dcl
9680 10173 y
10173 11077 axr
11077 12019 dcl
12019 12257 d
...
```

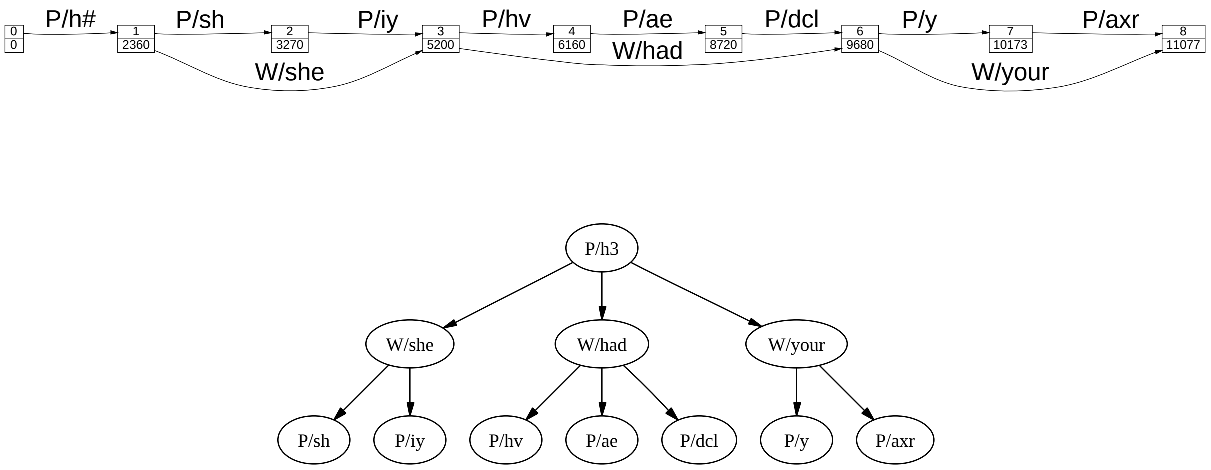


Figure 1.3: The TIMIT corpus: flat file, annotation graph and tree representations

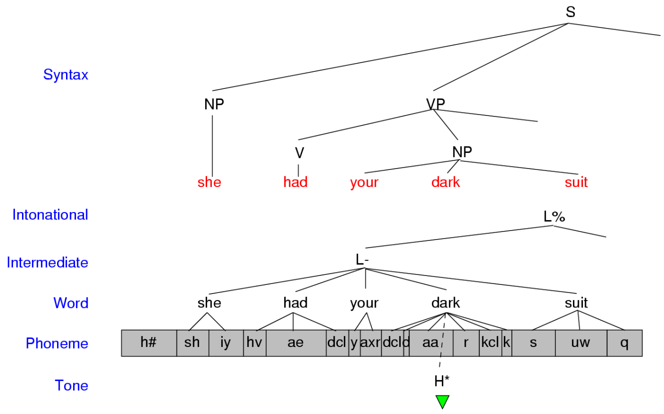


Figure 1.4: Intersecting Hierarchies

1.4 Thesis Overview

The main part of the tree query problem is finding a suitable language for describing the structures in question. Formalisms for describing tree shaped structures have been well studied in both linguistics and database theory. The tree query problem is a growing point of convergence for research in these two fields. This thesis uses work from both of these fields to develop a formal framework for linguistic tree query.

The first part of this is understanding what is actually required from such a language. To this end, Chapter 2 surveys current treebank query languages. Although these languages tend to lack a formal basis, they are actually geared towards querying linguistic structure. The functionality they exhibit provides a first point for understanding linguistic tree query requirements. This chapter also introduces a formal view on tree query. The varying abilities of the existing languages suggest that the expressiveness of first-order logic over trees is necessary. The main result of this survey is a set of requirements for linguistic tree query.

The main feature missing from the current tree query languages is the ability to specify complex transitive closures. On the other hand, formalisms that can express these closures have been studied in both model theoretic syntax and database theory. These are Propositional Dynamic Logic (PDL) and Monadic-Second Order Logic (MSO).

PDL provides a strong formal foundation for the path-based languages used to query XML and semistructured data in general. Chapter 3 reviews recent work characterizing the navigational component of XPath (Core XPath) in terms of PDL over trees. We also consider caterpillar expressions as another paradigm for querying trees that can be framed in terms of PDL. A number of extensions of Core XPath have also been identified that have counterparts in model-theoretic syntax. In particular, Conditional XPath is an extension of Core XPath with restricted closures that is first-order complete. We review linguistic evidence to suggest that this level of expressiveness is sufficient for a linguistic tree query language.

On the other hand, MSO has been proposed as the yardstick of expressiveness for tree query languages. MSO is also one of the cornerstones of model theoretic syntax. The use of MSO as tree query language is reviewed in Chapter 4. Although MSO is a highly expressive language, its combined query evaluation complexity is generally considered intractable. As a result, a number of query languages have been developed with the same level of expressiveness as MSO. As such, we examine monadic datalog in detail as a tractable, usable formalism that can describe the same structures as MSO.

Chapter 5 contains the main contributions of this thesis. In this chapter we investigate the XPath family, caterpillars and monadic datalog as linguistic tree query languages. This is done with respect to a new linguistic tree query language, LPath. LPath can be considered to be a linguistically oriented extension of Core XPath. We examine the ability of these languages to express LPath's linguistically motivated operators. In fact, it appears that the level of expressiveness required for the task lies on the weaker end of this spectrum of languages. All LPath operators and the closures suggested in the original requirements can be defined in Conditional XPath — but not always in an intuitive way. This motivates the use of explicit operators as they appear in LPath.

This study also provides the formal foundation for LPath that was previously lacking. We find that it is indeed a distinct new language in the XPath family with expressive capabilities between Core and Conditional XPath. However, LPath is not capable of defining non-primitive transitive closures. Conditional LPath, on the other hand, is a new extension of LPath that can. We find that Conditional LPath has the same expressiveness as Conditional XPath and hence is first-order complete. This provides further evidence that first-order logic has the required level of expressiveness and is natural fixpoint for a linguistic tree query language. Thus, Conditional LPath has the right level of expressiveness coupled with strong formal foundations. Moreover, Conditional LPath exhibits a good balance of expressiveness, tractability and linguistic usability. As such, we argue that Conditional LPath has all the properties required of a general purpose linguistic tree query language.

While Conditional LPath seems to be *the* tree query language, we have to recognize that the structures linguists deal with are not always strictly trees. Thus, there are a number of further requirements for a fully comprehensive linguistic query language. We conclude with a map of further work in Chapter 6.

Chapter 2

Linguistic Treebank Query Tools and Requirements

The major problem we face in developing a general purpose linguistic tree query language is balancing the level of expressiveness with computational tractability. This first means understanding the level of expressiveness needed for the task. The aim of this chapter is to formulate a set of requirements that will allow us to perform this balancing act properly.[1]

Current query languages have generally been developed for a particular corpus annotation scheme and flat file format. Annotated corpora, in turn, are usually to be developed with a particular task in mind. This means that query tools tend to be somewhat ad hoc in design and do not generalize well to other corpora. On the other hand, these query languages are built for specific purposes. This gives us a good indication of the types of queries a linguist would like to perform on a particular type of data. Thus, these 'naturally occurring' query languages provide primary data in our investigation of query requirements. To this end, this chapter presents a survey of existing linguistic tree query languages.

It is clear from this survey that a general purpose tree query language needs to be able to describe hierarchical and sequential relations between nodes. Moreover, tree query languages need to be able to handle negation and quantification in a sensible way. The differences between the surveyed languages become more transparent when viewed as fragments of the first-order predicate calculus. In fact, as we shall see, this first-order logic over trees appears to be expressive enough to satisfy the basic navigational requirements for linguistic tree query.

This chapter is organized as follows. Section 2.1 presents a survey of current query tools available for linguistic treebanks. Section 2.2 and Section 2.3 examine how trees and tree querying can be understood in a more formal light. Section 2.4 reviews first-order logic as a formal framework for comparing the the surveyed query languages. From this survey, a set of requirements for linguistic tree query is presented in Section 2.5. Finally, Section 2.6 maps out plans for tackling these requirements.

2.1 Linguistic Tree Query Languages

The prototypical hierarchical linguistic annotation is the syntax tree, an ordered tree where terminals (leaves) contain the text of a sentence being analyzed. Non-terminals represent syntactic categories, and the hierarchical organization represents constituency. The leaf level is usually considered immutable as it is derived from an external source such as a text collection.

[1]This chapter is a substantially revised and expanded version of the work published in Lai and Bird (2004)

Q1. Find sentences that include the word *saw*.
Q2. Find sentences that do not include the word *saw*.
Q3. Find noun phrases whose rightmost child is a noun.
Q4. Find verb phrases that contain a verb immediately followed by a noun phrase that is immediately followed by a prepositional phrase.
Q5. Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.
Q6. Find a noun phrase which dominates a word *dark* that is dominated by an intermediate phrase that bears an L-tone.
Q7. Find an noun phrase dominated by a verb phrase. Return the subtree dominated by that noun phrase only.

Figure 2.1: Syntactic Queries for Comparing Tree Query Languages

The queries in Figure 2.1 have been chosen to highlight the expressive capabilities of current tree query languages.

A query language must be able to accurately specify which subtrees to match in a corpus. This means quantifying the existence of nodes and succinctly stating the relationships between them. Q1 is a simple query based on the dominance relation inherent in trees. As mentioned earlier, however, the sequential ordering of nodes is also an important factor. Q3 and Q4 test how precisely subtrees can be described using dominance and precedence relations.

It is also desirable to specify subtrees by what they do not contain. This requires some form of negation. Q2 is a simple example of this type of query. Q5 contains an implicit negation: we need to select the common ancestor that has no descendant that is also a common ancestor of the nodes in question. These queries also explore the interaction between quantification and negation of nodes and subtrees.

Linguistic query languages, in general, need to be able to deal with heterogeneous data. Many interesting queries fall on the interface of linguistic fields. For example, Q6 requires both syntactic and phonological data. This can be represented as two trees that intersect at the word level (Cassidy and Harrington, 2001).

Finally, trees can be large, and it is often undesirable to return whole trees for which the query tree matches a tiny fragment. Thus, we need a way to specify what part of a query simply constrains context, and what part should be returned. Q7 is a simple test of a query language's ability to control output.

Note, in the following query example, the star ($*$) indicates queries that may produce incorrect output (false positives).

2.1.1 Penn Treebank and Tgrep2

Penn Treebank contains approximately 50,000 parse trees of Wall Street Journal text (Marcus et al., 1994). Each parse is represented as an ordered tree, and syntactic dependencies are indicated using zero-width *trace* elements co-indexed with a full noun phrase (cf. Figure 1.1).

Tgrep2 is a *grep*-like tool for this database (Rohde, 2001). All our example queries can be specified correctly in this language except Q6 (cf. Figure 2.2). Tgrep2 queries are nested expressions involving nodes and relationships between nodes. Nodes are specified using strings or regular expressions. Tgrep2 supports a large number of node relationships, including immediate precedence: `A` immediately precedes `B` (ie. `A . B`) if the right corner of `A` immediately precedes the left corner of `B` in the sentence on which the tree is built. A wildcard (`*`) can be used for a node when there is no constraint on its name. Node identifiers allow multiple sub-expressions

```
Q1.  S << saw
Q2.  S !<< saw
Q3.  NP <- N
Q4   VP=vp << (V . (NP >> =vp . PP >> =vp))
Q5.  *=p << (NP=n .. (VP=v >> =p !>> (* << =n >> =p)))
Q6.* Not expressible
Q7.  VP << ‘NP
```

Figure 2.2: Tgrep2 queries; Q5 is taken from Rohde (2001)

```
Q1.   #s:[cat="S"] & #l:[lex="saw"]
      & #s >* #l
Q2.*  #s:[cat="S"] & #l:[lex="saw"] & #s !>* #l
Q3.   #n1:[cat="NP"] & #n2:[pos="N"]
      & (#n1 >* #n2) & (#n1 >@r #n3) & (#n2 >* #n3)
Q4    #vp:[cat="VP"] & #v: [pos="V"]
      & #np:[cat="NP"] & #pp:[cat="PP"]
      & #vp >* #v & #vp >* #np
      & #vp >* #pp & #v >@r #vr
      & #np >@l #npl & #vr .1 #npl
      & #pp >@l #ppl & #npl .1 #ppl
Q5.   #vp:[cat="VP"] & #np:[cat="NP"]
      & (#x > #a) & (#x > #b) & (#a $.* #b)
      & (#a >* #np) & (#b >* #vp)
Q6.   Not expressible
Q7.*  #vp:[cat="VP"] & #np:[cat="NP"] & (#vp >* #np)
```

Figure 2.3: TigerSearch Queries

to refer to the same node (e.g. `=vp` in Q4). Tgrep2 can specify non-inclusion or non-existence (Q2, Q5). However, Tgrep2 is not complete for negation.

Query execution uses a binary file representation of the data, including an index on the words in the trees. The top node in a query is first matched and the rest of the tree is matched recursively, constrained by node relations. The output is a set of human-readable subtrees. The subtree matched by the first node mentioned in the query is returned by default, but a different subtree can be specified for output by the user as shown in Q7. This output cannot be queried, since it must first be converted into the binary format; ie. Tgrep2 is not compositional.

Not all node relationships need to be represented as a primitive. For example, the sibling relation (`$`) can be phrased in terms of parent (`<`) and precedence (`..`) relations. However, dominance type relations such as leftmost descendant (`<<,`) cannot be derived. The depth of such a descendant is unknown. Thus, the language could be greatly simplified with a closure operator.

```
Q1.  (E x (E y (& (cat x S) (tok y saw) (>> x y))))
Q2.  (E x (E y (& (cat x S) (tok y saw) (!(>> x y)))))
Q3.  (E x (E y (& (cat x NP) (cat y N) (> x y)
     (!(E z (& (> x z) (. y z)) )))))
Q4.  (E x (& (cat x VP) (E y (& (cat y V) (>> x y)
     (E z (& (cat z NP) (. y z) (>> x z)
     (E w (& (cat w PP) (. z w) (>> x w)))))))))
Q5.  (E x (E y (& (cat y NP)
     (E z (& (cat z VP) (.. y z) (>> x y) (>> x z)
     (!(E w (& (>> w y) (>> w z) (>> x w))))))))
Q6.  (E x (& (cat x NP) (E y (& (tok y dark) (>> x y )
     (E z (& (cat z L) (>> z y)))))))
Q7.* (E x (E y (& (cat x NP) (cat y VP) (>> x y))))
```

Figure 2.4: fsq queries

2.1.2 The TIGER Corpus and TIGERSearch

The TIGER corpus contains syntactically annotated German newspaper text (Brants et al., 2002). Syntactic analyses of German usually permit *discontinuous constituents*, represented in the corpus as trees with crossing edges. TIGERSearch is a logic and constraint programming approach to querying these so-called 'syntax graphs' (König and Lezius, 2001). Query graphs are built using two main relations: immediate dominance (`>`) and immediate precedence (`.`). Transitive, reflexive closures of these relations (`>*` and `.*` respectively) and other node relations such sibling (`$`) and left and right corners (`>@l` and `>@r` respectively) increase expressiveness. However, intersecting hierarchies are not supported (eg. Q6).

`A` precedes `B` (i.e. `A .. B`) if the left corner of `A` precedes the left corner of B. Immediate precedence means the distance between left corners is 1. Queries requiring immediate precedence (e.g. Q4) will not be correctly described using this relation. Left and right corners can be used to define the query we want (cf. Figure 2.3). However, this mimics the Tgrep2 precedence definition and may fail if the syntax graph has crossing branches.

Nodes are implicitly existentially quantified before the graph description. This means non-inclusion in Q2 will fail unless the negated node exists in the graph. In fact, there is no correct translation for any query requiring the non-existence of structure. The expression of Q5 uses the fact that if two nodes are descendents of two different siblings, then the parent of those siblings must be the first common ancestor. We can use the (somewhat unintuitive) fact that a rightmost child of a node must dominate the right corner of that node to formulate Q3.

The corpus of syntax graphs is indexed before querying. This index contains inferred facts about the corpus graphs with respect to TIGERSearch relations and predicates. In effect, the corpus data becomes a Prolog fact database. The query processor attempts to unify elements of corpus graphs with the query graph. Exhaustive search is avoided using label based filtering and by re-ordering the query graph traversal.

Matching syntax graphs are returned in their entirety. That is, TIGERSearch performs boolean queries. This means output reduction of Q7 cannot be expressed.

```
Q1.  [Syntax=S ^ Word=saw]
Q2.* [Syntax=S ^ Word !=saw]
Q3.  end(Syntax=NP, Syntax=N)=1
Q4.* [Syntax=VP ^ [Syntax=V -> [Syntax=NP -> Syntax PP] ]]
Q5.* [Syntax!=x ^ [Syntax=NP -> Syntax=VP]]
Q6.  [Syntax=NP ^ [Word=dark ^ intermediate=L-]]
Q7.? [Syntax=VP ^ #Syntax=NP]
```

Figure 2.5: Emu Queries

2.1.3 Finite Structure Query

Kepser (2003) presents the Finite Structure Query tool (fsq) for querying syntactically annotated corpora in the NEGRA format. The query language employed by fsq is full first-order logic. Kepser argues that this is necessary primarily to express negative descriptions of structure (as in Q2). As the name suggests, fsq is geared towards querying finite structures. That is, graph structures that may be disconnected, include crossing and secondary edges, or nodes with multiple parents. This allows us to represent Q6. Like TIGERSearch, fsq implements boolean querying so the subtree selection in Q7 cannot be expressed even though the required structure can be.

The sample queries (cf. Figure 2.4) demonstrate fsq's Lisp-like syntax. Atomic formulae are defined over (immediate) dominance, (immediate) precedence, secondary edges and node labels. Brackets provide a way of denoting atomic formulae. They also constrain the scope of variables. As seen in the examples, fsq queries are somewhat difficult to read due to the abundance of brackets. However, they make the job of query parsing easier.

The bracketing syntax of queries defines a tree which can be evaluated using structural recursion. A variable assignment is constructed during this recursion. Corpora need to be preprocessed into the appropriate binary format. fsq queries can be evaluated in time $O(|T|^k)$ where $|T|$ is the number of nodes in the tree and k is the quantifier depth. That is, PTIME.[2] Although some queries can be reformulated with low quantifier depth this is not always the case. For example, Q4 requires quantifier depth four.

2.1.4 Emu Query Language

The Emu speech database system (Cassidy and Harrington, 2001) defines an annotation scheme involving temporal constraints of precedence and overlap. Emu annotations are stratified into levels; each level is an interval structure, and elements on each level overlap those on the same and on other levels. The overlap relation is called dominance in the Emu documentation, but it is a reflexive, symmetric and non-transitive relation best understood as temporal overlap. Given this approach to dominance, nodes in an Emu structure can be "dominated" by multiple parent nodes. Hence Emu supports querying of multiple intersecting hierarchies. Emu translations for our example queries are given in Figure 2.5.

Built-in functions `start()`, `end()`, `mid()` and `position()` allow expression of the positional constraint in Q3. However lack of precision in dominance and precedence relations is a problem when dealing with syntax. Immediate dominance is only expressible between the appropriate levels. However, syntax trees do not easily split into such identifiable levels. Precedence is only

[2]For definitions, see for example Papadimitriou (2004).

```
Q1.  node:  S
     query: saw exists
Q2.  node:  S
     query: NOT(exists saw)
Q3.  node:  NP
     query: NP iDomsLast1 N
Q4.  node:  VP
     query: (V iprecedes NP) AND (NP iprecedes PP)
Q5.? node:  $ROOT
     query: ([1]NP precedes [2]VP)
            AND ([3]* dominates [1]NP)
            AND ([3]* dominates [2]VP)
            AND NOT(([4]* dominates [3]*)
                AND ([3]* dominates [1]NP)
                AND ([3]* dominates [2]VP))
Q6.* Not expressible
Q7.  node: VP
     query: exists NP
     node: NP
     query: exists *
```

Figure 2.6: CorpusSearch Queries

defined for nodes on the same level. There is no way to describe *immediate precedence* within a level. It is possible to express negation on node labels but we cannot express negation on nodes themselves. For example, `[Syntax=S ^ Word !=saw]` is evaluated to be true if there is some node on the word level that is not labelled `saw`. This means Q2 cannot be expressed. In fact, the `!=x` construct is used to specify wildcard where `x` does not appear in the set of node labels.

Each query only returns one node per match. The target node can be specified by the user (Q7). However, this is inadequate when we want to return whole subtrees. It also prevents query composition. Query constraints are tested in all possible match positions in the structure. This brute force method is satisfactory for small data sets but does not scale up well.

2.1.5 CorpusSearch

CorpusSearch was developed for the Penn-Helsinki Parsed Corpus of Middle English, though it can be used with any corpus annotated in the Penn Treebank style. CorpusSearch contains a large number of primitive relations, some very specific to linguistics. For example, the `Ccommands` relation is included. The `iDomsLast` relation directly expresses the positional constraint required for Q3. There are also a large number of options specifying output format. For example, CorpusSearch can be directed to print the complement of a query. That is, every tree that does not match.

An interesting feature of this language is that queries are limited in scope to subtree rooted by a specific type of node. In Q1, `exists` asks that the word ‘saw’ exists in a subtree rooted with an ‘S’. `$ROOT` denotes the root of any tree. The query language has two types of negation. The `!` operator applies to labels. Thus, `VP dominates !saw` describes a verb phrase that dominates something that is not the word `saw`. The `NOT` operator specifies structures not in a tree. CorpusSearch also allows prefix indices that force arguments to coincide. These are used in

```
Q1.  ($s syntax) ($w word):
       ($w@orth=="saw") && ($s@cat=="S") && ($s ^ $w)
Q2.  ($s syntax) (forall $w word):
       ($s@cat=="S") && (($s ^ $w) -> ($w@orth !="saw"))
Q3.  ($np cat) ($w word) :
       ($np@cat=="NP") && ($w@pos=="N") && ($np ^1[-1] $w)
Q4.  ($vp syntax) ($v word) ($np syntax) ($pp syntax):
       ($v@pos=="V") && ($np@cat=="NP")
       && ($pp@cat=="PP") && ($vp@cat=="VP")
       && ($v <>1 $np) && ($np <>1 $pp)
       && ($vp ^ $v) && ($vp ^ $np) && ($vp ^ $pp)
Q5.? ($vp syntax) ($np syntax) ($x syntax) (forall $y):
       ($vp@cat=="VP") && ($np@cat=="NP")
       && ($x ^ $vp) && ($x ^ $np) && ($np <> $vp)
       && (($x ^ $y) -> (!($y ^ $np) || !($y ^ $vp)))
Q6.  ($s syntax) ($i intermediate) ($w word):
       ($s@cat=="NP") && ($i@tone=="L-")
       && ($w@orth=="dark")
       && ($s ^ $w) && ($i ^ $w)
Q7.  (exists $vp syntax) ($np syntax):
       ($vp@cat=="VP") && ($np@cat=="NP") && ($vp ^ $np)
```

Figure 2.7: NiteQL Queries

Q5 to formulate the common ancestor query in same way as was shown for Tgrep2. However, the `NOT` operator has not been fully implemented and it is not clear from the documentation whether it may to be used this way. The query language was built specifically to deal with syntactic corpora so multiple domination, as in Q6, is not supported. CorpusSearch is compositional, allowing Q7 to be specified in two stages.

Overall, CorpusSearch displays a lot of linguistic functionality. However, the lack of formal foundation in the query language has resulted in some inconsistencies. For example, regular expressions over node labels serve as variable names. Thus `A iprecedes B*|C and B*|C iprecedes D` describes a sequence of three nodes, while `A iprecedes B*|C and C|B* iprecedes D` describes two unrelated sequences, each of two nodes.

2.1.6 NiteQL

NiteQL extends the MATE workbench query language Q4M (McKelvie et al., 2001) and has been released as part of the NITE XML Toolkit (Heid et al., 2004). Queries consist of weakly typed variable declarations followed by match conditions. Matches are evaluated over attribute, structure and time constraints. Precedence can be defined by the application designer depending on the data model.

The queries in Figure 2.7 assume the *immediate precedence* definition used by Tgrep2. If crossing branches are permitted and the left corners precedence definition is used, then NiteQL will behave like TIGERSearch instead. Dominance and precedence relations can take modifiers that provide more positional constraints. In Q3, `^1` indicates immediate dominance and `[-1]` indicates rightmost descendant. Similarly, `<>`n represents precedence with distance n. Thus, any sibling position at any level can be specified.

```
Q1.  /S[//saw]
Q2.  /S[not //saw]
Q3.  //NP{/N$}
Q4   //VP{//V -> NP -> PP}
Q5.  //_[/_[//NP] ==> _[//VP]]
Q6.* Not expressible
Q7.  //VP/NP
```

Figure 2.8: LPath Queries

Like TIGERSearch, variables are implicitly existentially quantified and negation is only explicitly defined on structural relations. Unlike TIGERSearch, NiteQL returns sets of nodes matching the declared variables rather than a tree. On the other hand, quantification (`exists`, `forall`) can be used in variable declarations. Their main purpose is to suppress query output (as shown in Q7), however quantification can be used with the implication connective to negatively describe structure. This is used in Q2, however, a similar approach to Q5 does not appear to work on the NiteQL web demo.[3]

The NITE project encourages storage in standoff XML. In this framework, hierarchical annotations point at the base data rather than being embedded in that data. An XPath-like pointer relation enables the formation of secondary (non-tree) edges between nodes. These can also occur between separate hierarchies. The final output is an XML document listing pointers to matches in the corpus. This is useful for finding specific sorts of nodes during the annotation process. NiteQL's type system allows queries on intersecting hierarchies as seen in Q6. Complex queries provide compositionality and can be used to structure results to some extent.

2.1.7 LPath

LPath is a path language for linguistic trees extending XPath with an *immediate precedence* axis and a scoping operator Bird et al. (2004). LPath queries are currently translated into SQL for evaluation. The LPath versions of the example queries are shown in Figure 2.8. Note, in Q5 we could take advantage of the XPath positional predicate `[1]` to form the query as follows `//_{//NP --> VP}/ ancestor-or-self::*[1]`. In this case, only the first node in *reverse document order* is selected. However, this would require the query language to treat node *sets* as *lists*.

2.1.8 Summary

This survey has revealed plethora of approaches to querying linguistically annotated trees. The surveyed languages showed a wild variation in query syntax. This ranged from queries based on variables and predicates to path-based, variable-free languages. Only fsq and TIGERSearch had a clear theoretical basis. This lack of formal foundation makes it difficult to predict how operators will behave (eg. negation). There are generally no results on the expressiveness or tractability of these languages. Thus, it is not clear from the outset just how expressive these languages are. Similarly, it is not clear that these languages will scale-up well when faced with complex queries on increasingly large data sets.

[3]http://zaunkoenig.ims.uni-stuttgart.de:65042/holger/webdemo/index.html

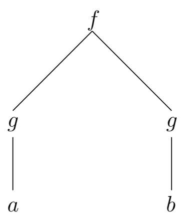


Figure 2.9: Tree from the term $f(g(a), g(b))$

Each language also differs in the number of primitive relations included. Only Tgrep2 and CorpusSearch are true tree query languages. All other languages consider some form of non-tree structure. On one end of the scale, CorpusSearch encodes the C-command relation (Chomsky, 1981). At the other, fsq can basically only includes (immediate) dominance and (immediate) precedence relations. Unsuprisingly, these relations appear to be the core of all of these languages. The most striking differences between these query languages are the use of variables and negation.

NiteQL, TIGERSearch and fsq explicitly declare variables and describe tree structure declaratively. That, the syntax of languages have a close relation to classical first-order logic. On the other hand, Tgrep2 queries describe paths augmented with variables when necesssary. A similar augmentation is used in CorpusSearch. The Emu query language and LPath are both variable free. This lack of variables affects how easy it is to describe that a node dominates many other nodes (Q5) or that conditions hold between nodes that are more specific than dominance or precedence (Q6). However, the introduction of variables and other node markers produces extremely verbose queries.

These languages differ in their ability to negatively describe structure. The inability to express negative constraints on structure in languages like TIGERSearch clearly results in a serious gap in expressiveness. This occurs because of constraints in the way existential quantification and negation can interact. Thus, both existential quantification and negation appear to be necessary in a linguistic tree query language. This suggests the expressiveness of first-order logic is needed.

In fact, we have seen that all the sample queries can be described in first-order logic via fsq. This is modulo the fact that fsq cannot select nodes, only whole trees. Moreover, the first-order framework means that fsq can make do with only a few primitives. In the following sections we consider trees as relational structures and investigate how first-order logic can be used to query them. This will also give us a more formal basis to compare these query tools.

2.2 Trees as Relational Structures

In order to talk about the structure of trees formally we need to define trees formally. The types of trees we need to consider are **unranked ordered node-labelled trees**. They are unranked in the sense that the number of children a node has is not determined by its label. In fact, each node can have any finite number of children. Order of siblings is important, hence we deal with ordered trees. This is due to the sequential nature of the underlying text that (usually) resides at leaf level.

One very well studied approach is to consider node labels to be operators of an algebra. Here, labels have a predefined arity. Trees are then defined as terms in that algebra (Gécseg and Steinby, 1997). For example, Figure 2.9 shows the graphical representation of the tree term

$f(g(a), g(b))$. There are two main reasons why this approach is inappropriate for the current task. Firstly, the fact that labels are assigned arities means that the trees are ranked. Secondly, the domain considered here is whole trees rather than nodes and their internal structure. This has been a very fruitful line of research for understanding tree languages and their relationship to automata and universal algebra. However, this is not useful for reasoning about individual nodes in a tree.

Another approach is to consider a tree to be a relational structure. Here, we consider our domain to be nodes of a tree and define relations and constants used to represent the internal structure of the tree. The vocabulary, or signature, of relations and the structure they can represent are defined formally as follows. This follows the presentation of Libkin (1998, Chapter 2)

Definition 2.1. *A **signature** τ is a non-empty set of constant symbols and relations with associated arities.*

Definition 2.2. *A τ**-structure** $\mathcal{T}$ is a domain T, a finite set, together with*

(i) An interpretation $R^{\mathcal{T}} \subseteq T^k$ for every relation R of arity k in τ;
(ii) An interpretation $c^{\mathcal{T}} \in T$ for every constant c in τ;
(iii) An interpretation $f^{\mathcal{T}} : T^k \to T$ for every function f of arity k in τ.

A τ-structure $\mathcal{T}$ is then defined as the following tuple.

$$\mathcal{T} = (T, (c_i^{\mathcal{T}})_{i \in I}, (R_j^{\mathcal{T}})_{j \in J})$$

The classic way of specifying unranked ordered trees is as a directed graph augmented with a $<$, a binary relation specifying sibling order. This gives rise to the following types of τ-structures:

$$\mathcal{T} = (T,\ \Sigma,\ E,\ <\ ).$$

Here T is the set of nodes, Σ is a finite set of node labels and E is the binary edge relation. It is important to note that this is not the only way to define trees. The following structure (Gottlob and Koch, 2004) does just as well,

$$\mathcal{T}' = (T,\ \text{root},\ \text{leaf},\ \Sigma,\ \text{firstchild},\ \text{nextsibling},\ \text{lastsibling})$$

In this case, all relations are unary except *firstchild* and *nextsibling* which are binary. Here, each binary relation can be thought of as a partial function. The difference between these two signatures is shown in Figure 2.10

The following definitions give us a convenient way of partitioning a tree.

Definition 2.3. *Let $\mathcal{T}$ be a τ-structure (tree) with domain T and node $v \in T$ let $\mathcal{T}_v$ denote the **subtree** of $\mathcal{T}$ rooted at v*

Definition 2.4. *For tree $\mathcal{T}$ with domain T and node $v \in T$, let the **envelope** v of $\overline{\mathcal{T}_v}$ denote $\mathcal{T}$ with the subtree $\mathcal{T}_v$ deleted (excluding node v).*

Given a signature we can obtain a language $\mathcal{L}$ for describing the tree structure. The type of formulae allowed (type of structure describable) depends on the syntax of the formalism. The semantics is, of course, highly dependent on the interpretations of relations. Given a tree shaped structure $\mathcal{T}$ with signature τ and a formula φ we wish to be able to say whether or not $\mathcal{T} \models \varphi$. That is, is it true that the tree has some structural property that is expressed by ϕ. If this is the case then $\mathcal{T}$ is a **model** of φ

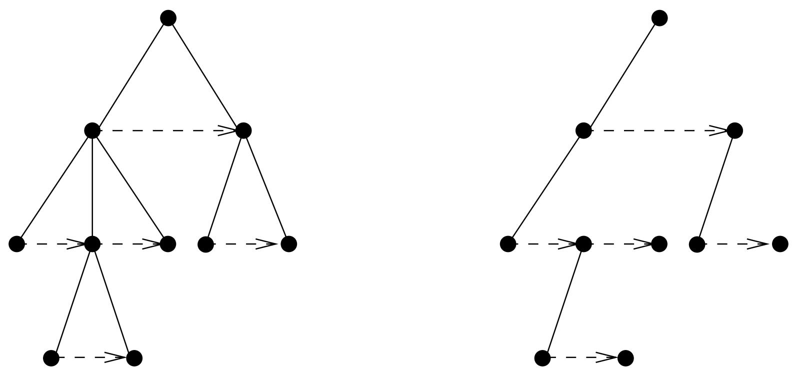


Figure 2.10: The same tree represented by two different signatures

Further constraints are still required to ensure that the structures we are dealing with are trees rather than graphs. For example, an unrestricted signature τ_u licenses structures $\mathcal{T}$ where there exists nodes $t, v \in T$ and firstchild(t, v) and nextsibling(t, v). We define the class of structures we want by stating these constraints as a set of axioms. Rogers (1994) describes an incomplete axiomatization of his theory of finite trees with bounded branching. A complete axiomatization is also provided by Backofen et al. (1995). However, Rogers argues that this incomplete version is sufficient because of the model-theoretic nature of the study. That is, non-tree (non-standard) models can safely be ignored if they never occur in the first place. This is clearly the case in a treebank.

Grädel (2005) notes that there are two main reasoning problems associated with any logic (hence logic for trees). The first of these problems is the **satisfiability** problem. The satisfiability problem asks whether, given a formula φ there exists any τ-structure $\mathcal{T}$, such that $\mathcal{T} \models \varphi$. An important computational problem is the **decidability** of satisfiability. That is, whether there is an algorithm that can determine in finite time whether a formula is satisfiable. However, this problem is not particularly relevant from the tree query point of view. In this context we are not so much interested in whether a tree exists as in finding the structures at hand that satisfy a formula. This is the model-checking problem, which we turn to next.

2.3 Model Checking

The model checking problem is very closely related to query evaluation. It basically asks a whether a τ-structure satisfies a particular formula.

Definition 2.5. *Consider a language $\mathcal{L}$ and a signature τ. Let ϕ be a sentence in $\mathcal{L}$, and $\mathcal{T}$ a structure in τ. The* ***model checking problem*** *for $\mathcal{L}$ and τ is the problem of checking whether $\mathcal{T} \models \phi$.*

The query takes the place of formula φ above. We can use the general idea of semantic entailment to formulate boolean or k-ary queries. The former considers the tree as a whole while the latter considers particular nodes. If the number of nodes in a tree is $|T|$, then k-ary queries can be formulated as $|T|^k$ model checking problems. We are generally interested in unary (1-ary) node-selecting queries.

The complexity of model checking (query evaluation) algorithms is usually considered in two parts, **data and query complexity**. These are the complexity with respect to the size of data and with respect to the size of the query respectively. In most cases, the size of the data far outweighs, the size of the query so only data complexity is considered. However, translations from a query language to the evaluation mechanism (such as a tree automata) can result in massive blow ups in the query size. Thus, it is important to keep track of combined complexity of model checking algorithms.

In general, considering the query evaluation problem in terms of model checking opens the door to rigorous and well studied tools for understanding and comparing the complexity of a query languages. We can implement a query tool by translating the query language into a well understood logic and then use algorithms suggested by complexity bound proofs. How these queries are actually formulated is specific to the logic being used. Models themselves take different forms in different logics. The surveyed tree query languages seem to be variants of first-order logic. As such, we clearly need to investigate classical first-order logic more closely as a tree query language. The syntax and semantics of first-order languages and the relationship to the survey languages is discussed in the following section.

2.4 First-Order Logic

In the language of **first-order predicate logic**, terms and formulae are built up over a signature τ inductively, and compound terms are created using the usual boolean connectives. A countably infinite set of variables may also be assumed. These variables are only quantified over elements in the structure's domain.

The survey in Section 2.1 contained a number of languages whose syntax looked a great deal like this language. Moreover, the survey suggested that the expressiveness of full first-order logic might be necessary to properly capture linguistic structures. This section formally defines the syntax and semantics the first-order predicate calculus. These definitions are taken from the presentation by Libkin (1998).[4]

Definition 2.6. *Let $x, y, z, \ldots$ be an infinite set of variables subscripts and superscript. The terms and formulae of the* ***first-order predicate calculus*** *over signature τ is inductively defined as follows.*

- *Each variable x is a term.*
- *Each constant symbol c is a term.*
- *If t_1 and t_2 are terms, then $t_1 = t_2$ is an (atomic) formula.*
- *If $t_1, \ldots, t_k$ are terms and P is a k-ary relation symbol, then $P(t_1, \ldots, t_k)$ is an (atomic) formula.*
- *If φ_1 and φ_2 are formulae, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi$ are formulae.*
- *If φ is a formula, the $\exists x \varphi$ and $\forall x \varphi$ are formulae.*

Free variables are variables that are not bound to any quantifier. Let $\vec{x} = x_1, \ldots, x_k$ be a tuple of free variables. Now, $\varphi(\vec{x})$ denotes a first-order formula with $\vec{x}$ free variables. Let $\vec{a} = a_1, \ldots, a_k \in M^k$. Now, $\varphi(\vec{a})$ represents the formula φ where each free variable x_i is bound to a domain element $a_i \in \vec{a}$. The model-theoretic interpretation, or semantics, of $\varphi(\vec{a})$ with respect to $\mathcal{M}$ is defined next.

Definition 2.7. *Let $\mathcal{M}$ be a τ-structure with domain M. Let $\varphi(\vec{x})$ be a formula and t be a term with free variables in $\vec{x} = (x_1, \ldots, x_n)$. in Let $\vec{a} = a_1, \ldots, a_n \in M^n$. The value of t, $t^{\mathcal{M}}$ and the notion of $\mathcal{M} \models \varphi(\vec{a})$ are defined inductively a follows.*

[4] We consider first-order predicate calculus without function symbols.

Tgrep2, CorpusSearch, fsq, LPath	$\exists x \neg \exists y\, (\text{descendant}(x, y) \wedge S(x) \wedge \text{saw}(y))$
NiteQL	$\exists x \forall y\, (S(x) \wedge (\text{descendant}(x, y) \rightarrow \neg\text{saw}(y)))$
TIGERSearch	$\exists x \exists y\, (\neg\, \text{descendant}(x, y) \wedge S(x) \wedge \text{saw}(y))$
Emu	$\exists x \exists y\, (\text{descendant}(x, y) \wedge S(x) \wedge \neg\text{saw}(y))$.

Figure 2.11: First-order representations of negation.

- *If t is a constant symbol c, then the value of t in $\mathcal{M}$ is $c^{\mathcal{M}}$.*
- *If t is a variable x_i, then the value of $t^{\mathcal{M}}(\vec{a})$ in $\mathcal{M}$ is a_i.*
- *If $\varphi \equiv (t_1 = t_2)$, then $\mathcal{M} \models \varphi(\vec{a})$ iff $t_1^{\mathcal{M}}(\vec{a}) = t_2^{\mathcal{M}}(\vec{a})$.*
- *if $\varphi \equiv P(t_1, \ldots, t_k)$, then $\mathcal{M} \models \varphi(\vec{a})$ iff $(t_1^{\mathcal{M}}(\vec{a}), \ldots, t_k^{\mathcal{M}}(\vec{a})) \in P^{\mathcal{M}}$.*
- *$\mathcal{M} \models \neg\varphi(\vec{a})$ iff $\mathcal{M} \not\models \varphi(\vec{a})$*
- *$\mathcal{M} \models \varphi_1(\vec{a}) \wedge \varphi_2(\vec{a})$ iff $\mathcal{M} \models \varphi_1(\vec{a})$ and $\mathcal{M} \models \varphi_2(\vec{a})$*
- *$\mathcal{M} \models \varphi_1(\vec{a}) \vee \varphi_2(\vec{a})$ iff $\mathcal{M} \models \varphi_1(\vec{a})$ or $\mathcal{M} \models \varphi_2(\vec{a})$.*
- *$\psi(\vec{x}) \equiv \exists y \varphi(y, \vec{x})$, then $\mathcal{M} \models \psi(\vec{a})$ iff $\mathcal{M} \models \varphi_1(a', \vec{a})$ for some $a' \in M$*
- *$\psi(\vec{x}) \equiv \forall y \varphi(y, \vec{x})$, then $\mathcal{M} \models \psi(\vec{a})$ iff $\mathcal{M} \models \varphi_1(a', \vec{a})$ for all $a' \in M$*

The satisfiability of first-order formulae is undecidable. The general model checking problem is PSPACE-complete. If the number of free variables in a formula is less than some fixed $k \geq 2$ then the model checking problem is PTIME-complete (Grädel, 2005).[5]

When using first-order logic to reason about unranked ordered trees we will generally use the following signature.

$$\tau_{fo} = (\Sigma, \text{descendant}, \text{following\_sibling}).$$

Σ, as usual, is the set of node labels. This signature describes the minimal set of relations needed to fully describe a tree's structure while allowing relations of unbounded distance. The first-order language over this signature is denoted FO_{tree}.

All of the query languages we have surveyed can be considered in terms of FO_{tree}. However, as previously noted, they differ in the extent they allow quantification and negation. Figure 2.11 shows first-order formulae demonstrating the different ways in which negation is interpreted in the surveyed languages. Fsq is the only language in which negation is unrestricted. In Tgrep2 and LPath negation is restricted to quantifiers. In NiteQL, negation applies to labels (unary predicates) and relations but not quantifiers. CorpusSearch provides two types of negation, one for labels and one that applies to quantifiers. However, the intended meaning of latter is somewhat unclear. Negation only applies to relations in TIGERSearch and only to labels in the Emu query language. It is evident that negation of quantifiers is necessary for linguistic tree query.

Each of these languages includes some form of the *descendant* and *following sibling* relations as or atomic formulae. Considering this and the need for negation and quantifiers, Kepser (2003) claim that the expressive power of full-first order logic is required, a claim that seems very plausible.

However, the sample queries were selected to demonstrate the differences between existing query languages. Of course, just because the query tools are limited in some way, does not

[5]Definitions of these standard terms from computational complexity can be found in, for example, Papadimitriou (2004).

Immediate dominance:
A dominates B, A may dominate other nodes

Positional constraint:
A dominates B, and B is the first (last) child of A

Positional constraint with respect to a label:
A dominates B, and B is the last B child of A

Multiple dominance:
A dominates both B and C, but the order of B and C is unspecified.

Sibling precedence:
A dominates both B and C, B precedes C; A dominates both B and C, B immediately precedes C

Complete description:
A dominates B and C, in that order, and nothing else

Multiple copies:
A dominates B and B, and the two Bs are different instances

Negation:
A does not dominate node with label B

Figure 2.12: Subtree Matching Queries

mean the needs of linguists are limited in the same way. Thus, the search for requirements is necessarily ongoing and open-ended. Nevertheless, requirements for a linguistic tree query language resulting from this survey and are considered in the next section.

2.5 Requirements for Linguistic Tree Query

Tree query languages need to be able to express node relationships succinctly. The languages we have surveyed needed more than dominance and precedence relations to express the specialized relations invoked in the example queries. However, it appears that most of these relations are really just syntactic sugar. These convenience relations indicate how tree query languages may be used by linguists. However, they do not tell us much about the expressiveness required by a linguistic tree query language in a formal sense.

This section outlines a set of requirements for linguistic tree query. This is really a first step in understanding the level of expressiveness require of a general purpose linguistic tree query language. We first review requirements for partially specifying simple structures. This can be thought of as ways of navigating through the tree to be queried. A reasonable definition of immediate precedence is clearly necessary. However, more is required than positive descriptions. Non-inclusion queries such as Q2 and Q5 could not be expressed in all languages. We will then look at some of the types of queries that could not be expressed in surveyed languages, namely transitive closures and multiple trees. Finally we consider the role linguistic tree query has to play in updating annotated corpora.

2.5.1 Navigation

Subtree Matching. A vital part of subtree matching is accurate specification of the query tree. An inventory of subtree description types is given in Figure 2.12. Most of this inventory deals with positively and partially specifying structures that we wish to find. Subtree description

(a) Flat Structure (b) Chomsky Adjunction

Figure 2.13: Two Representations for Optional, Repeatable Constituents

may also be facilitated with a graphical interface that maps tree diagrams to expressions in a query language. This is an attractive option for non-computer scientists and already exists in tools such as TIGERSearch. However, as we have seen in the previous sections, negative descriptions of structure are of vital importance and must not be neglected. Thus, we require a query *language*, not just a way of representing the partial objects we wish to match.

Returning subtrees. The query languages had some capacity to constrain the output of a query as was shown in example query Q7. This sort of precise reduction needs to be further supported. Moreover, NiteQL, LPath and the Emu query language were capable of choosing specific nodes (as opposed to subtree roots) to output. This ability to node-selecting is important if we wish to use a query language to produce new views or transform the data.

Reverse navigation. Query specifications tend to reflect top down, left to right tree navigation. On the other hand, context can occur in any direction from a node. This is a problem for languages such as Tgrep2 where graph description focuses on one particular node at a time. Reverse relations, such as 'follows', are implemented in Tgrep2 and are necessary for queries such as Q5. However, reverse relations have less value in a language like TIGERSearch where the ordering of graph description is not important. In terms of expressiveness, necessity of these relations depends on the query structure.

Non-tree navigation. Queries are not always described in terms of edge traversal structure. Queries are often specified over the sequence of terminals (i.e. the text), regardless of hierarchical organization. Example queries have shown that this notion of sequential navigation needs to be extended to non-terminal nodes. For example, to searching for sequences of one or more adjective followed by a noun, regardless of the internal organization of the noun phrase (cf. Figure 2.13). All surveyed languages support some notion of precedence, although not all allowed immediate precedence. More generally, immediate precedence can be thought of as relating a constituent to the one before it. It can be conceived in terms of the "proper analyses" of early generative grammar (Chomsky, 1963) or syntactic charts, as shown in Figure 2.14.

Subtree Scoping. In linguistics we often wish to talk about constituents. In particular, the types of structure that occur within certain phrasal categories. That is, we constrain the scope of queries to be within a particular subtree. This requirement is reflected in the obligatory `node` component of CorpusSearch queries.

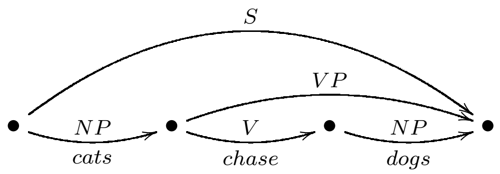
Scoping is also straightforward to describe in query languages that use variables, albeit rather verbosely. As seen in the examples, we can constrain the scope of a query to some subtree by stating that a particular node dominates all nodes in that structure. Subtree scoping is more difficult in variable free languages as many nodes may bear the same label. This, in part, motivates the addition of variables to path-based languages like Tgrep2. However, this tends to make queries appear excessively complicated. As such, LPath uses a scoping device to specify this constraint.

S → NP VP
VP → V NP
NP → cats
NP → mice
V → chase

(a) Phrase Structure Grammar

cats chase mice
NP chase mice
cats V mice
cats chase NP
NP V mice
NP chase NP
cats V NP
cats VP
NP VP
NP V NP
S

(b) Proper Analyses



(c) Syntactic Chart

Figure 2.14: Immediate Precedence in Linguistic Trees

2.5.2 Closures

The major requirement for linguistic tree query that evaded the languages surveyed is the ability to specify complex closures. The surveyed languages include closures of basic relations such as dominance, precedence and sibling precedence. These are necessary as the distances between nodes of interest can be arbitrarily large. These closures are generally represented as separate relations in tree querying languages. However, closures are required involving more complicated structures.

As a syntactic example, there is no bound on the distance between the head of a phrase (at leaf level) and the maximal projection of the head's category, in X-bar theory (Chomsky, 1981). Here the maximal projection tells us the category of the constituent. In another case, we may wish look for sequences of nodes on, for example, a tree's frontier that conform to a particular regular expression. This type of query has use in phonology in comparing groupings of syllables against the actual words they represent. The examples above and many others can be naturally frame in terms of transitive closure.

However, first-order languages can only express local properties (Courcelle, 1990). A corollary of this is that transitive reflexive closure of a relation cannot be expressed by a first order language (Libkin, 1998, Chapter 2). This is part of the motivation for including *descendant* and *following sibling* in signature of FO_{tree}. Note that, some complex closures can be expressed in FO_{tree}. For example, the following formulae defines pairs of nodes x and y such

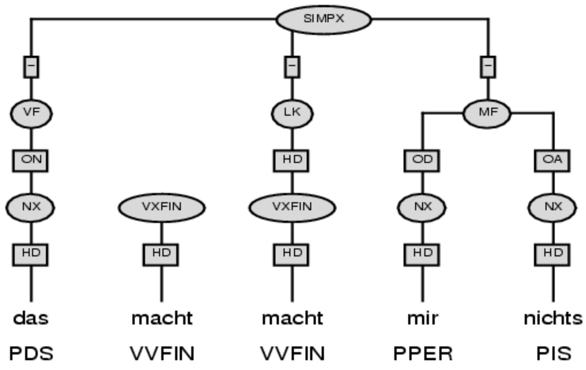


Figure 2.15: Forest representation of the Verbmobil corpus (Steiner and Kallmeyer, 2002).

that y is a descendant of x and every node between x and y of on that descendant line has property ϕ.

$$\varphi(x, y) \equiv (descendant(x, y) \land \phi(x) \\ \land \forall z\, (descendant(x, z) \land descendant(z, y) \rightarrow \phi(z)))$$

However, not all closures can be defined in FO_{tree}. While we can predefine a number of closed relations, we certainly cannot anticipate them all.

Consider the common situation where we wish to find parts of a corpus that fit a particular grammatical theory. For example, Chomsky adjunction in Figure 2.13 can be described by the productions: `NP` $\rightarrow$ `Adj NP`, `NP` $\rightarrow$ `N`. This could be translated to an LPath-like expression `(/NP[/Adj])*/N`. Here closure is atomic and each repetition involves a single step along some axis. In general we would like to be able to express any self-recursive rule $\texttt{A} \rightarrow \texttt{L}_1 \cdots \texttt{L}_n \texttt{A} \texttt{R}_1 \cdots \texttt{R}_n$ using a closure. In LPath this might possibly be specified by `(/A[<=Ln...<=L1, =>R1...=>Rn])+`. Similarly, outside of tree navigation we wish to find regular sequences such as consonants and vowels (Cassidy, 2002): `(->C)+(->V)+-(>C)+`.

2.5.3 Beyond ordered trees

Queries may need to extend beyond sentence boundaries. For example, anaphoric arguments may occur in previous sentences (Prasad et al., 2004). If trees represent sentences and querying is restricted to subtree matching this is a problem. One solution is to include multiple sentences in trees. However, this drastically increases the size of trees. Query trees are generally very small (if spread widely) so massive trees decrease filter effectiveness during query processing and have a bad effect on matching algorithms.

This constitutes a strong motivation for querying over ordered forests. In fact this is necessary when querying the Verbmobil treebanks of spontaneous speech (Hinrichs et al., 2000). Here, discourse turns are modelled to include repetitions, interjections, disfluencies and sentence fragments. These are represented as trees disconnected from surrounding well-formed sentences. Trees can occur wrapped in other trees as seen in Figure 2.15.[6]

[6]VIQTORYA is a query language developed for these treebanks (Steiner and Kallmeyer, 2002). However, this can be considered a subset of the TIGERSearch language so was not discussed in the survey

There is a general need to move beyond single tree searches and integrate different types of linguistic data. Querying intersecting hierarchies has been well motivated by the workbenches such as Emu and the NITE project. There is also a need to query over relational and structural data. (e.g. Switchboard Treebank (Graff and Bird, 2000)). We may want to match subtrees depending on the attributes of a word stored elsewhere (e.g. verb class in dictionary). Scope for these types of queries needs to be included in query language development.

Beyond this, there is a need to query non-tree structure. For example, Penn Treebank and the TIGER corpus include secondary edges. It would be useful to navigate these links to extract information about the long range phenomena stored there. This means a definite move from tree based models which needs to be explored further.

2.5.4 Update

Curating a corpus of trees requires frequent updates. Tree edits often describe restructuring of constituents. For example, transforming structure to (resp. from) a small clause representation involves insertion (resp. deletion) of a common parent. For example, changing annotation style to reflect X-bar theory may involve relabelling certain `NP` nodes to `NX'`. Another useful transform is to reattach a phrasal adjunct to a higher level node, which calls for a notion of subtree movement.

Insertion, deletion and relabelling nodes are standard tree editing operations. However, linguistic trees are more constrained than general trees. Freedom of movement of constituents almost always depends on preserving the base text. Subtree deletion is not allowed (except zero-width elements) nor is re-ordering of leaves. Any subtree can only legally move to a limited number of locations without perturbing the text.

Subtree movement can be described in terms of node insertion and deletion. However, this will be extremely tedious for the user to specify as subtrees may be extremely large. Thus subtree movement should appear as a basic operation. Cotton and Bird (2002) present a tree edit operations all in terms of movement of a distinguished node. The direction and surrounding structure determines where the node is reattached. Further operations are required to deal correctly with empty constituents. All update operations require inverses so edits can be reversed.

Syntactically annotated corpora are often annotated with respect to a particular grammar. These grammars may be updated and annotations need to be changed to reflect this. However, it is inefficient to manually reannotate the entire corpus every time this happens. A useful update mechanism should be able to compare grammars and then implement changes only where necessary. The closures described previously will be useful here.

2.6 The Quest for Closures

The tree query languages surveyed in this chapter varied in their ability to express the constructs of the first-order logic on trees, FO_{tree}. The major difference in these languages was in their ability to express negation. First-order logic provides a well understood formal framework in which to compare tree query languages. From this survey, it appears that the full power of quantification and negation of first-order logic is needed to accurately specify linguistic structures. In fact, first-order logic is expressive enough to describe all the basic navigational requirements set out in Section 2.5.1.

However, linguistic principles, such as projection in X-Bar theory, naturally call for complex closures. On the other hand none of the languages described is explicitly geared for specifying non-atomic transitive closures. It is unreasonable to expect that every such closure can be included as a primitive in a query language. Although it is possible to express some closures (such as projection) in FO_{tree}, the transitive closure of a arbitrary binary relation cannot be expressed in first-order logic (Fagin, 1974). Thus, the next major task is to find a formalism that has tractable query evaluation and can express the complex closures applicable to linguistics.

These unbounded relations have been studied in the context of model theoretic syntax. Model-theoretic syntax focuses on the structures licensed by grammars rather than the grammars themselves (Rogers, 1994). This declarative approach has proven a useful device for comparing theories from fundamentally dissimilar traditions of generative and unification based grammar. In fact, it highlights commonalities between the principles and parameters based approach of Government and Binding and the constraint-based Generalized Phrase Structure Grammar (Rogers, 1996). This separation of structure from theory and tradition is exactly what is required of a general purpose tree query language.

Two logics that can express transitive closures are Monadic Second-Order logic (MSO) and Propositional Dynamic Logic (PDL). Both of these logics also have a strong connection to database theory and are known to have linear time data complexity. However, these logics represent very different approaches to describing structure. MSO is an extension of first-order logic and so built on variables and predicates. PDL can be seen as an extension of modal logic and is variable free. These differences lead to substantial differences in expressiveness, evaluation complexity and generally how queries are formulated. PDL is a very path oriented language whereas MSO is of the classical declarative fold.

These formalisms and associated tree query languages are investigated in detail in the following two chapters, with a view to finding a suitably expressive yet tractable tree query language that fulfils all requirements set out above.

Chapter 3

Path-Based Tree Query

One of the main purposes of querying linguistic trees is to find out how linguistic features and structures are related. That is, what tree properties can be used to characterize a particular linguistic phenomena. This usually means describing structure connecting the linguistic constructions in question. Thus tree queries can often be constructed as paths between nodes, possibly with tests on the properties of nodes and edges on the way.

In fact, path-based approaches have been widely used to query semistructured data. This irregular, hierarchically structured type of data has a natural correspondence to trees. In the past few years the problem of querying semistructured data has become bound to the problem of querying XML documents. The demand for efficient tree query languages has grown with the increased use of XML on the web. Moreover, XML has found increasing use as a format for encoding linguistically annotated corpora (Carletta et al., 2004).

It makes sense to examine what light XML query languages can shed on the linguistic tree query problem. This chapter surveys two major paradigms for XML tree query, XPath (Clark and DeRose, 1999) and caterpillar expressions (Brüggemann-Klein and Wood, 1999). The survey is done as part of our quest to find an expressive and tractable formalism for querying linguistic data. As it turns out, both of XPath and caterpillar expressions have a straightforward relationship to the propositional dynamic logic (PDL) over trees used in model-theoretic syntax. Model-theoretic syntax focuses on declarative descriptions of syntactic structures over language divorced from the mechanisms they were generated from. Thus, this connection to PDL not only provides a number of results on the expressiveness and complexity of these languages, it also provides strong linguistic evidence about the expressiveness required for our task.

This chapter is organized as follows. Sections 3.1 - 3.5 investigate PDL and its connection to model-theoretic syntax. We next examine the connection between this logic and XML query languages. Semistructured data and XML are surveyed in Sections 3.6 and 3.7. XPath is introduced in Section 3.8 while Section 3.9 and Section 3.10 review recent work on the navigational component of XPath, (Core XPath) and proposed extensions (Conditional and Regular XPath). These languages correspond to fragments of PDL proposed in the model-theoretic syntax literature. To complete the survey, Sections 3.11 and 3.12 review caterpillar expressions, their use as a tree query language, and their connection to tree automata and PDL.

Overall, this survey suggests that Conditional XPath provides the right level of expressiveness for a linguistic tree query language while maintaining tractability. In order to understand why this is the case, we need to understand the basics of PDL and what it can express.

3.1 Propositional Dynamic Logic

Propositional Dynamic logic (PDL) is a powerful formal tool for reasoning about programs (Harel et al., 2002).[1] PDL can be considered as both a subsystem of first-order dynamic logic and an extension of propositional modal logic. In general, first-order dynamic logic allows reasoning about programs with a given domain of computation. This domain specifies the types of variable assignments allowed as well as operators and tests. For example, a domain of computation might be the integers with addition and equality. However, PDL abstracts reasoning away the domain of computation. Hence, reasoning in this formalism is about interactions between programs and propositions that are independent from any specific domain.

PDL formulae only consider programs and propositions. There is no notion of variable assignment. Compound programs are formed from atomic programs, atomic propositions and program constructs. In PDL, atomic programs are executed in a single step. Atomic propositions represent sets of states. The usual boolean connectives are available.

We can use PDL to reason about a tree's structure by defining programs that navigate through that tree. For example, the binary relation *child*. We can consider the program `child` with input node x node returns y if $(x, y) \in child$. Propositional constants then represent, for example, sets of nodes with a particular label (or feature).

The syntax of PDL formulae is defined as follows. Let $\Pi_0 = \{\pi_0, \pi_1, \dots\}$ be a set of **atomic programs** and $A = \{p_0, p_1, \dots\}$ a set of **atomic propositions**. Both of these sets may be countably infinite. The usual boolean connectives are allowed over propositions. PDL contains **set-theoretic union** ($\cup$), **relational composition** (;), and **reflexive transitive closure** (*) as program connectives. That is, PDL allows regular expressions over programs.

There are also two 'mixed' operators in PDL. The first is modal **necessity operator** []. If α is a program then $[\alpha]$ is a modal operator. $[\alpha]\phi$ defines a proposition that is interpreted as meaning that after executing α, ϕ must be true. The second operator is the propositional **test** ?. Given a proposition ϕ, $\phi?$ is a program that succeeds if ϕ is true for a given input (state) and fails otherwise. Adding the test allows us to make programs out of propositions. These mixed constructs mean that the definitions for propositions and programs are highly dependent on each other. Given these constructors, propositions ϕ and programs α are defined recursively:

$$\alpha := \pi \in \Pi_0 \mid \alpha; \alpha \mid \alpha \cup \alpha \mid \alpha^* \mid \phi?$$
$$\phi := p \in A \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid [\alpha]\phi$$

As usual, the boolean connectives $\vee, \rightarrow, \leftrightarrow$ can be constructed from $\wedge, \neg$. We also define the **possibility operator** $\langle\rangle$. This is the modal dual of the necessity operator []. So, $\langle\alpha\rangle\phi = \neg[\alpha]\neg\phi$. We also have the programs **skip** $:= \top?$ and **fail** $:= \bot?$. These are syntactic sugar for representing no-op and failing programs respectively.

PDL formulae are interpreted over Kripke frames.

Definition 3.1. *A* ***Kripke frame*** *is a pair $\mathcal{F} = \langle F, R\rangle$ where F is a (possibly empty) set and $R = \{R_i\}$, $i < k$ is a set of k binary relations.*

F is called the set of **possible worlds**. Each R_i is an **accessibility relation** associated with a modal (necessity) operator $\Box_i$. A Kripke frame $\mathcal{F}$ is **polymodal** if it contains more than one accessibility relation. Kripke frames are naturally visualized as edge-coloured directed graphs where the set of nodes is just the set of possible worlds, F. Edges are derived from accessibility relations. That is, if xR_iy then (x, y) is in the edge set. The colour of each edge is determined

[1] This presentation of PDL is adapted primarily from Harel et al. (2002) and Afanasiev et al. (2005)

by the the accessibility relation. This visualisation hints why Kripke frames are very natural formalization of trees (and graphs in general).

In PDL terms,we can think of $R : \Pi_0 \to F \times F$ as a mapping with $R(\pi)$ assigning a binary relation to each atomic program π. Moreover, we have a modal operator for every PDL definable program. We extend R to assign a binary relations to PDL programs, α, dynamically as follows:

$$
\begin{aligned}
R(\alpha;\alpha') &:= R(\alpha) \circ R(\alpha') \\
R(\alpha \cup \alpha') &:= R(\alpha) \cup R(\alpha') \\
R(\alpha^*) &:= R(\alpha)^* \text{ ie. the transitive reflexive closure of } R(\alpha) \\
R(\phi?) &:= \{(t,t) \,|\, \mathcal{M}, t \models \phi\}
\end{aligned}
$$

A **valuation** is a function $\beta : \Phi \to 2^F$ that maps propositions to sets of possible worlds. From a tree perspective, we can intuitively think of $\beta(p)$ to be the set of states in a Kripke frame labelled with p. Valuations allow us to define Kripke-models.

Definition 3.2. *A **Kripke model** is a triple $\langle \mathcal{F}, \beta, x \rangle$ where $\mathcal{F} = \langle F, R \rangle$ is a Kripke frame, β is a valuation function, and $x \in F$.*

Let α be a well formed PDL program. The semantics of PDL as follows:

$$
\begin{aligned}
&\langle \mathcal{F}, \beta, x \rangle \models \top \Leftrightarrow true \\
&\langle \mathcal{F}, \beta, x \rangle \models p \Leftrightarrow x \in \beta(p) \\
&\langle \mathcal{F}, \beta, x \rangle \models \neg\varphi \Leftrightarrow \langle \mathcal{F}, \beta, x \rangle \not\models \varphi \\
&\langle \mathcal{F}, \beta, x \rangle \models \varphi \wedge \psi \Leftrightarrow \langle \mathcal{F}, \beta, x \rangle \models \varphi \text{ and } \langle \mathcal{F}, \beta, x \rangle \models \psi \\
&\langle \mathcal{F}, \beta, x \rangle \models [\alpha]\varphi \Leftrightarrow \forall y \; xR(\alpha)y \to \langle \mathcal{F}, \beta, y \rangle \models \varphi
\end{aligned}
$$

Note that PDL is closed under negation. As usual, a PDL formula φ is **satisfiable** if there exists a Kripke model $\langle \mathcal{F}, \beta, x \rangle \models \varphi$. We write $\langle \mathcal{F}, \beta \rangle \models \varphi$ if φ is satisfied at every state $x \in F$.

These definitions provide the basic framework for using PDL. The next section reviews fragments of PDL actually proposed to reason about linguistic trees.

3.2 PDL and Trees

Various authors have proposed extensions of modal logics for work in model theoretic syntax. This approach to syntax calls for declarative descriptions of the structures licensed by syntactic theories. In this section we review three such languages that fall inside the PDL framework. These are $\mathcal{L}_K$ (Kracht, 1997), $\mathcal{L}_P$ (Palm, 1999) and $\mathcal{L}_B$ (Blackburn et al., 1996).

For each of these languages, the Kripke frames we are interested describe finite sibling ordered trees. That is, structures of the form $\mathcal{T} = \langle T, R_d, R_r \rangle$. Here T is a set of nodes and R_d, R_r are the daughter-of and right sibling relations respectively. We can derive the following binary relations:

$$
\begin{aligned}
R(d) &= R_d \\
R(r) &= R_r \\
R(u) &= R(d)^{-1} \\
R(l) &= R(r)^{-1}
\end{aligned}
$$

That is, we can consider four atomic programs (or modal operators):

$$\Pi_0 = \{u, d, l, r\}.$$

These programs correspond to up (parent), down (child) left and right (sibling) navigation respectively. We also have a set of atomic propositions A that we interpreted as node labels. We consider models $\mathcal{M} = \langle \mathcal{T}, \beta \rangle$ where the valuation function β essentially describes the distribution of node labels.

The differences between between each of $\mathcal{L}_K$, $\mathcal{L}_P$ and $\mathcal{L}_B$ can be seen as restrictions on the programs they allow. $\mathcal{L}_K$ is the most expressive language of the three (Kracht, 1997). It is basically PDL over the four atomic programs. The syntax and semantics of $\mathcal{L}_K$ is as described in the previous section.

$\mathcal{L}_P$, the propositional tense logic for trees presented by Palm (1999), is somewhat weaker than $\mathcal{L}_K$. The main difference is that this language does not allow all PDL program connectives. The permissible programs in $\mathcal{L}_P$ are:

$$\alpha := \pi \in \Pi_0 \mid \phi?; \pi \mid \alpha^*$$

Programs of the form $(\phi?; \pi)^*$ are called **conditional paths**. They require the proposition ϕ to hold at every point in a π-path. The restriction only really applies to the kleene star. Outside of the star, composition, union and the test can be described as follows:

$$\begin{aligned}
\langle \alpha; \alpha' \rangle \phi &\equiv \langle \alpha \rangle \langle \alpha' \rangle \phi \\
\langle \alpha \cup \alpha' \rangle &\equiv \langle \alpha \rangle \cup \langle \alpha' \rangle \phi \\
\langle \psi? \rangle \phi &\equiv \psi \wedge \phi
\end{aligned}$$

A less expressive logic, $\mathcal{L}_B$, has also been proposed by Blackburn et al. (1996). The language of $\mathcal{L}_B$ basically only allows the atomic programs described above and their reflexive transitive closures as programs. That is,

$$\alpha := \pi \in \Pi_0, \mid \alpha^*$$

This is equivalent to the star-free fragment of PDL with the following atomic programs:

$$\Pi_0 = \{u, d, l, r, u^*, d^*, l^*, r^*\}$$

In other words, $\mathcal{L}_B$ is a polymodal logic with eight modal operators. $\mathcal{L}_B$ can express relations between two nodes at an unbounded distance but cannot express constraints on the path between those nodes. This can certainly be expressed in $\mathcal{L}_P$ using conditional paths. On the other hand, $\mathcal{L}_P$ cannot express all of the properties that $\mathcal{L}_K$ can. For example, $(r; r)^*$ is a well-formed program in $\mathcal{L}_K$ that 'outputs' nodes an even number of right siblings across from the input node. However, all $\mathcal{L}_P$ formulae are translatable into FO_{tree} formulae (Palm, 1999). This is because atomic propositions, atomic programs in Π_0 and their transitive closures are expressible in FO_{tree}. For example, can express non-reflexive conditional paths as follows.

$$\begin{aligned}
(\phi?; d)^+ \equiv &\exists y\, (descendant(x, y) \wedge \phi(x) \\
&\wedge \forall z\, (descendant(x, z) \wedge descendant(z, y) \rightarrow \phi(z)))
\end{aligned}$$

On the other hand $(r; r)^*$ expresses a second-order property so this is inexpressible in $\mathcal{L}_P$.

In fact, $\mathcal{L}_P$ is strongly connected to temporal logic. Programs of the form $\langle(\phi?;\pi)^*\rangle\psi$ can be considered in terms of the temporal *until* operator. For propositions ϕ and ψ and program π the *until* operator can be defined as:

$$\mathcal{M}, t \models U_\pi(\psi, \phi) \Leftrightarrow \exists t' \text{ such that } \forall t'' \text{ with } tR(\pi^*)t''R(\pi)t', \\ \mathcal{M}, t'' \models \phi \text{ and } \mathcal{M}, t' \models \psi.$$

Blackburn et al. (2003) have proven that $\mathcal{L}_P$ is precisely as expressive as $\mathcal{L}_B$ with additional *until* operators for each atomic program in $\mathcal{L}_B$. *Until* operators cannot be expressed in basic modal languages (Blackburn et al., 2001). So $\mathcal{L}_B$ is strictly included in $\mathcal{L}_P$. The translation of $\mathcal{L}_P$ to FO_{tree} means that $\mathcal{L}_P$ is a strictly included in $\mathcal{L}_K$. In fact, Marx (2004a) has proven that an $\mathcal{L}_P$ equivalent formalism is first-order complete.[2] The hierarchy of expressiveness of these logics is as follows:

$$\mathcal{L}_B \subset \mathcal{L}_P \equiv FO_{tree} \subset \mathcal{L}_K$$

3.3 The Right Level of Expressiveness

Each of the logics presented in the preceding section was developed to describe linguistic phenomena. The question remains, which is the weakest logic capable of describing the structures we are interested in. This is especially important from tree query perspective as increased expressiveness generally means decreased tractability. While the debate is far from settled, a number of linguistic principles seem to motivated the use of $\mathcal{L}_P$. The argument for this centers around the use of transitive closures in linguistics.

The differences between the three languages described are characterized by different levels of restriction on the Kleene star. It is clear that some form of transitive closure (as represented by the Kleene star) is required. The need for such closures arises from a need to represent relations between nodes at an unbounded distance. Palm (1999) claims that conditional paths (restricted kleene star) most accurately describe linguistic use of transitive closures. He notes that descriptions of structural relations between two nodes usually require some constraints on path between those nodes.

An example of this is the projection principle (Chomsky, 1981). This says that every node has a unique projecting head. That is, every node x dominates a leaf y such that this leaf and all nodes on path between x and y share the same category. The path between y and x is a projection line. The concept of maximal nodes is important in the theory of Government and Binding. These are nodes that occur at the end of a projection line. Let φ represent a category. We write an $\mathcal{L}_P$ formula max_φ that defines maximal nodes as follows.

$$max_\varphi \equiv \langle(\varphi; d)^*\rangle(\varphi \wedge \neg\langle d\rangle\top) \wedge \neg\langle u\rangle\varphi$$

Palm argues that, in general, descriptions of long distance dependencies in the linguistics literature, such as anaphora and antecedent, are usually described in terms of the conditional paths in $\mathcal{L}_P$. As previously noted, conditional paths are not expressible in $\mathcal{L}_B$. This suggest that $\mathcal{L}_B$ is too weak for our needs.

Conditional paths are, of course, expressible in $\mathcal{L}_K$. However, Palm also notes that counting is possible in PDL and hence $\mathcal{L}_K$. On the other hand, Berwick and Weinberg (1984) have argued against this property in natural language. Evidence from morphology and phonology certainly seems to back this argument. For example, Hoeksema and Janda (1988) argue that

[2]This formalism was Conditional XPath which will be discussed later sections

prefixation (suffixation) only depends on the properties of leftmost (rightmost) constituents at a certain level in the tree. That is, we need to be able to distinguish the beginning or end of a constituent and the next constituent, but we do not need to be able to count higher than this. Unlike $\mathcal{L}_K$, FO_{tree}, hence $\mathcal{L}_P$, is considered to be non-counting. Since we are looking for the weakest logic that serves our needs, this suggests that we do not need more expressiveness than is provided by $\mathcal{L}_P$.

On the other hand, all three languages are contained in $\mathcal{L}^2_{K,P}$, the monadic second-order logic for variably branching trees.[3] Rogers (1994) has shown that $\mathcal{L}^2_{K,P}$ defines the derivation trees of of extended context-free grammars. That is, context-free grammars where regular expressions may occur on the right hand side of productions. The inclusion in $\mathcal{L}^2_{K,P}$ means that none of these languages can describe trees in which context-sensitive relations occur. For example, cross serial dependencies (scrambling) in Swiss-German (Vogel et al., 1996).

However, the ability of a language to express relations between nodes really depends on what features nodes are labelled with. For example, we could label each nodes that are maximal projections of a leaf. In fact, Palm (1999) has shown how the Kleene star can be removed from $\mathcal{L}_P$ formulae in general by extending the set of atomic propositions.

More generally, Kracht (1997) has introduced the notion of inessential features. These are features (labels) whose distribution can be derived from other features. This seems to imply some sort of redundancy. However, Kracht found that there are inessential features that eliminable from MSO but not from the weaker $\mathcal{L}_K$. In the same vein, Tiede (2005) has recently shown that tree languages that are undefinable in $\mathcal{L}_B$ but definable in $\mathcal{L}_P$ are definable in $\mathcal{L}_B$ with additional inessential features. The same relationship holds between $\mathcal{L}_P$ and $\mathcal{L}_K$. Tiede argues that a logic in where all inessential features can be eliminated may be too strong. That is, the weaker modal logics are preferable.

This evidence suggests that the right level of expressiveness can be found in $\mathcal{L}_P$. We now need to consider this logic in terms of query evaluation complexity. These complexity results along with a number of approaches for model checking PDL formulae are presented in the next section.

3.4 Complexity and Model Checking in PDL

The two main reasoning tasks associated with modal logics and their extensions are **satisfiability** and **model checking** (Grädel, 2001). Satisfiability is not of particular interest for this querying project. However, it is worthwhile noting some basic results. Satisfiability of PDL (ie. $\mathcal{L}_K$) is EXPTIME-complete (Harel et al., 2002). The same problem for the basic modal language (ie. $\mathcal{L}_B$) is PSPACE. Now, $\mathcal{L}_K$ is decidable with respect to satisfiability due to an embedding into the monadic second-order logic $\mathcal{L}^2_{K,P}$ (Rogers, 1994). Reasons for the robustness of extensions of modal logic with respect to decidability have been examined by Vardi (1996) and Grädel (2001).

What we are really interested in is how these logics fare in the model checking scenario. That is, for a PDL formula φ we want to know whether $\mathcal{T}, t \models \varphi$, for nodes t in a tree $\mathcal{T}$ (the Kripke structure). From a complexity point of view the outlook is quite good: PDL model checking is PTIME-complete (Lange, 2002). Moreover, PDL model checking has linear *combined* time complexity. That is, formulae (queries) can be evaluated in time linear in the size of the tree and linear in the size of the query.

Theorem 3.3. *(Alechina and Immerman, 2000) The model checking problem for PDL has linear combined time complexity.*

[3] Monadic second-order logic is discussed in more detail in the next chapter.

Alechina and Immerman (2000) describe the PDL model checker with linear *combined* complexity. This model checking algorithm works by inductively labelling the nodes of $\mathcal{T}$ with the subformulae of φ that they satisfy. This is very similar to the approach showing the linear combined complexity of model checking in the basic modal language (Vardi, 1996). The only difference is that PDL formulae allow regular expressions in modal operators.

Consider modal operator $\langle\alpha\rangle$ where α is a regular expression. In this case, α is equivalent to a NFA, N_α with $|N_\alpha| = O(\alpha)$. Now, consider the graph G, with vertex set $V = T \times S_\alpha$ where S_α is the set of state in N_α. There is an edge between $((v, s), (v', s'))$ iff $s' \rightarrow^a s$ in N_α and v is an a-successor, or v succeeds with test a. Now, each transition out of state $s \in S_\alpha$ takes us to a unique state $s' \in S_\alpha$. So each transition in N_α contributes at most T edges to G. This means, G has at most $|T| \times |N_\alpha|$ edges. We can then do a depth-first search on G from nodes of the form (v, s_f) to those of the form (v, s_i), where s_f, s_i are final and initial states of N_α respectively.

A very different approach is taken by Lange (2002) who presents a game-theoretic approach to model checking for PDL. The game involves two players, $\exists$ and $\forall$. Player $\exists$ tries to show that the formula φ holds at node t while Player $\forall$ tries to prove the opposite. Starting at t the players consider φ. Each move gives a new configuration consisting of a formula and a node. Player $\exists$ moves by picking a subformula of a disjunction she thinks can be proven true. Similarly, player $\forall$ tries to pick subformulae that may be false in conjunctions. Modal operators in subformulae cause transitions to other nodes in the tree. The size of formulae considered in each step decreases unless the Kleene star is involved. The game ends if an atomic formula is reached, the necessary transition from the current node, or an infinite loop is induced by the kleene star. Player $\exists$ wins if the current node t satisfies the atomic formula, the current formula is of the form $[\alpha]\phi$ and there is no α transition from the current node, there are infinitely many configurations of the form $t \models [\alpha^*]\phi$.

PDL can also be embedded into a number of logics with linear time combined complexity algorithms for model checking. This provides a number of 'plug and play' approaches to PDL model checking. For example, Alechina and Immerman (2000) also show that PDL has a linear time translation into their *Reachability Logic*. This is a fragment of two-variable first-order logic with transitive closure and booleans operators. This model checker uses a similar subformula labelling approach to the one described above. The regular expressions over programs are basically encoded using boolean variables.

Lange (2002) also notes that the PDL model checking game can also be derived from similar approaches to model checking for the modal μ-calculus. PDL can also be embedded into the alternation free modal μ-calculus, L^0_μ (Lange, 2002). A linear time algorithm for model checking L^0_μ has been presented by Cleaveland and Steffen (1993). Automata-based model checking for PDL can also be obtained from L^0_μ algorithms (Kupferman et al., 2000). A general discussion of model checking and μ-calculus can be found in Emerson (1996).

PDL admits a range of tractable options for model checker implementation. Clearly, these can be applied to PDL fragments such as $\mathcal{L}_P$ and $\mathcal{L}_B$. However, these two languages are contain in FO_{tree} and so have a more appropriate level of expressiveness. Thus, this framework puts us in good stead to implement a query evaluator over linguistic trees.

3.5 PDL as Tree Query Language

The preceding discussion of PDL has shown it to have a good balance between expressiveness and tractability. The path-based syntax of PDL is perspicuous and closures can be formulated easily. Query evaluation can be framed in terms of the model-checking problem. There are a number of linear combined time solutions to this problem described in the literature. Moreover, PDL has been used in linguistics as part of the model-theoretic syntax program. That is, it is a formalism for describing structure without any preconceived notions of the theory generating that structure. This is exactly what is needed in a linguistic tree query language.

The linguistics literature has also pinpointed the PDL fragment $\mathcal{L}_P$ as the suitable level of expressiveness for describing linguistic structures. $\mathcal{L}_P$ provides a way of expressing transitive closures seen in linguistic theories such as Government and Binding. The restrictions placed on the Kleene star in $\mathcal{L}_P$ also have linguistic motivation in the non-counting principle.

Overall PDL, more particularly $\mathcal{L}_P$, presents the theoretical foundation on which to develop an expressive linguistic tree query language. It happens that modal framework has been taken up, almost accidentally, in query languages for semistructured data. Semistructured data is tree shaped and so the query problem for this form of data has much in common with ours. The emergence of XML as a web interchange standard has been credited with renewing interest in the theory of unranked ordered trees (Libkin and Neven, 2003). The following sections review the two main paradigms for querying semistructured data, XPath and caterpillar expressions. In particular, we review how these paradigms fit into frameworks of PDL and first-order logic, and how well they suit the job of querying linguistic trees.

3.6 Semistructured Data and Trees

Interest in querying ranked unordered trees has also arisen from the database community. This has been driven by the increased use of semistructured data, predominantly on the web.

Unlike classical relational data, semistructured data is highly irregular. 'Semistructured data is data that is neither raw data, nor very strictly typed as in conventional database systems' (Abiteboul, 1997). This sort of data may include groups of similar concepts with different internal structures. For example, a movie may or may not have animators in the crew. The case is similar for syntactic annotation where annotators will not necessarily follow a strict prescribed grammar. This would result in missing entries in a relational table. Moreover, parts of the data may be unstructured. For example, there may be a plain text included. As such, this sort of data is not succinctly represented by ranked trees.

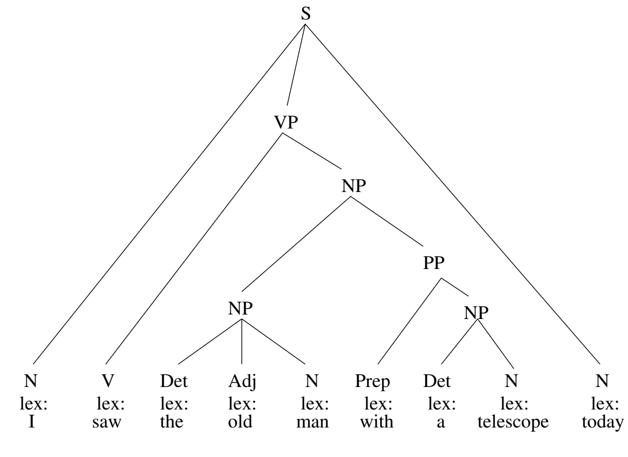
This incompleteness and irregularity means that a relational schema may be just as complicated as the data itself. Developers of query tools cannot expect a user to know an exact schema. Nor can they expect authors of such data to follow one. This is certainly the case for syntactically annotated corpora where any schema would be extremely large and complicated. Instead, the querying problem becomes one of partial specification. As a result of this, semistructured data is considered to be *self-describing*. The structures used to describe semistructured data are labelled directed graphs. As such, the semistructured data query problem can be seen as a more general case of our linguistic tree query problem. However, research in semistructured data has been overtaken by interest in the XML (Berners-Lee et al., 1994) and its associated query languages.

3.7 XML

XML was designed as universal exchange format for information on the web (Berners-Lee et al., 1994). Its design goals included human readability, ease of processing and terseness of markup. Like HTML, an XML file uses markup to describe document structure. In both cases, there are few restrictions the internal form of the document. The usual analogy is that HTML expresses presentation while XML expresses content. This makes XML a better format for querying the web for its content.

```
<S>
   <NP lex="I"/>
   <VP>
      <V lex ="saw" />
      <NP>
         <NP>
            <Det lex="the"/>
            <Adj lex ="old"/>
            <N lex="man"/>
         </NP>
         <PP>
            <Prep lex="with"/>
            <NP>
               <Det lex="a"/>
               <N lex="telescope"/>
            </NP>
         </PP>
      </NP>
   </VP>
   <NP>
      <N lex="today"/>
   </NP>
</S>
```

(a) XML encoding



(b) Tree structure

Figure 3.1: An XML document and its logical structure

Structure in an XML document is described by tags. Every start tag must have a corresponding end tag. Arbitrary nesting within tag pairs induces hierarchical structure. Figure 3.1 shows the logical form of an XML document as a tree. A start and end pair are represented as nodes. Leaves can be represented in the document with the empty tag. Unlike HTML, arbitrary tags may be defined. Start tags may include identifiers (IDREF). This allows representation of secondary, non-tree edges (references) between nodes.

Suciu (1998) notes that XML and semistructured data are not exactly the same thing. For one thing, XML is a syntax rather than a data model. On the one hand, semistructured data has traditionally been viewed as edge labelled, unordered trees Abiteboul et al. (1997) On the other hand, XML structure basically admits *ordered* node-labelled trees. This is a result of the textual basis of XML markup. This fits well with our need to keep sequential structure in linguistic data intact. XML maps easily to the bracketing structure in Penn Treebank with secondary edges. Thus, XML provides a close approximation to the linguistic structures previously discussed.

3.8 XPath

XPath (Clark and DeRose, 1999) is the core of XML query languages XQuery and XSLT. The primary job of XPath in these contexts is to select parts of an XML document. The XPath data model considers XML documents as sibling ordered unranked trees. Thus, XPath can be seen as a node-selecting, or unary, query language. XPath is navigational (path-based). The search for nodes starts from some point in the tree, usually the root, and proceeds through the tree via pre-defined axes. The axes specify the various ways one may 'step' through a tree. The axes we consider are *child*, *following sibling*, their closures and their inverses.

The most important type of XPath expression is a location path. This expression evaluates to the set of nodes that can be reached from a particular node (the context node) by the path. A location path consists of a series of location steps. Location steps also return a node set. The final node set returned by a location path is the result of the composition of location steps in the path.

Each location step is made up of an axis, a node-test and filter expression. Denote these as χ, t, fexpr respectively. Consider a location step applied to a node x, $\chi :: t[\text{fexpr}]$. The node-set selected by this step is the set of all χ-successors of x that has node property t and further structural properties defined by fexpr. Node tests can test a number of properties such as namespace and attribute values. However, we will mostly be interested in node labels which means checking tags. A filter expression can itself be a location path. A filter expression applied to node y returns true if the structure described in expression exists from the context of y. Filter expressions are like PDL model truth evaluations.

The above description is not the entirety of XPath 1.0. The nodes in the XPath data model we are interested in are *element nodes*. These are the nodes induced by start and end tags in an XML document. Besides this, the data model also has root, attribute, namespace, processing instruction, text and comment nodes to describe the rest of the information in an XML document. Note, attribute or namespace are modelled as children of element nodes but are accessed by special axes (ie. are not considered with the *child* axis). In its full glory, XPath 1.0 expressions may actually return one of four things: node sets, booleans, floating-point numbers, or strings. We may test nodes for attribute and namespace properties. We may also call functions from a pre-defined library in filter expressions.

The real expressive power of XPath is constrained by which axes, tests and filters are allowed and the interactions permitted between them. In a sense, the extra functionality distracts from analysis of this expressive power. Moreover, we wish to evaluate XPath in terms of the linguistic tree query problem. Now, XPath certainly was not designed as a linguistic tool. As such, it

```
locpath    :=   locstep | /locpath | locpath '/' locpath |
                locpath '|' locpath
locstep    :=   axis::test | axis::test[fexpr]...[fexpr]
fexpr      :=   locpath | fexpr 'and' fexpr |
                fexpr 'or' fexpr | 'not' fexpr | '(' fexpr ')'
axis       :=   ancestor | ancestor_or_self | parent | child |
                descendant | descendant_or_self | self |
                following | preceding |
                following_sibling | preceding_sibling |
test       :=   p | _
```

Figure 3.2: Syntax of Core XPath

is reasonable to aim to constrain our view of XPath to a minimal set of operations accounting for linguistic requirements. We can then include the extra pieces, if necessary, with a better understanding of their relationship to the more basic set of operations.

3.9 Core XPath and Extensions

Marx (2004b) presents a family of XPath languages that encompass for and extend the navigational functionality of XPath 1.0. **Core XPath** was originally presented by Gottlob et al. (2003). This language can be seen as XPath 1.0 stripped of non-navigational components such as attributes and namespaces. The basic axes available are exactly those included in the XPath 1.0 specification. **Conditional XPath** extends Core XPath primarily by adding a conditional axis. This expresses conditional paths (cf. Section 3.2) where every node in that path satisfies a particular condition represented as a filter expression. A further extension is **Regular XPath**. In this language axes can be defined as regular expressions over other axes. This allows nested Kleene closures.

The syntax of Core XPath is shown in Figure 3.2. Marx and de Rijke (2004) have noted that the *following* axis (respectively *preceding*) can be expressed in terms of the axes set including *child*, *descendant* and *following-sibling* and inverses.

Conditional XPath replaces the definition of `axis` in Figure 3.2 as follows:

```
axis      := primaxis | '[' fexpr ']' primaxis  |
               primaxis '[' fexpr ']' | axis '*'
primaxis  := self | child | parent |
               immediate_following_sibling |
               immediate_preceding_sibling
```

Primary axes (`primaxis`) represent the smallest steps that can be taken in each direction from nodes in a tree. Note, Core XPath does not include the one-step sibling axis *immediate following sibling* or its converse. However, it does include its transitive closures, *following sibling*. In fact, the transitive closures of each primary axis are included in Core XPath so we will use these axis names for non-conditional closures. For example, `child*` is equivalent to `descendant_or_self`. We also define `axis+` as the non-reflexive transitive closure of an axis as `/axis::_/(axis)*`. Again, we will denote `child+` as `descendant`.

$$
\begin{array}{lcl}
[\![\chi :: t]\!]_{\mathcal{T}} & = & \{(x,y) \mid x[\![\chi]\!]_{\mathcal{T}}y \text{ and } t(y)\} \\
[\![\chi :: t[\text{fexpr}]]\!]_{\mathcal{T}} & = & \{(x,y) \mid x[\![\chi]\!]_{\mathcal{T}}y \text{ and } t(y) \text{ and } \mathcal{E}_{\mathcal{T}}(y, \text{fexpr})\} \\
[\![/\text{locpath}]\!]_{\mathcal{T}} & = & \{(x,y) \mid (root, y) \in [\![\text{locpath}]\!]_{\mathcal{T}}\} \\
[\![\text{locpath}/\text{locpath}]\!]_{\mathcal{T}} & = & [\![\text{locpath}]\!]_{\mathcal{T}} \circ [\![\text{locpath}]\!]_{\mathcal{T}} \\
[\![\text{locpath}|\text{locpath}]\!]_{\mathcal{T}} & = & [\![\text{locpath}]\!]_{\mathcal{T}} \cup [\![\text{locpath}]\!]_{\mathcal{T}} \\
\\
\mathcal{E}_{\mathcal{T}}(x, \text{locpath}) & \Leftrightarrow & \exists y : (x,y) \in [\![\text{locpath}]\!]_{\mathcal{T}} \\
\mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1 \texttt{ and } \text{fexpr}_2) = true & \Leftrightarrow & \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1) = true \text{ and } \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1) = true \\
\mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1 \texttt{ or } \text{fexpr}_2) = true & \Leftrightarrow & \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1) = true \text{ or } \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}_1) = true \\
\mathcal{E}_{\mathcal{T}}(x, \texttt{not } \text{fexpr}) = true & \Leftrightarrow & \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}) = false \\
\\
[\![child]\!]_{\mathcal{T}} & := & R(child) \\
[\![right]\!]_{\mathcal{T}} & := & R(right) \\
[\![parent]\!]_{\mathcal{T}} & := & R(child)^{-1} \\
[\![left]\!]_{\mathcal{T}} & := & R(right)^{-1} \\
[\![self]\!]_{\mathcal{T}} & := & \{(x,y) \mid x = y\} \\
[\![p;q]\!]_{\mathcal{T}} & := & [\![p]\!]_{\mathcal{T}} \circ [\![q]\!]_{\mathcal{T}} \\
[\![p|q]\!]_{\mathcal{T}} & := & [\![p]\!]_{\mathcal{T}} \cup [\![q]\!]_{\mathcal{T}} \\
[\![p^*]\!]_{\mathcal{T}} & := & [\![self]\!]_{\mathcal{T}} \cup [\![p]\!]_{\mathcal{T}} \cup [\![p]\!]_{\mathcal{T}} \circ [\![p]\!]_{\mathcal{T}} \cup \ldots \\
[\![?\text{fexpr}]\!]_{\mathcal{T}} & := & \{(x,y) \mid x = y \text{ and } \mathcal{E}_{\mathcal{T}}(x, \text{fexpr}) = true\}
\end{array}
$$

Figure 3.3: The semantics of the Regular XPath. For the sake of brevity *right* corresponds to the *immediate following sibling axis* (*left* is similarly defined).

In addition to the `primaxis` definition Regular XPath further extends the definition of `axis` as follows:

```
axis      := primaxis | '?'fexpr | axis '|' axis |
             axis ';' axis | axis*
```

The semantics of Regular XPath is shown in Figure 3.3. This figure accounts for the other two languages as well.

Each of these languages has a counterpart in the three linguistically based PDL fragments presented in Section 3.2. Axes in this family of languages correspond to programs in the PDL based languages. Core XPath axes can be seen as basic modal operators. The conditional paths of $\mathcal{L}_P$ represent the correspond to the conditional axis in Conditional XPath. Regular XPath allows full use of the Kleene star over axes in the same manner as $\mathcal{L}_K$ does over programs. For the most part, the XPath variants appear almost identical to their PDL counterparts. However, PDL formulae are evaluated with respect to a distinguished node. Thus, they have the same semantics as XPath filter expressions rather than path expressions. On the other hand, the following result by Marx and de Rijke (2004) shows that these XPath languages are in fact expressively equivalent to their PDL counterparts.

Theorem 3.4. *(Marx and de Rijke, 2004) Every answer set definable in Core XPath is can be defined by an expression of the form* `/descendant::_[`A`]`.

This holds for because Core XPath axes are closed under converses. So we can safely select A above to be the converse of the original path expression. Conditional and Regular XPath are also closed under converses so this result holds for these languages as well. Now, Marx (2004b) gives translations from $\mathcal{L}_K$ to Regular XPath filter expressions and vice-versa. From this it is easy to see that Regular XPath is expressively equivalent to PDL. Similarly, Conditional

XPath is expressively equivalent $\mathcal{L}_P$. Core XPath is equivalent to $\mathcal{L}_B$ with atomic programs $\Pi_0 = \{u, d, l^+, r^+\}$ where l^+ corresponds to the *preceding sibling* axis (similarly for r^+) (Marx, 2004b).

This relationship with PDL shows that these XPath languages have two basic types, paths and filters. Filters define unary relations or **node sets**. As such Marx (2005b) calls these tests node well-formed formulae (node wff). Location steps of the form $/\chi :: t$[fexpr] can be written as $/\chi :: *$[self $:: t$ and fexpr] so the distinction between node tests and filter expressions is unnecessary. Location paths and steps can be understood as defining binary relations or **path sets**. These are expressed as path well-formed formulae (path wff). Conditional and Regular XPath define extensions to the syntax of path wff.

Following Marx (2005b), we can frame queries as path wff, θ, with filter expressions as node wff, η as follows. Let ? and $\langle\eta\rangle$ represent the tests and modal possibility operator outlined in Section 3.1. Let Σ be a countable set of node labels and let *primaxis* be the set of primary axes outlined above.

$$\theta := primaxis \,|\, primaxis^* \,|\, \eta? \,|\, \theta/\theta \,|\, \theta \cup \theta$$
$$\eta := p \in \Sigma \,|\, \top \,|\, \langle\theta\rangle \,|\, \neg\eta \,|\, \eta \wedge \eta$$

In Conditional XPath η remains the same, For θ,

$$\theta := primaxis \,|\, \eta? \,|\, \theta/\theta \,|\, \theta \cup \theta \,|\, (prim\_axis/\eta?)^+$$

For Regular XPath

$$\theta := primaxis \,|\, ?\eta \,|\, \theta;\theta \,|\, \theta \cup \theta \,|\, \theta^*$$

Although we now consider path sets (binary relations) instead of programs, this syntax highlights the strong connection between this XPath family of languages and PDL.

3.10 The XPath Family and First-Order Logic

This connection with modal logic and PDL has also lead to a characterization of Core and Conditional XPath in terms of the first-order logic. Let FO_{cxp} be the signature FO_{tree} extended with the *child* relation. The next theorem follows from the fact that Core XPath, like $\mathcal{L}_B$, can be considered a modal language.

Theorem 3.5. *(Marx and de Rijke, 2004) Every Core XPath node set is definable as a formula with one free variable over the signature* FO^2_{cxp}*.*

This follows now from the fact that the standard translation from modal first-order logic only requires two variables (Blackburn et al., 2001). From Theorem 3.4, answer sets are definable in terms of node sets. This means that this result holds for all Core XPath definable answer sets. Similarly, Core XPath path sets (path wff) are definable as FO_{cxp} formulae two free variables, $\varphi(x, y)$ say. We can define these formulae more precisely in terms of of conjunctive path queries.

Definition 3.6. *A* ***conjunctive path query*** *is a conjunctive query of the form:*

$$Q(x, y) := R_1, \ldots, R_n, \phi_1, \ldots, \phi_m,$$

where $R_1, \ldots, R_n$ *where each* $R_i \in \{child, descendant, following\_sibling\}$ *and* $\phi_1, \ldots, \phi_m$ *are formulae in* FO^2_{cxp} *with one free variable.*

These are similar to the conjunctive queries over trees considered in, for example, Gottlob et al. (2004) but allow negation in unary formulae ϕ_i. A **union of conjunctive queries** is a disjunction of conjunctive queries that share the same two free variables, x and y say. Marx and de Rijke (2004) has shown every Core XPath expressions can be translated into a union of conjunctive queries. In fact, these unions exactly characterize Core XPath expressions (these are unions of location paths).

Theorem 3.7. *(Marx and de Rijke, 2004) Every union of conjunctive queries has an equivalent Core XPath expression and vice versa.*

The restriction of answer sets to FO^2_{cxp} means that Core XPath is not first-order complete for FO_{tree}. That is, there exist FO_{tree} formulae in one free variable $\varphi(x)$ that do not have a Core XPath expression. However, Marx (2005b) has shown that Conditional XPath path sets are first-order complete.

Theorem 3.8. *(Marx, 2005b,* Theorem 2*)*

1. *Any expansion of Core XPath which is closed under complementation is first order complete for path sets.*
2. *Conditional XPath is closed under complementation, and so it is first order complete for path sets.*

That is, any FO_{tree} formula with two free variables can be expressed as a Conditional XPath path set. The semantics of complement of a binary relation R, $\overline{R}$, for a structure $\mathcal{T}$ is defined as follows.

$$[\![\overline{R}]\!]_{\mathcal{T}} = \{(n, n')|(n, n') \notin [\![R]\!]_{\mathcal{T}}\}.$$

The first part of this theorem is proved by first showing that every FO_{tree} is equivalent to an expression in Tarski's relation algebra (Tarski and Givant, 1987). This algebra is as expressive as FO_{tree} with two free variables. Each operator of Tarski's algebra can be expressed in Core XPath except complementation. So assuming complementation gives us the first part of the theorem. The second part uses the notion of separation to show that the complement of of any Conditional XPath path set can be expressed in Conditional XPath.

First-order completeness extends to Conditional XPath definable node sets. Let $\varphi(x)$ be a unary FO_{tree} expression. We can make this unary expression into a binary expression by adding a dummy variable. That is, $\phi(x, y) \equiv \varphi(x) \wedge x = y$. This means, the theorem above has the following corollary.

Corollary 3.9. *(Marx, 2005a) For every FO_{tree} formula $\varphi(x)$, there exists an equivalent Conditional XPath node wff.*

That is, Conditional XPath answer sets are also first-order complete.

The main result on the expressiveness of Regular XPath comes from its equivalence to $\mathcal{L}_K$. That is, Regular XPath is strictly more expressive than FO_{tree} and strictly less expressive than monadic-second order logic on trees. It also means that the exact relationship between Regular XPath and MSO equivalent formalisms is unknown. This is pressing problem facing the use of PDL for database querying is that there is no classical or automata-theoretic characterization of this logic.

On the other hand, we have already seen evidence that $\mathcal{L}_K$ is too expressive for describing linguistic tree structures (cf. Section 3.3). This discussion indicated that $\mathcal{L}_P$, and hence Conditional XPath, provides the right level of expressiveness for a linguistic tree query language.

The first-order characterization gives Conditional XPath a very solid theoretical foundation. Moreover, we know that query evaluation of Conditional XPath has linear time data and query complexity due to the model-checking result for PDL outlined in Section 3.4. This section also indicates a number of existing algorithms that could be used to perform this evaluation.

However, although we know Conditional XPath is first-order complete, it is not always easy to first-order formulae translate into path expressions. In terms of our linguistic requirements we need to be able to express certain notions of constituency. Subtrees in syntactically annotated trees can be viewed as constituents. There is often a need to distinguish first and last constituents in a phrase or to find the next constituent. Moreover, linguistic phenomena are often restricted to particular types of constituents and their contents. This is reflected in the scoping requirement discussed in Section 2.5.1. In order to express scoping we need expressions that have some way of remembering the subtree they should be in. However, path-based, variable-free languages like Conditional XPath have no explicit memory and so can only do this implicitly. These issues are further discussed in Chapter 5.

To complete this survey of path-based languages, however, we will next look at a the other major paradigm for querying trees, regular path queries (Abiteboul and Vianu, 1999). Caterpillar expressions (Brüggemann-Klein and Wood, 1999) are the representative formalism of this paradigm. Unlike the XPath family of languages surveyed above, caterpillar expression have a clear connection to automata theory connection. Specifically, tree walking automata. We will see that this formalism, too, fits into the PDL framework.

3.11 Caterpillars and Automata

A number of other path-based query languages for semistructured data have been proposed outside of the XPath mould. Most of these are based on the concept of regular path expressions (Abiteboul and Vianu, 1999). These include the UnQL (Buneman et al., 2000) and Lorel (Abiteboul et al., 1997) query languages. Both of these languages assume the edge-labelled model of semistructured data. Hence, regular path expressions are regular expressions over the edge label alphabet of a graph.

The spread of node-labelled XML has seen regular path expressions over edges turn into caterpillar expressions. Caterpillar expressions were originally developed by Brüggemann-Klein and Wood (1999) as a language for describing structured document layout. In particular, they were developed for use by typesetters with a non-technical background. Caterpillar expressions are defined as follows.

Definition 3.10. *Let / and $\cup$ denote string composition and union respectively. A* ***caterpillar expression*** *ϕ is a string of the following*

$$\begin{aligned} E &:= E/E \mid E \cup E \mid E^* \mid E^{-1} \mid (E) \mid axis \mid test \\ axis &:= \{\text{up}, \text{down}, \text{left}, \text{right}\} \\ test &:= \{\text{first, last, isFirst, isLast, isLeaf, isRoot}\} \cup \Sigma. \end{aligned}$$

Where Σ denotes a finite set of node labels.

Notice that caterpillar expressions are closed under converses ($^{-1}$). As in PDL, Caterpillar expressions originally described whether a node had a particular context. However, they now more generally used to define binary relations or path sets (for example, Gottlob and Koch (2002)). The meaning of a caterpillar expression as a binary relation is formally defined as follows.

Definition 3.11. *Let E be a caterpillar expression. The semantics of E is defined inductively according to the following rules.*

$$
\begin{aligned}
[\![E]\!] &:= \{(x,y) \mid xR(E)y, E \in axis\} \\
[\![E]\!] &:= \{(x,x) \mid x \in E, E \in test\} \\
[\![E_1/E_2]\!] &:= \{(x,z) \mid \exists y : (x,y) \in [\![E_1]\!] \wedge (y,z) \in [\![E_2]\!]\} \\
[\![E_1 \cup E_2]\!] &:= [\![E_1]\!] \cup [\![E_2]\!] \\
[\![E*]\!] &:= \textit{reflexive and transitive closure of} [\![E]\!] \\
[\![E^{-1}]\!] &:= \{(y,x) \mid (x,y) \in [\![E]\!]\}
\end{aligned}
$$

For example, $NP(/down/NX)^*/N$ is a caterpillar expression defines a paths that begin with an NP, end with an N and all intermediate nodes on the path are labelled with NX. This example shows the similarity between caterpillar expressions and Conditional and Regular XPath path sets.

Caterpillar expressions can be naturally represented using the PDL-like syntax for Regular XPath given in Section 3.9. Let $\Pi_0 = \{\text{up}, \text{down}, \text{left}, \text{right}\}$ be the set of atomic programs (primitive axes). Let $A = \{\text{first, last, isFirst, isLast, isLeaf, isRoot}\} \cup \Sigma$ be the set of atomic propositions (atomic tests). Caterpillar expressions are the following fragment of Regular XPath,

$$
\begin{aligned}
\theta &:= \pi \in \Pi_0 \mid ?\eta \mid \theta;\theta \mid \theta \cup \theta \mid \theta^* \\
\eta &:= a \in A
\end{aligned}
$$

As a fragment of Regular XPath, unary caterpillar expressions can be evaluated with linear combined time complexity.[4] However, caterpillar expressions do not fit neatly into the hierarchy of XPath languages considered so far.

On the one hand, caterpillar expressions can express second order properties such as $(child/child)^*$ that cannot generally be expressed in Core and Conditional XPath. On the other hand, caterpillars cannot represent FO_{tree} definable path sets. Gorris and Marx (2005) give the following example, $\varphi(x,x) = \exists z(up(x,z) \wedge P(z))$. This is expressed as `/self::*[parent::P]` in Core XPath. However, the closest caterpillar is $parent/P/child$. Since child is not necessarily a partial function this expression is only correct for unary trees. These differences appear to be caused by the lack of explicit negation and user defined tests or filter expressions in caterpillar expressions and, of course, the restriction of the star in Core and Conditional XPath.

Caterpillar expressions are only really designed to describe positive structure. There is no explicit negation. Non-existence of structure must be done - if it can be done at all - using enumeration techniques. In the simplest case we can may state that the current node does not have label P with a disjunction $\cup_{\sigma \in \Sigma - P} \sigma$. Larger structures are dealt with in the same complicated way.

This restricted syntax also mean that tests involving path wff cannot be explicitly expressed. Such tests can be thought of as branches in the query graph. Since a caterpillars cannot spawn new caterpillars, it must find its way back to the node where the trail branched during the traversal. The Core XPath example above shows that this is not possible in all cases. The inability to return to a specific node is an inherent problem in navigational languages without variable bindings. However, the combination of this and the lack of tests makes query writing more complicated than in languages with full tests such as Core XPath.

[4] Caterpillar expressions can be *unary* in the sense that, like XPath, they are generally consider to start at a specific node and select nodes reachable by the expression.

On the other hand, caterpillar expressions can be characterized in terms of Tree Walking Automata (TWA). TWAs are generalization of (two-way) string automata for unranked trees. This basically allows the automata head to move in the four basic unranked tree 'directions' identified as atomic programs in $\mathcal{L}_K$ (cf. Section 3.2). From a node in a particular state, a transition function determines a new state and whether the head moves to the node's parent, one of its children, the immediate left or immediate right sibling, or whether the head stays put. The following definition from Neven (2002) formalizes this concept.

Definition 3.12. *A **Tree Walking Automaton** (TWA) is a tuple $C = (Q, \Sigma, q_0, q_F, \delta)$ where,*

- *Q is a set of states,*
- *Σ a finite alphabet,*
- *q_0 and q_F are unique start and final states respectively, and*
- *$\delta : Q \times \Sigma \to Q \times \{up, down, left, right\}$ is a transition function.*

A TWA steps through a tree leaving a trail of state assignments to nodes. We can use these automata to evaluate node selecting queries by defining a set of selecting states. Nodes are selected if they are labelled with a selecting state at some point in the run of the TWA. Boolean queries are phrased as queries that select the root.

A number of important questions about the formal properties of tree walking automata have recently been answered. The following result is due to Bojanczyk and Colcombet (2005):

Theorem 3.13. *Tree walking automata do not recognize all regular tree languages.*

This means the expressiveness of tree walking automata is strictly less than that of monadic second-order logic (MSO) over trees. Unfortunately, like PDL, the exact expressiveness of TWAs with respect to MSO is still unknown.

Tree walking automata and caterpillar expressions define the same path sets so results on the expressiveness of the former apply to the latter. This equivalence gives this formalism a strong automata-theoretic grounding. It also highlights the differences between procedural and declarative descriptions of structure. (Neven and Schwentick, 2001) note that caterpillar expressions are to tree walking automata what regular expressions are to string automata. Like their string counterpart, caterpillar expressions are much easier to visualize and manipulate. However, the lack of explicit tests leaves something of a gap in what caterpillars can express. An extension of caterpillar expressions that, at least theoretically, covers this gap is discussed next.

3.12 Looping Caterpillars

Gorris and Marx (2005) extend caterpillar expressions with the loop operator. This formalism embeds both caterpillar expressions and Core XPath (in fact Conditional XPath). As might be expected, the loop operator forces a caterpillar expression to end where it began. In the new syntax we have:

$$E := E/E \,|\, E \cup E \,|\, E^* \,|\, E^{-1} \,|\, (E) \,|\, E^{loop} \,|\, axis \,|\, test$$

E^{loop} has the following interpretation in tree T:

$$[\![E^{loop}]\!]_T = [\![E]\!] \cap \{(x, x) \,|\, x \in T\}$$

Regular XPath	Looping caterpillars
$(/locpath)^c$	$root/locpath^c$
$(locpath/locpath)^c$	$locpath^c/locpath^c$
$(axis :: t[fexpr])^c$	$axis^c/t^c/(fexpr^c/all)^{loop}$
$(axis; axis)^c$	$axis^c/axis^c$
$(axis \cup axis)^c$	$axis^c \cup axis^c$
$(axis^*)^c$	$(axis^c)^*$
$(fexpr_1 \wedge fexpr_2)^c$	$(fexpr_1^c/all)^{loop}/(fexpr_2^c/all)^{loop}$
$(fexpr_1 \vee fexpr_2)^c$	$(fexpr_1^c/all)^{loop} \cup (fexpr_2^c/all)^{loop}$
$primaxis^c$	$primaxis$

Figure 3.4: Translation, c, from Positive Regular XPath to looping caterpillars. trees are by definition we can define the universal binary relation *all* connecting all pairs of nodes.

These looping caterpillars can define every FO_{tree} binary relation (Gorris and Marx, 2005) hence any Conditional XPath definable path set. This gives us theoretical means to express Conditional XPath filter expressions. The previously inexpressible example above becomes $(/up/P/child)^{loop}$. In fact, the unrestricted kleene star means that we may express the positive filter expressions of regular XPath.

Looping caterpillars also have an automata-theoretic counterpart. Loop tree walk automata (LTWA) are an extension of tree walking automata that characterize the expressiveness of looping caterpillars. These are pebble tree automata that cannot inspect their pebbles. Gorris and Marx (2005) has shown that looping caterpillars can be evaluated in polynomial time.

The nature of the inclusion between MSO and looping caterpillars is still unknown, as is its exact relationship with PDL. Every positive regular XPath expression has an equivalent looping caterpillar (cf. Figure 3.4). Now, negation of first-order formulae can be expressed although this is still implicit and rather unintuitive. However, it is unclear how negation might be extended to the second-order fragment of regular XPath.

Looping caterpillars provide a simple language in which both Core, Conditional XPath and caterpillar expressions can be embedded. It is also and algebraic language with a small number of operators that has a known automata-theoretic characterization. As in the case of Regular XPath, this level of expressiveness appears to be more than necessary for linguistic query. The most serious drawback, however, is the implicit nature of negation. Even though looping caterpillars contain FO_{tree} it is anything but simple to actually defined a negative description of structure. Nevertheless, looping caterpillars still present a rigorous foundation for exploring complex path expressions.

3.13 Path-Based Query Languages

Many linguistic queries can be naturally represented as paths relating features in a tree. A number of path-based tree query languages have been developed to query XML and semistructured data. Both XPath and caterpillar expressions are variable free language that fit very nicely into the modal framework of PDL. Moreover, both can be characterized in terms of types of closures (i.e. the Kleene star) and tests allowed. Caterpillar expressions can be viewed as PDL over trees where propositions are restricted to a predefined set of unary predicates. On the other hand, the navigational component of XPath 1.0, Core XPath, can be seen as a slight variation on star-free PDL. Core XPath does not have *immediate following sibling* as an axis (atomic program) but does have its transitive closure.

PDL fragments have also been used in model-theoretic syntax. In this chapter we reviewed three fragments $\mathcal{L}_K$, $\mathcal{L}_P$ and $\mathcal{L}_B$. These were developed as formalisms for reasoning about the structure of syntax trees. These fragments correspond to extensions of Core XPath recently proposed by Marx (2004b). Conditional and Regular XPath are expressively equivalent to $\mathcal{L}_P$ and $\mathcal{L}_K$ respectively. $\mathcal{L}_P$, hence Conditional XPath, restricts closures to those expressible by conditional paths. $\mathcal{L}_K$ (Regular XPath) allows full regular expression on PDL programs (axes). These correspondences provide a number of results on the evaluation complexity and expressiveness of these languages.

The main result on the tractability front is that the model-checking problem for PDL has linear combined time complexity. This result holds for each of the fragments discussed above. Thus PDL based query languages have very viable options for implementation. In terms of expressiveness, both Core and Conditional XPath can be characterized in terms of first-order logic on trees, FO_{tree}. Core XPath is equivalent to the two variable fragment. The main result reviewed, however, is that Conditional XPath is first-order complete.

In fact, it appears that first-order expressiveness is all that is needed to describe linguistic structures. A strong argument is presented in non-counting principle in linguistics. Counting, in general is not expressible in FO_{tree} but it is in more expressive languages like $\mathcal{L}_K$ (and to some extent caterpillar expressions). Moreover, conditional paths seem sufficient to express command relations of Government and Binding.

However, there are a few trade-offs to be made. Path-based and variable-free languages generally do not have any form of memory. This makes it hard to formulate queries that return to a particular node or to remain in a particular subtree. On the other hand, this is easy to achieve in classical predicate logic using variable bindings. Moreover, there are linguistic relations, such as scrambling, that are not expressible in MSO let alone FO_{tree}. As such, the following chapter investigates MSO as a tree query language.

Chapter 4

MSO as a Tree Query Language

Monadic Second-Order Logic (MSO) has been generally considered the benchmark for expressiveness for semistructured query languages . It has been suggested that MSO should play the same role for semistructured databases as first-order logic (FO) does for relational databases (Neven and Schwentick, 2001). One of the main reasons for this is that MSO can express transitive closures, a feature that FO lacks (Libkin, 1998). This allows the specification of complex path expressions required when querying semistructured data. As such, MSO has been the focus of much of the literature on querying semistructured data.

Existing research has, in turn, focused on unary MSO as a node-selecting query language. Semistructured query languages are generally used not only to select data, but to transform it. Semistructured queries are generally made up of two components; node selection and construction (Fernandez et al., 2000). A prime example of an embedded node selection language is XPath in both XSLT and XQuery. Thus, MSO has been proposed as a formal foundation for tree query languages like XPath.

MSO has also been used as a formalism for describing and analyzing linguistic structure. It has proven a useful tool for understanding structures generated by particular syntactic theories (Rogers, 1996) and for comparing extremely different theories. This basis in linguistics and in database theory has made MSO a natural candidate for a node-selecting linguistic tree query language.

A large part of why MSO has been so attractive to both areas is its strong and well-studied connections to formal language theory and, particularly, tree automata (Gécseg and Steinby, 1997). Tree automata (and extensions) provide a means of evaluating MSO queries in time linear in the size of the tree. However, the translation into tree automata requires non-elementary space. Moreover, MSO is not a particularly easy language in which to specify queries. While transitive closures can be expressed, they can only be defined indirectly. As a result, a number of distinct tree query formalisms exist that are expressively equivalent to unary MSO but differ in syntax and evaluation complexity. If we wish to implement a linguistic query language with the expressiveness of MSO, we need to consider these alternatives and their formal properties.

This chapter surveys work on MSO and equivalent formalisms as tools for tree query. Section 4.1 sets out the syntax and semantics of MSO, while Section 4.2 demonstrates how unary and boolean queries can be framed in this logic. The relationship between MSO and model-theoretic syntax is discussed in Section 4.3. Transitive closure is the main expressive capability of MSO applicable to both linguistics and databases. The construction of closures is reviewed in Section 4.4. Section 4.5 reviews the main tool for reasoning about the expressiveness of MSO, Ehrenfeucht-Fraissé games. Sections 4.6 to 4.8 discuss evaluation mechanisms for MSO formulae, in particular the classic connection between MSO and tree automata, evaluation complexity and how these results have been extended to unary MSO. Given the evaluation complexity considerations outlined in these sections, Section 4.9 surveys a number of MSO equivalent formalisms. From a usability evaluation complexity standpoint, the most promising formalism is monadic datalog (Gottlob and Koch, 2004). Theoretical results for this query formalism are investigated in Section 4.10. We conclude in Section 4.14.

4.1 MSO Syntax and Semantics

Monadic second order logic (MSO) extends first order logic (cf. Section 2.4) with set variables and quantifications over those variables. We can also express inclusion of a first order variable in a second order set (i.e. $x \in X$, usually denoted $X(x)$). As usual, free variables are defined as first and second-order variables that do not fall under the scope of a quantifier. Let $\varphi(\vec{x}, \vec{X})$ denote an MSO formula with $\vec{x} = x_1, \ldots, x_n$ free first-order variables and $\vec{X} = X_1, \ldots, X_m$ free second-order variables. As in the first-order case, MSO languages are defined via terms and formulae. The following definitions are taken from Libkin (1998, Chapter 7).

Definition 4.1. *Consider the signature τ consisting of relation symbols and constants. Let FO_τ be the first-order language over τ. The formulae of the MSO language MSO_τ are defined inductively as follows.*

- *Every first-order variable x and every constant symbol c is a first-order term.*
- *Every FO_τ atomic formula is an MSO_τ atomic formula.*
- *$X(t)$ is an MSO_τ atomic formula if t is a term and X is a set variable (second order variable of arity 1).*
- *MSO_τ formulae are closed under Boolean connectives and first-order quantification.*
- *If $\varphi(\vec{x}, Y, \vec{X})$ is a formula with Y a set variable, then $\exists Y \varphi(\vec{x}, Y, \vec{X})$ $\forall Y \varphi(\vec{x}, Y, \vec{X})$ are formulae.*

We can now define the semantics of MSO. This coincides with the semantics of FO (cf. Section 2.4) but with the additional rules as outlined in the following definition.

Definition 4.2. *Let $\mathcal{M}$ be a τ-structure with domain M. Also, let $\varphi(\vec{x}, \vec{X})$ be a MSO_τ formula with $\vec{x} = x_1, \ldots, x_k$ and $\vec{X} = X_1, \ldots, X_l$. Consider $\vec{b} = b_1, \ldots, b_k$, $b_i \in M$ for $1 \leq i \leq k$. Similarly, $\vec{B} = B_1, \ldots, B_l$, $B_i \subseteq M$ for $1 \leq i \leq l$. We determine whether $\varphi(\vec{b}, \vec{B})$ is true in $\mathcal{M}$, $\mathcal{M} \models \varphi(\vec{b}, \vec{B})$, as in first-order logic with the following additional rules.*

- *If $\varphi(x, X) = X(x)$ then $\mathcal{M} \models \varphi(b, B)$ if $b \in B$.*
- *If $\varphi(\vec{x}, Y, \vec{X}) = \exists Y \psi(\vec{x}, Y, \vec{X})$ then $\mathcal{M} \models \varphi(\vec{b}, \vec{B})$ if there exists $C \subseteq M$ with $\mathcal{M} \models \psi(\vec{b}, C, \vec{B})$*
- *$\varphi(\vec{x}, Y, \vec{X}) = \forall Y \psi(\vec{x}, Y, \vec{X})$ then $\mathcal{M} \models \varphi(\vec{b}, \vec{B})$ if for all $C \subseteq M$ with $\mathcal{M} \models \psi(\vec{b}, C, \vec{B})$*

Given these semantics, we can now define MSO queries.

4.2 Unary and Boolean Tree Queries in MSO

The main goal in describing structure using MSO is query formulation. This is again analyzed in terms of the classical model checking problem. That is, given a τ-structure $\mathcal{T}$ and MSO formula φ we want to know whether $\mathcal{T} \models \varphi$. The nature of queries in MSO then depends on the number of free variables contained in a formula.

Definition 4.3. *A **boolean MSO query** is a MSO formula with no free variables.*

Boolean queries test if a tree as a whole satisfies the properties expressed in the MSO formula. These queries determine the set of trees $\{t \mid t \models \varphi\}$ for some MSO formula φ. The following is an example query, φ, that defines all trees containing a VP labelled node that has an NP labelled node as its leftmost child.

$$\varphi \equiv \exists x \exists y(\text{firstchild}(x, y) \wedge \text{VP}(x) \wedge \text{NP}(y))$$

In order to select particular nodes we need to define unary queries.

Definition 4.4. *A **unary MSO query** is a MSO formula with one free first-order variable.*

A MSO formula φ with one free first-order variable x is denoted $\varphi(x)$. Given a tree t, a unary MSO query $\varphi(x)$ defines the set of nodes $\{x \in \text{dom}|t \models \varphi(x)\}$. The following query selects the VP labelled nodes that are parents of NP labelled nodes.

$$\varphi(x) = \exists y\,(\text{firstchild}(x, y) \wedge \text{VP}(x) \wedge \text{NP}(y)) \tag{4.1}$$

The ability to express unary (or node selecting) queries is important to the linguistic tree query problem for a number of reasons. First, linguists often want to view structures directly related to the query. That is, we are generally only interested in a very small portion of the tree being queried. We can reduce the amount of data output by restricting it to certain nodes or induced subtrees. Second, linguistic query languages have a clear application in updating and transforming treebanks. The ability to select specific nodes for transformation is very important. Moreover, the definition of unary query can be extended to k-ary MSO queries $\varphi(x_1, \ldots, x_k)$. Unary queries can be seen as first step towards formally understanding the problem of multiple node selection.

This gives us our basic framework for considering MSO as a general purpose tree query language. The motivation for using MSO to describe *linguistic* structure is discussed below.

4.3 MSO and Linguistics

The connection between MSO and linguistics is via model-theoretic syntax. Rogers (1994) has presented $\mathcal{L}^2_{K,P}$, a MSO language over trees. $\mathcal{L}^2_{K,P}$ is defined over the following signature:[1]

$$\sigma = (K, P, \text{child}, \text{descendant}, \text{preceding}, \text{equals}).$$

Here, K is a countable set of constants and P is a countable set of unary predicates. Thus, we can consider $\mathcal{L}^2_{K,P}$ to be parameterized. However, for our purposes $K = \emptyset$ and P will largely be determined by node labels occurring in the trees at hand.

In his dissertation, Rogers (1994) demonstrates how a substantial set of principles of Government and Binding theory (GB) for English can be expressed in $\mathcal{L}^2_{K,P}$. Moreover, $\mathcal{L}^2_{K,P}$ has been effectively used to comparing ideologically different formalisms. In Rogers (1996), $\mathcal{L}^2_{K,P}$ is used to analyze principles of Generalized Phrase-Structure Grammar (Gazdar et al., 1985). He notes that while explanations for phenomena, such as filler-gap relationships, can be very different, the structures licensed by these theories may not be. On the other hand, the formalization highlights a strong difference in how these two formalisms encode universals.

Rogers (1994) also uses $\mathcal{L}^2_{K,P}$ to develop a theory of language-theoretic complexity. Here, he gives a reduction of $\mathcal{L}^2_{K,P}$ to SωS, the MSO theory of the complete infinitely branching tree. The reduction has a number of corollaries. Firstly, the trees $\mathcal{L}^2_{K,P}$ describes are exactly those

[1] Note, Rogers uses different notation.

of strongly context-free languages, that is, the set of trees generated by context-free grammars where right hand side rules may be regular expressions.[2] Secondly, satisfiability of $\mathcal{L}^2_{K,P}$ is decidable. Both of these properties derive from the connection between these MSO and tree automata. This provides an interesting re-entry point for studying the generative capacity and complexity of language.

By capturing a large amount of Government and Binding theory in $\mathcal{L}^2_{K,P}$, Rogers uses these results to argue that English is context-free. However, this certainly does not extend to all other natural languages as seen in the case of scrambling in German. Investigations have also been made into how MSO can be used to describe and query context-sensitive relations such as scrambling. Scrambling can be specified in MSO. However, structures that do not satisfy the set of cross-serial dependencies described may satisfy the MSO formula. That is, there may be false positives. This problem is avoided in Mönnich et al. (2001) by lifting the MSO query and the grammar of a treebank to an algebra where mildly context sensitive relations can be coded. Once the appropriate trees have been selected, automata are used to project the lifted trees back into trees of the original treebank.

Kepser (2004) has proposed MSO as a general purpose linguistic query language for trees augmented with secondary edges. The proposed signature includes immediate dominance directed secondary relations. Surprisingly, no precedence relations are included as primitives. The main motivation for this is the precision with which linguistic structures can be described in MSO. In particular, transitive closures over MSO definable relations can be defined. Moreover, Kepser argues that the linear data complexity of over tree-like structures gives MSO the sought after balance of expressiveness and tractability.

These issues of expressiveness and evaluation complexity of MSO are discussed in more detail in the following sections. To understand the evaluation complexity problem, we need to understand the relationship between boolean MSO and tree automata and how these results can be extended to unary MSO. However, first we review the definition of transitivity in MSO as well as the basic tools for determining expressiveness results, Ehrenfeucht-Fraissé games.

4.4 Transitive Closures in MSO

The main limitation of first-order logic (FO) as a tree query language is that it can only express local properties of graphs (Fagin, 1974). However, the small extension MSO makes into second-order logic permits transitive (and reflexive) closures to be expressed. This means we can consider signatures containing only one step relations such as *child*. We may then define unbounded relations, such as *descendant*, as syntactic sugar.

Lemma 4.5. *(Courcelle, 1990) If a binary relation is definable in monadic second order logic, then so is its reflexive and transitive closure.*

Consider a τ-tree $\mathcal{T}$ with domain T. Let φ be a binary relation we wish to find the transitive (reflexive) closure of. We can construct an MSO formula that expresses this using the notion of an φ-closed set.

Definition 4.6. *Let X be a subset of T and let φ be a binary relation on $\mathcal{T}$. X is φ-closed if and only if*

$$\forall t \forall u \left((X(t) \land \varphi(t,u)) \rightarrow X(u) \right).$$

[2]A $\mathcal{L}^2_{K,P}$-definable set of trees is the projection of a set of trees generated by a finite set of context-free string grammars.

A φ-closed set X is minimal if it does not contain any other φ-closed set. We wish to define the transitive reflexive closure of φ as the binary relation φ^*. Now, $(x, y) \in \varphi^*$ if both x and y are in the same minimal φ-closed subset of T. We can express this, somewhat indirectly, in MSO as follows (Neven, 2002),

$$\varphi^*(x, y) \equiv \forall X((X(x) \wedge \forall z \forall z'(X(z) \wedge \varphi(z, z') \rightarrow X(z'))) \rightarrow X(y)).$$

If *child* is defined in MSO we may substitute it for φ and *descendant* for φ^* to define the *descendant* relationship. We can also express more complicated closures. Consider the following recursive phrase structure rule:

$$\begin{aligned} &\text{NP} \rightarrow \text{Adj NP} \\ &\text{NP} \rightarrow \text{N} \end{aligned}$$

Consider the signature that includes *lastsibling* and node labels as unary predicates as well as *firstchild* and *nextsibling* as binary relations. We can define an MSO formula ϕ to represent the first recursive rule thus:

$$\begin{aligned} \phi(x, y) \equiv\, &\text{NP}(x) \wedge \text{NP}(y) \wedge \text{lastsibling}(y) \\ &\wedge \exists z(\text{firstchild}(x, z) \wedge \text{Adj}(z) \wedge \text{nextsibling}(z, y)). \end{aligned}$$

In this case $(x, y) \in \phi^*$ if and only if every node on the unique path between node x and node y has two children: Adj as its first child and and an NP as the second. We can describe nodes that induce subtrees conforming to the grammar as follows:

$$\varphi(x) \equiv \exists y \exists z(\phi^*(x, y) \wedge \text{firstchild}(y, z) \wedge \text{N}(z) \wedge \text{lastsibling}(z)).$$

Moreover, we can generalize this translation for arbitrary CFGs.

The ability to express transitive closures is an important part of the why MSO is interesting to mathematical linguists and database theorists. The fact that these closures can be described for arbitrary MSO relations means that MSO has a high level of expressiveness. The main tool for exploring the boundaries of this expressiveness is Ehrenfeucht-Fraissé games. These games have played an important role in determining whether other formalisms have the expressive capability of MSO. As such, Ehrenfeucht-Fraissé games are reviewed in the next section.

4.5 MSO Equivalence and Ehrenfeucht-Fraisse games

Expressibility results for MSO are generally established using *Ehrenfeucht-Fraissé games*. These games basically allow us to determine if a pair of structures can be distinguished by a formula of a particular logic. This idea of distinguishability leads to the notion of **MSO types**. They also provide a way of determining how many variables are required to express structural properties. In the MSO framework, Ehrenfeucht-Fraissé games have been used extensively to prove that other formalisms (eg. monadic datalog, cf. Section 4.10), are expressively equivalent to MSO on a given signature. In order to define these games we need to establish the concept of MSO equivalence with respect to quantifier rank. Note, the definitions in this section are adapted from Libkin (1998, Section 7.2).

Definition 4.7. *The* ***quantifier rank*** *of an MSO formula φ is the maximum degree of nesting of first-order and set quantifiers in φ.*

Let $\mathcal{A}$ and $\mathcal{B}$ be finite structures on the same signature σ. Now, consider the tuple $\vec{a} = a_1, \ldots, a_n$ with $a_i \in A$ for $1 \leq i \leq n$ where A is the domain of $\mathcal{A}$. $(\mathcal{A}, \vec{a})$ denotes a finite structure $\mathcal{A}$ with $\vec{a}$ distinguished constants. Similarly let $(\mathcal{B}, \vec{b})$ be a finite stucture with $\vec{b} = b_1, \ldots, b_n$ distinguished constants taken from B the domain of $\mathcal{B}$.

Definition 4.8. $(\mathcal{A}, \vec{a}) \equiv_k^{MSO} (\mathcal{B}, \vec{b})$ *if for each MSO formula φ of quantifier depth k:*

$$(\mathcal{A}, \vec{a}) \models \varphi \Leftrightarrow (\mathcal{B}, \vec{b}) \models \varphi$$

Now, $\equiv_k^{MSO}$-equivalence is an equivalence relation on finite structures. So, if we can determine that a structure satisfies an MSO formula φ we know that all structures in its $\equiv_k^{MSO}$-equivalence class do as well. Structures have the same $\equiv_k^{MSO}$**-type** if they belong to the same $\equiv_k^{MSO}$-equivalence class. Membership in this equivalence class can be determined using the technique of Ehrenfeucht-Fraissé games.

Ehrenfeucht-Fraissé games are also known as pebble games. Players take turns to place k pebbles on domain elements of each structure. Each pebble represents a quantified variable. At the end of the game we check if the structures induced by the pebbles are isomorphic. If we can prove that the induced structures are always isomorphic no matter what strategy is used to place the pebbles then we know the original structures cannot be distinguished with MSO formulae of quantifier depth k. Hence they are in the same $\equiv_k^{MSO}$-equivalence class. The formal definition of an Ehrenfeucht-Fraissé game is given below.

Definition 4.9. *An* ***MSO Ehrenfeucht-Fraissé game*** *has two players, a spoiler and a duplicator. The game is played over structures $(\mathcal{A}, \vec{a})$ and $(\mathcal{B}, \vec{b})$ with a shared signature σ. Let A and B be the domains of $\mathcal{A}$ and $\mathcal{B}$ respectively. The spoiler can make two types of move:*

1. *Point move: The spoiler selects an element $p \in A$ or $q \in B$. The duplicator picks a element in the opposite structure.*

2. *Set move: The spoiler selects a subset $U \subseteq A$ or a subset $V \subseteq B$. The duplicator responds with a subset of the other structure.*

After k rounds $\vec{p} = p_1, \ldots, p_r \in A$ and $\vec{q} = q_1, \ldots, q_p \in B$ have been selected in point moves. Similarly, $\vec{U} = U_1, \ldots, U_s \subseteq A$ and $\vec{V} = V_1, \ldots, V_s \subseteq B$ are selected in set moves. Note $r + s = k$. The duplicator wins a k-round game if the mapping $a_i \rightarrow b_i, 1 \leq i \leq r$ is a partial isomorphism from $(\mathcal{A}, \vec{a}, \vec{U})$ to $(\mathcal{B}, \vec{b}, \vec{V})$. That is, for all $1 \leq i \leq r$, $1 \leq j \leq s$, $p_i \in U_j$ if and only if $q_i \in V_j$ and for every atomic formula $\varphi(\vec{x})$ containing no set variables $\mathcal{A} \models \varphi[\vec{p}, \vec{a}]$ if and only if $\mathcal{B} \models \varphi[\vec{q}, \vec{b}]$.

The duplicator has a winning strategy if they can win every k-round game regardless of the choices of the spoiler. If the duplicator has a winning strategy in any k-round MSO game on $(\mathcal{A}, \vec{a})$ and $(\mathcal{B}, \vec{b})$, then $(\mathcal{A}, \vec{a}) \equiv_k^{MSO} (\mathcal{B}, \vec{b})$ for some fixed quantifier depth k.

Ehrenfeucht-Fraissé games are, unsurprisingly, hard to play on large structures. Quantifier depth is usually relatively small compared to the size of structures. This means that a large number of games to need to be played to establish $\equiv_k^{MSO}$. The following theorem by Neven and Schwentick (2002) allows us to derive $\equiv_k^{MSO}$-equivalence of trees in terms of subtrees and envelopes.

Theorem 4.10. *(Neven and Schwentick, 2002) Let $\mathcal{T}$ and $\mathcal{S}$ be trees with domains T and S respectively. Consider nodes $v \in T$ $w \in S$ that have $n(\geq 0)$ children each. Let v_i and w_i*

($1 \leq i \leq n$) be the i-th child of v, w resp. Let label(x) *denote the set of unary predicates labelling node x.*

1. *If* $(\overline{\mathcal{T}_v}, v) \equiv_k^{MSO} (\overline{\mathcal{S}_w}, w)$ *and* $(\mathcal{T}_v, v) \equiv_k^{MSO} (\mathcal{S}_w, w)$ *then* $(\mathcal{T}, v) \equiv_k^{MSO} (\mathcal{S}, w)$.
2. *If* $(\mathcal{T}_{v_i}, v_i) \equiv_k^{MSO} (\mathcal{S}_{w_i}, w_i)$, *for* $1 \leq i \leq n$ *and* label(v) = label(w) *then* $(\mathcal{T}_v, v) \equiv_k^{MSO} (\mathcal{S}_w, w)$
3. *Let* $1 \leq i \leq n$. *If* $(\overline{\mathcal{T}_v}, v) \equiv_k^{MSO} (\overline{\mathcal{S}_w}, w)$, label($v_i$) = label($w_i$) *and* $(\overline{\mathcal{T}_{v_j}}, v_j) \equiv_k^{MSO} (\overline{\mathcal{S}_{w_j}}, w_j)$ *for* $j \in \{1, \ldots, n\} - \{i\}$ *then* $(\overline{\mathcal{T}_{v_i}}, v_i) \equiv_k^{MSO} (\overline{\mathcal{S}_{w_i}}, w_i)$.

The proof of this theorem uses the fact that winning strategies of the duplicator on different subtrees can be combined to provide a winning strategy for the combined structure. This theorem allows us to determine the $\equiv_k^{MSO}$-type of a tree inductively. It turns out to be crucial for proving the equivalence of MSO to the other formalisms described. In order to motivate the existence of these alternative formalisms we need to understand the mechanisms for evaluating MSO queries and their complexity complexity. The most important of these are the unranked tree automata.

4.6 Unranked Tree Automata

MSO has strong ties to the theory of tree automata. Unranked tree automata are similar to the tree-walking automata reviewed in Section 3.11. However, unranked tree automata use a parallel mode of computation. That is, multiple heads are used at each step in the run of the automaton. Tree automata provide a procedural view on trees that complements the declarative nature of MSO formulae. This relationship has led a number of results on the expressiveness and evaluation complexity of both of these formalisms. In this section we review the basics of unranked tree automata and review the classic result linking MSO sentences with bottom-up tree automata. From this, we see how tree automata can be used to compute MSO queries on trees. Tree automata also provides a number of insights into the tractability of MSO query evaluation.

The purpose of this section is to examine the relationship between tree automata and MSO in the context of querying unranked trees. Thus, we only review unranked automata although similar result were first developed for ranked tree automata (Gécseg and Steinby, 1997). The definitions and results reviewed below have been adapted from Neven and Schwentick (2002).

Definition 4.11. *A* ***non-deterministic bottom-up unranked tree automaton*** *(NBTA^u) is a tuple $B = (Q, \Sigma, F, \delta)$. Q is a finite set of states, $F \subseteq Q$ are final states, and δ is a transition function. $\delta : Q \times \Sigma \rightarrow 2^{Q^*}$ and $\delta(q, a)$ is a regular language $\forall \sigma \in \Sigma, q \in Q$.*

That is, a node v with label a will obtain state q if the states of its children are in the language defined by $\delta(q, a)$. A *deterministic* BTA^u is defined as above with the restriction $\delta(q, a) \cap \delta(q', a) = \emptyset$ for $a \in \Sigma$, $q, q' \in Q$ and $q \neq q'$. Let t be a node in tree $\mathcal{T}$ with label σ and n children. Let $t_1, \ldots, t_n$ be the subtrees rooted by the n children of t. The *semantics* of B on the tree rooted at t, $\delta^*(t)$, is defined inductively as follows (Neven and Schwentick, 2002). If t consists of only one node then $\delta^*(t) = \{q \mid \epsilon \in \delta(q, a)\}$. Otherwise,

$$\delta^*(t) = \{q | \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n) \wedge q_1, \ldots, q_n \in \delta(q, a)\}.$$

The concept of accepted trees, tree languages (i.e. $L(B)$) and recognizability are analogous to those for string automata (Gécseg and Steinby, 1997). That is, a tree language, $L(B)$ is the set of trees accepted by the tree automaton B. However, unlike string automata, the actual runs of unranked tree automata involve parallel evaluation of several nodes at the same time. This parallel model is based on *cuts* and *configurations*. The following definitions give the framework for how a BTA^u is actually run. In the following, let $\mathcal{T}$ be a τ-tree and let T be the domain of $\mathcal{T}$.

Definition 4.12. *A* ***cut*** *of tree $\mathcal{T}$, C, is subset of T that contains one node from each path from the root to the leaves.*

Definition 4.13. *A* ***configuration*** *of the automaton A on a tree $\mathcal{T}$ is a map $c : C \to Q$ where C is a cut of $\mathcal{T}$. A BTA^u on tree $\mathcal{T}$ makes a* transition *between two configurations $c_1 : C_1 \to Q$ and $c_2 : C_2 \to Q$ iff it makes an upwards transition. This is denoted $c_1 \to c_2$.*

Definition 4.14. *A* ***run*** *is a sequence of configurations, $c_1, \dots c_n$, with c_1 a start configuration and we have $c_1 \to \dots \to c_n$.*

Definition 4.15. *Let* $\mathrm{root}(\mathcal{T})$ *denote the root of tree $\mathcal{T}$. A run is accepting if c_n is accepting. ie $c_n : \mathrm{root}(\mathcal{T}) \to Q$ and $c_n(\mathrm{root}(\mathcal{T})) \in F$ $\mathcal{T}$ is accepted by a BTA^u B if there is an accepting run of B on $\mathcal{T}$.*

Thus, a bottom up tree automaton B starts with all the leaves of tree $\mathcal{T}$ in the cut of the initial configuration. If all children of a node v are in a configuration c_i and no transition is possible then the automaton rejects. The automaton must also reject if the final configuration does not map the root node to a final state (i.e. $q \in F$). The notion of regular languages can now be extended to unranked trees.

Definition 4.16. *A unranked tree language $L(B)$ is regular if it accepted by an unranked tree automaton.*

The following theorem provides the bridge between bottom up unranked tree automata and MSO.

Theorem 4.17. *(Neven and Schwentick, 2002) An unranked tree language is regular iff it is definable in MSO.*

This means boolean MSO queries can be performed using unranked tree automata. It also provides a way of establishing the computation complexity of MSO as a tree query language.

4.7 Data Complexity of MSO

In reviewing the tractability of MSO as a query language we need to consider both data and query complexity. The size of the data tends to overshadow that of the query in this respect. This section reviews the data complexity of MSO. This demonstrates how tree automata can be used to understand MSO queries on trees. This problem is best considered in terms of model-checking (cf. Section 2.3). First we consider the notion of fixed-parameter tractability as a way of separating data and query complexity. These definitions are taken from Libkin (1998).

Definition 4.18. *A model-checking problem for a logic $\mathcal{L}$ on a class of structures $\mathcal{C}$ is* ***fixed-parameter tractable*** *if there is a constant p and a function $g : \mathbf{N} \to \mathbf{N}$ such that for every $\mathcal{A} \in \mathcal{C}$ and every sentence $\phi \in \mathcal{L}$ checking whether $\mathcal{A} \models \phi$ can be done in time $g(|\phi|) \cdot |\mathcal{A}|^p$.*

Definition 4.19. *A model-checking problem for $\mathcal{L}$ on $\mathcal{C}$ is* ***fixed-parameter linear****, if the problem is fixed-parameter tractable with $p = 1$. That is, the problem of checking $\mathcal{A} \models \phi$ can be done in time $g(|\phi|) \cdot |\mathcal{A}|$.*

Libkin (1998) derives the data complexity of MSO on trees using by showing it is fixed-parameter linear.

Theorem 4.20. *(Libkin, 1998) Over strings and trees (ranked or unranked), evaluating MSO sentences is fixed-parameter linear. In particular, over strings and trees, the data complexity of MSO is linear.*

The proof of this theorem uses the fact that MSO sentences over trees can be encoded as bottom-up unranked tree automata. Once this encoding is completed, the MSO sentence is evaluated in one run of the the automaton that visits each node once. That is, the evaluation time of the automaton is linear in the size of the tree. This result has also been extended to MSO formulae with free variables (that is, k-ary queries) (Arnborg et al., 1991). Thus, unary MSO queries have linear data complexity. As previously mentioned, our primary focus is on node-selecting tree query languages. Hence, the next section reviews formalisms for evaluating unary MSO queries.

4.8 Evaluating Unary MSO Queries

This section reviews how the existing computational models of tree automata and attribute grammars be extended to evaluate MSO queries.

Query automata (Neven and Schwentick, 2002) extend classical bottom up unranked tree automata, allowing them to perform node selecting queries, in three ways. First, expression of unary queries is done via a selection function. Nodes are selected for output depending on whether they are labelled with certain states during a run. Second, two-way automata are used. Both up and down transitions are allowed and occur depending on the current states of nodes in a configuration. Third, stay transitions are allowed. Stay transitions basically allow nodes in a configuration the ability to change state depending on the states of siblings. **Strong query automata** are query automata that make at most one stay transition for the children of each node. Strong query automata have been shown to be equivalent to unary MSO over unranked trees.

Neven and van den Bussche (2002) present **Boolean attribute grammars** (BAGs) as a formalism for specify unary queries over derivation trees of context-free grammars (CFGs). BAGs are used to assign boolean valued attributes to nodes. Nodes are selected if they are assigned specific a attribute and boolean value combination. This assignment is described by a CFG in which terminal and non-terminal symbol have a (possibly empty) set of attributes. Each production rule in the CFG is also associated with a set of *semantic rules*. Semantic rules are propositional formulae that determine the values of attributes associated with symbols in a production rule. Nodes are assigned attributes associated with their label in the CFG. The productions involving that label determine the semantic rules used to evaluate those attribute values. These rules are evaluated over attribute values of a node's children, siblings or parent. Neven (2005) has extended this formalism to query the derivation of extended context free grammars and show that this coincides with unary MSO. That is, attribute grammars for querying unranked trees.

Like the unranked tree automata of the previous section, both query automata and BAGs have linear time data complexity. Data complexity tends to be the dominant factor in database query evaluation. However, we also need to consider the contribution of the query, ϕ to evaluation complexity.

In the previous section we reviewed the fixed-parameter linearity of MSO over trees. In the definition of fixed-parameter linearity, query complexity was represented as a function over query size, $g(|\phi|)$. Now, just as the linear data complexity result depended on the encoding of MSO formulae into tree automata, so does the function g. However, such an encoding may require determinization of automata to allow negation. This leads to a massive blow-out in the number of states required. Unfortunately, the conversion of MSO formulae into automata is **non-elementary** in terms of the size of the query ϕ (Libkin, 1998).[3] That is, the function of the query size results in an extremely large value. The casts serious doubt over the tractability of MSO. The situation does not improve for k-ary MSO queries.

Similarly, mappings from MSO to both query automata and BAGs also require non-elementary space (Neven and Schwentick, 2001). So the combined complexity for evaluating unary MSO using these formalisms is no better than that of MSO sentences. This motivates an investigation into tree query formalisms that are expressively equivalent to MSO but better combined evaluation complexity. A number of such formalism have been developed and they are reviewed in the next section.

4.9 Unary MSO Equivalent Formalisms

Although MSO is able to express closures and has linear time data complexity it is still not an ideal tree query language. MSO is expressive and precise as a query language. However, it is not very user friendly. Queries can quickly become complicated and it does not lend itself to easily visualization. This is especially so if we require users to fully specify formulae for closures. Moreover, the non-elementary nature of the mapping between MSO and the formalisms described above is not very favourable in terms of combined complexity. However, it has been argued that MSO provides the right level of expressiveness for querying unranked trees (Neven and Schwentick, 2000). As a result, much work has been done developing efficient formalisms that are expressively equivalent to unary MSO on unranked trees. A number of these formalisms are discussed below.

Efficient Tree Logic (ETL) (Neven and Schwentick, 2000) was developed as a first step towards an easy to use unary MSO equivalent formalism. ETL consists of a guarded fragment of MSO where quantifiers are only allowed to bind sets or nodes below the current node. Besides the usual boolean connectives, ETL allows vertical and horizontal regular path expressions over binary ETL formula. That is, explicit regular expressions over siblings and descendants. Horizontal path expressions can be nested inside vertical ones and vice versa. Moreover, ETL proposes a modularization of queries into path and subtree formulae. Thus a node satisfies a query if the subtree and envelope induced by that node satisfies certain separated properties. Properties below a node are expressed with subtree formulae. Envelope properties are expressed using a path formula from the root to the node in question.

ETL query evaluation has linear time data complexity and doubly exponential time query complexity [4]. This is a considerably better than query evaluation complexity of MSO. The restricted syntax and use of regular expressions certainly makes ETL much easier to understand. The use of regular expressions and nesting similar to that of Regular XPath filter expressions (cf. Section 3.9).

Murata (2001) takes a similar regular expression based approach in defining **hedge regular expressions**. Hedges are sequences of ordered trees and so the hedge query problem is a generalization of the query problem for trees. Like ETL, hedge regular expressions include two

[3]A function $f : \mathbf{N} \to \mathbf{N}$ is **elementary** if for some fixed l:

$$f(n) < 2^{n^{\cdot^{\cdot^{n}}}} \}l \text{ times}, \forall n$$

[4]See: http://ls1-www.cs.uni-dortmund.de/%7Etick/PubFiles/node1.html#ab:etlcor

types of closures to deal with the two dimensions inherent in trees. The Kleene star is used to represent horizontal closures over hedges. In the vertical dimension, rather than regular path expressions, substitution symbols are used to embed trees within trees. The embedding operator allows subtrees to be embedded an arbitrary number of times. Thus we can finitely specify an arbitrary number of ordered trees with arbitrary depth.

Hedge regular expressions are evaluated using hedge automata. Unary queries are specified using pointed hedge representations. These constrain which nodes are selected by using hedge regular expressions to the specify properties of a node's subtree, envelope and the subtrees of siblings. Selection queries using pointed hedge representations can be evaluated in time linear in the size of the data and exponential in the size of the expression.

Finally, **Monadic datalog** (Gottlob and Koch, 2004) is a restricted form of the database programming language datalog (Abiteboul et al., 1995).

From this brief survey of formalisms equivalent to unary MSO, monadic datalog appears to be the most straightforward to use as a tree query language. Its usability derives in part from its origins as a programming language. Moreover, monadic datalog programs have linear time combined complexity. Thus, it also presents the best balance between expressiveness and evaluation complexity. The following sections examine monadic datalog and theoretical properties in more detail.

4.10 Monadic Datalog Syntax and Semantics

Monadic datalog is a restricted version of the query language datalog (Abiteboul et al., 1995). As such, this section reviews the syntax and semantics of datalog and the restrictions imposed on monadic datalog. The following definitions are adapted from Gottlob and Koch (2004).

A datalog program consists of datalog rules equivalent to horn clauses of first-order logic without functions. That is, formulae of the form

$$h \leftarrow b_1, \ldots, b_n.$$

where $h, b_1, \ldots, b_n$ are atoms. An **atom** $p(x_1, \ldots, x_m)$ is a predicate p of arity m where each $x_1, \ldots, x_m$ may be a constant drawn from a domain, T say, or a variable. The **rule head** is the 'implied atom' h and $b_1, \ldots, b_n$ form the **rule body**. Commas in the rule body denote conjunction. Variables appearing in the head must also appear in the body. That is, rules must be safe. However, datalog rules may be recursive. As usual, a **valuation** is a function that maps variables to elements of the domain T and is the identity on constants. An atom is **ground** if does not contain any variables. The **size** of a datalog rule is the number of atoms in its body. The size of a datalog program is the sum of the sizes of its rules.

Predicates that appear as the head of a rule are **intensional**. In *monadic datalog* all intensional (IDB) predicates must be unary. That is, intensional predicates are user defined. All other predicates are **extensional** (EDB). The **extension** of a monadic datalog program is the set of ground atoms or facts. The *query predicate* is a distinguished intensional predicate. In the following, we will assume that monadic datalog programs have extensional predicates drawn from the following tree signature:

$$\tau_{md} = (\text{root}\ , \text{leaf}\ , a \in \Sigma,\ \text{firstchild},\ \text{nextsibling},\ \text{lastsibling})$$

Thus, monadic datalog programs are evaluated with respect to τ_{md} structures (trees), $\mathcal{T}$, with domain T and valuations map variables to nodes in a tree. The semantics of monadic datalog program are derived using the immediate consequence operator. The immediate consequence of a set of ground atoms is defined as follows.

Definition 4.21. *Let* Vars(p) *denote the set of variables in atom p. Now, given a monadic datalog program $\mathcal{P}$, and a set of ground atoms X, $h(t)$ $t \in T$ is an* ***immediate consequence*** *of X if $h \leftarrow b_1, \ldots, b_n \in \mathcal{P}$ and there is some valuation ϕ such that $\phi(\text{Vars}(b_1)), \ldots, \phi(\text{Vars}(b_n)) \in X$ and $\phi(\text{Vars}(h) \cap \text{Vars}(b_i)) = t$ for some $1 \leq i \leq n$.*

That is, the immediate consequence of a set of ground atoms X is the set of facts that can be derived from X using the rules of program $\mathcal{P}$. Now, let $\mathcal{B}$ be the set of all possible ground atoms that can be derived from relations in the signature τ_{md} and the domain T. That is, for any set of ground atoms X, $X \subseteq \mathcal{B}$.

Definition 4.22. *Given a monadic datalog program $\mathcal{P}$, and a set of ground atoms X, the* ***immediate consequence operator*** *is a function $T_{\mathcal{P}} : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$ that maps X to the union of X and all facts that are immediate consequences of X.*

Now, we can inductively define $T^{i+1}_{\mathcal{P}} = T_{\mathcal{P}} \circ T^{i}_{\mathcal{P}}$ with $T^{0}_{\mathcal{P}}(X) = X$. The least k such that $T^{k+1}_{\mathcal{P}}(X) = T^{k}_{\mathcal{P}}(X)$ is the least fixpoint of $T_{\mathcal{P}}$. This is denoted $T^{\omega}_{\mathcal{P}}$. $T^{\omega}_{\mathcal{P}}$ is equivalent to the minimal model of $\mathcal{P}$. This defines the meaning of a monadic datalog program.

Definition 4.23. *Let $extension(\mathcal{P})$ be the extension of program $\mathcal{P}$. The meaning of $\mathcal{P}$ is the set of facts $T^{\omega}_{\mathcal{P}}(extension(\mathcal{P}))$.*

Monadic datalog expresses unary queries. Let q be our distinguished query predicate in program $\mathcal{P}$. The result of such a query is

$$result = \{x \mid q(x) \in T^{\omega}_{\mathcal{P}}(extension(\mathcal{P})\}.$$

Having covered the basic syntax and semantics of this language we now turn to more important matters. The next section reviews the expressive equivalence of monadic datalog and unary MSO over trees.

4.11 The Expressiveness of Monadic Datalog

Although monadic datalog program is made up of first-order rules, it is more expressive than a first-order theory. In fact, monadic datalog has the same expressiveness over τ_{md} as unary MSO.

Theorem 4.24. *(Gottlob and Koch, 2004) For each unary MSO query over unranked trees, there exists a monadic datalog program over τ_{md} that defines the same query and vice versa.*

The proof that every monadic datalog program has an equivalent unary MSO formula gives an intuition for the connection between monadic datalog and MSO. Intensional unary predicates can interpreted as set variables so that $P(x)$ means $x \in \mathrm{P}$. Let $\mathrm{P}_1, \ldots, \mathrm{P}_n, \mathrm{Q}$ be intensional predicates in monadic datalog program $\mathcal{P}$. Let Q be the query predicate. The following MSO formula is equivalent to $\mathcal{P}$:

$$(\forall\, \mathrm{P}_1) \ldots (\forall\, \mathrm{P}_n), (\forall\, \mathrm{Q})(\mathrm{SAT}(\mathrm{P}_1, \ldots, \mathrm{P}_n) \rightarrow x \in \mathrm{Q})$$

Where $\mathrm{SAT}(\mathrm{P}_1, \ldots, \mathrm{P}_n)$ is the conjunction of rules of $\mathcal{P}$ as logical formulae with free variables universally quantified and bound to the appropriate rule.

The proof of the converse is much harder and uses the result by Neven and Schwentick (2002) that allows us to infer the $\equiv^{MSO}_{k}$-type of a given tree (structure) via subtrees and envelopes (cf. Section 4.5). Recall, if $(\mathcal{A}, a) \equiv^{MSO}_{k} (\mathcal{B}, b)$ then $(\mathcal{A}, a) \models \varphi \leftrightarrow (\mathcal{B}, b) \models \varphi$. Gottlob and

$$\text{following\_sibling}_{NP}(x) \leftarrow \text{nextsibling}(x, x_0), \text{NP}(x_0)$$
$$\text{following\_sibling}_{NP}(x) \leftarrow \text{nextsibling}(x, x_0), \text{following\_sibling}(x_0)$$

Figure 4.1: Following sibling in monadic datalog.

$$\begin{aligned}
\text{Q}(x) &\leftarrow \overline{\text{descendant}_{\text{ON}}}(x), \text{SIMPX}(x). \\
\overline{\text{descendant}_{\text{ON}}}(x) &\leftarrow \text{firstchild}(x, x_0), \text{aux_}\overline{\text{descendant}_{\text{ON}}}(x_0). \\
\overline{\text{descendant}_{\text{ON}}}(x) &\leftarrow \text{leaf}(x). \\
\text{aux_}\overline{\text{descendant}_{\text{ON}}}(x) &\leftarrow \overline{\text{descendant}_{\text{ON}}}(x), \overline{\text{ON}}(x), \text{lastsibling}(x). \\
\text{aux_}\overline{\text{descendant}_{\text{ON}}}(x) &\leftarrow \overline{\text{descendant}_{\text{ON}}}(x), \overline{\text{ON}}(x), \text{nextsibling}(x, x_0), \text{aux_}\overline{\text{descendant}_{\text{ON}}}(x_0).
\end{aligned}$$

Figure 4.2: Negation in Monadic Datalog

Koch (2004) present a decomposition that simultaneously finds witness structures to recognize different $\equiv_k^{MSO}$-types. This leads to a monadic datalog program describing the $\equiv_k^{MSO}$-type that entails φ. Thus, monadic datalog is expressively equivalent to unary MSO over the signature τ_{md}.

Even though we know that monadic datalog has the same expressiveness as MSO, there are differences in the way structure can be described. On the one hand, we can define binary relations in MSO as formulae with two free variables. On the other hand, this is not possible in monadic datalog. Since all intensional predicates are unary in monadic datalog, the fixpoint operator can only derive facts about individual nodes. We cannot explicitly say that two nodes have some relation to each other. Thus the usual binary relations, like following sibling, must be considered in a different light.

On the other hand, we can say that node x is related in a specific way to some node y that has property P, where P is definable in monadic datalog. For example, in the program shown in Figure 4.1, the predicate following_sibling is true at nodes that are following siblings of NP labelled nodes. This also demonstrates how transitive closures can be specified.

Moreover, monadic datalog does not include explicit negation. Like caterpillar expressions, negation can be expressed implicitly. A monadic datalog version of the query: *Find clause nodes (SIMPX) that do not dominate any subject nodes (ON)* (cf. (Kepser, 2004)) is shown in Figure 4.2. We basically need to implement a traversal of the subtree below SIMPX labelled nodes.

In this program $\overline{\text{ON}}(v)$ is a fact if node v does not have label ON. In order to use the predicate $\overline{\text{ON}}$ we need to either define it as a set of rules of the form $\overline{\text{ON}}(x) \leftarrow \sigma(x)$ where $\sigma \in \Sigma - \{ON\}$. Alternatively we can extend τ_{md} to include the complement of every unary extensional predicate U. The latter is the preferred option as complements are simple to precompute and adding them does not change expressiveness or asymptotic complexity results.

These examples highlight an important difference between monadic datalog and MSO in the way queries are defined. The key point is that, once a fact (ground atom) has been derived by the immediate consequence operator it is not possible to tell what set of facts it was derived from. That is, monadic datalog programs easily forget how they came to a certain fact. This makes it natural to frame monadic datalog programs in terms of paths. The path-based nature of the query language is even clearer when we consider that every monadic datalog program can be rewritten into Tree-Marking Normal Form.

Definition 4.25. *A monadic datalog program $\mathcal{P}$ over τ_{md} is in* ***Tree-Marking Normal Form*** *(TMNF) if each rule of $\mathcal{P}$ is in one of the forms:*

1. $\mathrm{P}(x) \leftarrow \mathrm{U}(x)$.
2. $\mathrm{P}(x) \leftarrow \mathrm{P}_0(x_0), \mathrm{B}(x_0, x)$.
3. $\mathrm{P}(x) \leftarrow \mathrm{P}_0(x_0), \mathrm{B}(x, x_0)$.
4. $\mathrm{P}(x) \leftarrow \mathrm{P}_0(x), \mathrm{P}_1(x)$.

Where $\mathrm{P}, \mathrm{P}_0, \mathrm{P}_1$ are IDB predicates, U and B are unary and binary EDB predicates respectively.

TMNF rules involving two variables can be seen as moving one step in the a tree. In τ_{md} we can move to the parent, first child, or immediate left or right sibling. Similarly, one-variable rules can be seen as tests at particular nodes. TMNF rules have a striking similarity to the transition rules of tree walking automata and hence caterpillar expressions (cf. Section 3.11).

In fact, caterpillar expressions are often used as a shorthand for describing monadic datalog programs. Gottlob and Koch (2002) give a straightforward algorithm translating caterpillar expressions over the signature τ_{md} into TMNF programs. Moreover, Gorris and Marx (2005) have defined TMNF caterpillars as a path-based formalism that combines an explicit path-based syntax with the expressive power of MSO on trees as follows.

Definition 4.26. *Let $\mathcal{P}$ be a TMNF program. A* ***TMNF caterpillar expression*** *is a caterpillar expression where only the predicates $IDB(\mathcal{P})$ occur as tests.*

These theoretical results and examples have shown monadic datalog to be a very expressive query language. It is also very different to query languages thus far reviewed. Monadic datalog rules define sets of nodes from a set of facts. This leads to a naturally path-based approach to query formulation. Also, unlike other MSO equivalent formalisms, results of monadic datalog query are evaluated using the the immediate consequence operator. This fixpoint approach has lead to theoretically efficient evaluation complexity. The evaluation complexity of monadic datalog is considered next.

4.12 Evaluation Complexity of Monadic Datalog

Monadic datalog is NP-complete with respect to combined complexity over arbitrary finite structures (Gottlob and Koch, 2004). However, it has much better evaluation complexity when restricted to τ_{md}-trees. In fact, Gottlob and Koch (2004) have shown that monadic datalog queries have linear time data and query complexity.

Theorem 4.27. *(Gottlob and Koch, 2004) Let $\mathcal{T}$ be a τ_{md}-structure (tree), with domain T. Monadic datalog query evaluation has $O(|\mathcal{P}| \cdot |T|)$ combined complexity, where $|\mathcal{P}|$ is the size of the program and $|T|$ is the size of the tree.*

The proof of this uses the fact that binary relations in the signature τ_{md} are one-to-one mappings. Consider rules where each body predicate contains the head variable. This means that the valuation of one variable in the rule determines the valuation of all the others. As a result, the number of possible ground instances is linear in the size of the data (domain).

This set of ground instances is a ground program. The fixpoint of a ground program can be computed in time linear in the size of the program. Gottlob and Koch (2004) describe how rules of monadic datalog programs can be transformed into rules of the form described above using propositional variables. This can done in time linear in the size of the program. The composition of these two steps gives the required result.

This means that monadic datalog queries can be evaluated theoretically much more efficiently then unary MSO queries. In fact, it has also been shown that monadic datalog queries can evaluated in just two passes of the tree via selecting tree automata (Koch, 2003). This efficient evaluation of these queries is discussed in the next section.

4.13 Selecting Tree Automata and Arb

Selecting Tree Automata (STA) (Frick et al., 2003) provide a method for efficiently evaluating monadic datalog queries.

Definition 4.28. *A selecting tree automaton (STA) is a tuple $\mathcal{A} = (Q, \Sigma, F, \delta, S)$ where (Q, Σ, F, δ) is a non deterministic bottom-up binary tree automaton and $S \subseteq Q$ is a set of selecting states. The unary query, q, defined by an STA $\mathcal{A}$ maps trees $\mathcal{T}$ to the set:*

$$q(\mathcal{A}) = \{v \in V^T | \rho(v) \in S \text{ for every accepting run } \rho \text{ of } \mathcal{A} \text{ on } T\}.$$

Selecting tree automata are only defined with respect to binary trees. This requires unranked trees to be converted into a binary format (left child as *firstchild* and right child as *nextsibling*). There is nothing stopping an extension of STA to (non-binary) bottom-up unranked tree automata. However, the conversion to binary conveniently gets around the problem of having to define transition functions based on regular languages.

An STA only has upwards transitions and is non-deterministic. A consequence of the non-determinism is that a node must be in a selecting state in every accepting run for it to be selected. Frick et al. (2003) also show that purely deterministic STAs cannot express all unary queries definable by MSO. However, for each STA definable query there is (another) STA where it is equivalent to require that a node to be in a selecting state in at least one accepting run. This is done by defining the negation of the query in another STA, $\mathcal{B}$, and then setting selecting states to the complement of the selecting states of $\mathcal{B}$. Now,

Theorem 4.29. *(Frick et al., 2003) A query (on binary-trees) is definable in MSO if, and only if, if is definable by an STA.*

Hence, selecting tree automata are equivalent to query automata and monadic datalog. However, note that the proof of this still requires the translation of an MSO formula into a tree automaton and hence it requires non-elementary space. It is not at all obvious how to frame a query as an STA. Their main use, so far, has been as a way of evaluating monadic datalog (specifically TMNF) queries on streaming XML (Koch, 2003).

Koch (2003) describes **Arb**, an implementation of selecting tree automata for streaming XML. Here, the aim was to keep the expressiveness of MSO while efficiently dealing with the data. This means using a bounded amount of memory and limiting the number of passes through the input tree during query evaluation. To deal with these and the non-determinism of STAs, queries are processed in two-phases.

The first phase involves a run of a deterministic bottom-up automaton where states are sets of rules inferred from the TMNF program. This is very much like the powerset construction used determinize NFAs. This bottom-up run labels nodes with sets of reachable states. These are predicates that could be true at a particular node given the predicates reachable in subtrees below. The top-down run then determines states (predicates) from which the root can be reached. It also determines which which of the reachable predicates at a node are true at that node. That is, this two-phase evaluation determines the facts that are in the minimal model of the TMNF program.

Once the automata have been computed from the query, evaluation time is linear in the size of the data (ie. the time involved in running the two automata over the input tree) and independent of the query. The work involved in creating the automata is mostly centered on defining the transition functions. However, transitions are computed lazily during the run of the automaton.

Arb presents a very attractive mechanism for evaluating monadic datalog queries. The fact that queries can be evaluated in just two passes of the tree means that applications requiring highly expressive, streaming tree queries are well within reach.

4.14 Conclusion

MSO is expressive and well understood formalism for reasoning about trees. Moreover, tree automata provide a natural evaluation mechanism for MSO queries. This equivalence provides us with a choice of viewing tree language problems from a declarative or procedural perspective. However, the expressiveness of MSO comes at a price.

On the one hand, (Kepser, 2004) brushes aside the non-elementary nature of MSO noting that the complexity comes from the extra precision available in the language. On the other hand, this extra complexity does seem somewhat unnecessary when comparing MSO to other formalisms, particularly monadic datalog. MSO and monadic datalog are equivalent expressively. However, that does not mean it is as simple to describe a structure in one formalism as the other. Monadic datalog is somewhat path oriented. This seems to account for the relative ease in which queries can be written compared to other query languages of equivalent expressiveness. However, lack of negation in makes monadic datalog somewhat less succinct than MSO in many circumstances. While Gottlob and Koch (2004) suggest monadic datalog programs remain relatively small for web wrapping applications, it is not at all clear that this is true for linguistic tree query.

There is still no convincing case for requiring linguistic tree query languages to have the power of MSO. The example queries used to argue for monadic datalog as linguistic query language in Kepser (2004) are all expressible in FO_{tree}. That is, the signature containing binary relations *descendant* and *following sibling*. In fact only the last, most complicated, query cannot be expressed in FO^2_{tree} (ie. Core XPath). MSO has been advocated as the expressive benchmark for tree query. However, there is increasingly strong evidence that MSO should be the expressiveness yardstick for linguistic queries. The next chapter examines the suitability of monadic datalog and the path-based query languages for linguistic tree query.

Chapter 5

Linguistic Tree Query and LPath

The previous chapters surveyed a number of distinct tree query formalisms. This chapter evaluates query languages associated with these formalisms as linguistic tree query languages. The main query languages to emerge from these surveys were the XPath family of languages, caterpillar expressions and monadic datalog. Although these languages are have been designed for querying trees, it is not clear how well they will fare in the linguistic context.

To test this, we examine the linguistically motivated operators of the LPath query language (Bird et al., 2005) in detail. LPath was briefly introduced as an extension of Core XPath for querying treebanks (cf. Section 2.1.7). The XPath family, caterpillars and monadic datalog form a hierarchy in terms of expressiveness. The ability of each of these languages to express the LPath operators gives us a clear indication of the expressive requirements for linguistic tree query. It also indicates how well suited these distinct query paradigms are in general for formulating linguistic queries.

The following sections formally describe LPath and investigate its expressiveness with respect to the tree query formalisms discussed in previous chapters. We find that its expressiveness lies between Core and Conditional XPath. However, LPath cannot express the types of closures required in linguistic queries. This leads us to define Conditional LPath as LPath extended with the conditional axis. We find that Conditional LPath fulfils the navigational and closure requirements for querying linguistic trees. Moreover, it has the same expressiveness as Conditional XPath which suggests a natural fixpoint in expressiveness. This presents further evidence that first-order expressiveness is sufficient for our purposes.

We also consider how LPath operators can be expressed in looping caterpillars. Particularly, their similarity between the *loop* operator and LPath's scoping braces. We also look at expressing LPath operators in monadic datalog. In fact, an examination of the scoping problem in monadic datalog highlights the similarities and differences between path-based formalisms and monadic datalog as query languages.

The syntax and semantics of LPath are introduced in Section 5.1. Section 5.2 and Section 5.3 examine how LPath operators can be expressed in Core and Conditional LPath. Section 5.4 investigates the expressiveness of Conditional LPath. Section 5.5 then discusses the merits and drawbacks of Conditional LPath as a linguistic tree query language. Regular XPath, caterpillars and monadic datalog are also consider in Sections 5.6 to 5.10. We conclude in Section 5.11.

5.1 LPath

LPath was developed to be expressive enough for linguistic query but also to take advantage of relational database theory. As the name suggests, LPath is an extension of XPath 1.0. However, the extensions pertain only to the navigation component of XPath 1.0. Bird et al. (2005) present

`\`	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 5.1: LPath abbreviated axes

```
locpath   :=   abspath | abspath '{' locpath '}' |
               locpath '|' locpath
abspath   :=           | locstep abspath
locstep   :=   axis test | axis test '['fexpr']'
fexpr     :=   locpath | fexpr 'and' fexpr | fexpr 'or' fexpr
               | 'not' fexpr | '(' fexpr ')'
axis      :=   '\' | '\\' | '\\*' | '.' | '/' | '//' | '//*'
               '->' | '<-' | '-->' | '<--' |
               '=>' | '<=' | '==>' | '<=='
test      :=   p | _ | '^'p | p'$'
```

Figure 5.2: LPath syntax. Here `p` is a node label.

three linguistically motivated syntactic additions. These are the *immediate following* axis (and its converse), tree edge alignment and a scoping operator.

The syntax of LPath is described in Figure 5.2. This produces XPath expressions in abbreviated syntax. Figure 5.1 gives the translation of abbreviated axes. It is easy to see how this can be expanded to match the unabbreviated XPath syntax such as that given for Core XPath (cf. Figure 3.2) and vice versa. However, unlike the XPath 1.0 abbreviated syntax `//` stands for the *descendant* axis rather than *descendant or self* (similarly for `\\`). The latter is denoted `//*`. This was a result of a lack of linguistic use cases for the *descendant or self* axis. As in the other navigational XPath variants previously discussed, this syntax does not deal with attributes[1]. Instead we again assume a model where nodes may have multiple labels. The semantics of the new operators is shown in Figure 5.3.

The *scoping* operator is denoted by pairs of braces, `{}`. These braces represent queries constrained to a particular subtree. This subtree is defined as the subtree rooted by the context node immediately before the opening brace. The location path inside the scoping braces is evaluated as if this subtree were the whole of the input. This allows us to write queries restricted to a particular constituent. At first glance, it might appear that scoping can be implemented using query composition. For example, a query of the form `//A{//C-->B}` could be interpreted as `//A` ∘ `//C-->B`. That is, the second query is evaluated over the subtrees induced by the answer set of the first query. However, this does not adequately deal with cases where the

[1]Note, the syntax in Bird et al. (2005) does allow attributes

$$
\begin{array}{lcl}
[\![\hat{\ }p]\!]_{\mathcal{T}} & = & \{x \,|\, p(x) \text{ and } \neg\exists y(x,y) \in [\![\texttt{<--}]\!]_{\mathcal{T}}\}. \\
[\![p\$]\!]_{\mathcal{T}} & = & \{x \,|\, p(x) \text{ and } \neg\exists y(x,y) \in [\![\texttt{-->}]\!]_{\mathcal{T}}\}. \\
[\![\texttt{->}]\!] & = & \{(x,y) \,|\, \neg\exists z(x,z) \in [\![\texttt{-->}]\!]_{\mathcal{T}}, (z,y) \in [\![\texttt{-->}]\!]_{\mathcal{T}}\}. \\
[\![\texttt{<-}]\!] & = & \{(x,y) \,|\, \neg\exists z(x,z) \in [\![\texttt{<--}]\!]_{\mathcal{T}}, (z,y) \in [\![\texttt{<--}]\!]_{\mathcal{T}}\}. \\
[\![\text{locpath}_1\{\text{locpath}_2\}]\!]_{\mathcal{T}} & = & \{(x,y) \,|\, \exists z(x,z) \in [\![\text{locpath}_1]\!]_{\mathcal{T}}, (z,y) \in [\![\text{locpath}_2]\!]_{\mathcal{T}_z}\}.
\end{array}
$$

Figure 5.3: The semantics of LPath operators on tree $\mathcal{T}$. $\mathcal{T}_z$ denotes the subtree rooted at node z

scoping appears inside a path expression (eg. `//A{//C-->B}\\D}`) or inside filter expressions (eg. `//A[{C-->B}]`).

The *immediate following* axis, `->`, is the natural one-step version of the *following* axis, `-->`. We can consider this axis as taking a step to consituents immediately right of the current context node. Left and right tree edge alignment, `^` and `$` respectively, together with the scoping operator allow us to constrain a node to be leftmost (rightmost) edge in a constituent. For example, the following query returns sentences (`S` labelled nodes) that begin with a noun phrase and end with a verb phrase.

```
//S{[//^NP-->VP$]}
```

LPath also includes an *immediate following sibling* relation. The semantics of this has already been noted with respect to Conditional XPath. Note, the underscore character `_` is used to specify the wildcard, as `*` is used as the kleene star.

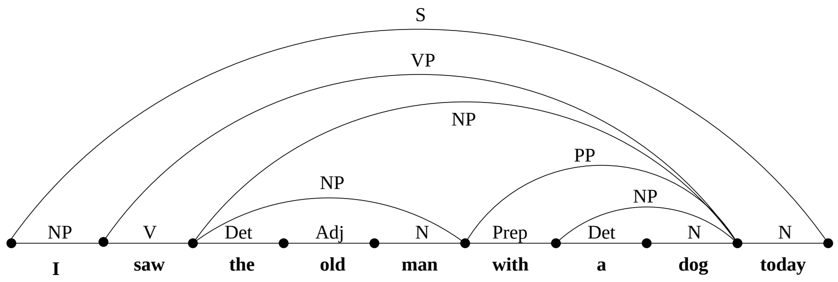
An interesting thing about the LPath interpreter is that it is based on annotation graphs (Bird and Liberman, 2001) rather than the tree signatures we have been examining. The annotation graph representation for the tree shown in Figure 3.1 is shown in Figure 5.4 This representation makes sequential relations between nodes explicit. Computing the *immediate following* relation requires a single join. It also facilitates mapping of queries from LPath to SQL. However, unlike the annotation graph model, hierarchical structure is also represented as the `pid` field in the relational representation of the tree.

Edge alignment, scoping and the *immediate following* relation were discussed in Section 2.5 where we motivated their inclusion as primitives in LPath. As seen in Section 2.1.7, these definitions give LPath the ability to express a range of linguistic tree queries. However, LPath cannot to express closures of any sort or deal with multiple trees. Moreover, even though LPath contains Core XPath, it is not clear where LPath's lies on the hierarchy of XPath languages surveyed in Chapter 3. Nor is it clear what extra expressiveness the LPath operators add to these path-based languages. Perhaps they are just syntactic sugar. The following sections explore this question in depth. This analysis of expressiveness also indicates how LPath can be extended to express closures, and how such an extension might be efficiently implemented.

Notation. The following sections take an incremental approach to investigating the expressiveness of Core XPath and LPath extensions. This involves many distinct languages constructed and related by restrictions on the kleene star and the LPath operators defined above. These languages are identified using the notation in Figure 5.5. Subscripts and superscripts denote the addition of a particular operator. Thus, $\mathcal{L}^{c*}$ denotes LPath extended with the conditional axis. Similarly, $\mathcal{X}^{c*}_{\{\}}$ denotes Conditional XPath extended with the scoping operator (but not `->` or its converse). Note that `=>` and its converse are primitive in all languages in Figure 5.5 except Core XPath, $\mathcal{X}$. Thus, $\mathcal{X}_{\texttt{->\{\}\$}}$ represents Core XPath with `->`, `=>` and their converses, scoping and edge alignment, that is, LPath or $\mathcal{L}$.

left	right	depth	id	pid	name
1	10	1	2	1	S
1	2	2	3	2	NP
1	2	2	4	2	I
2	9	2	5	2	VP
2	3	3	6	5	V
2	3	3	7	5	saw
3	9	3	8	5	NP
3	6	4	9	8	NP
3	4	5	10	9	Det
3	4	5	11	9	the
		...			

(a) Relational Representation



(b) Annotation graph

Figure 5.4: Trees and the annotation graph signature

Languages:			
$\mathcal{X}$	Core XPath	$\mathcal{L}$	LPath
$\mathcal{X}^{c*}$	Conditional XPath	$\mathcal{L}^{c*}$	Conditional LPath
$\mathcal{X}^{*}$	Regular XPath	$\mathcal{L}^{*}$	Regular LPath
Subscripts:			
`{}`	Scoping	`$`	Edge alignment
`->`	Axes `->` and `<-` (also `<=` and `=>` for $\mathcal{X}$)		

Figure 5.5: Notation for XPath languages and LPath operators

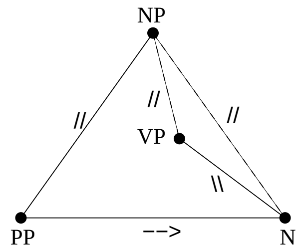


Figure 5.6: Scoping induced cycles: `NP{//PP-->N\\VP}`

5.2 LPath Operators and Core XPath

To better understand the expressiveness of both LPath, $\mathcal{L}$, and Core XPath, $\mathcal{X}$ we first consider whether each of the $\mathcal{L}$ operators can be expressed in $\mathcal{X}$. We find that neither scoping or the *immediate following* axis can be expressed in $\mathcal{X}$. Moreover, the interactions of $\mathcal{L}$ operators, also allow the $\mathcal{L}$ to express queries outside of allows $\mathcal{X}$.

From the semantics given in Figure 5.3 it is easy to see that edge alignment can be expressed in $\mathcal{X}$. We simply note the following equivalences.

`^A` $\equiv$ `A[not <-- _ ]` and `A$` $\equiv$ `A[not --> _ ]`.

The situation is not so simple for the scoping operator. The main reason for this is the scoping operator creates cyclic queries. This is very similar to the *loop* operator that defines looping caterpillar expressions (cf. Section 3.12). Scoping is, however, a somewhat stronger condition. The scoping operator can be thought of as asserting a dominance relation between the scoping node and those defined within the scoping braces. This is illustrated in Figure 5.6. Here, the query `NP{//PP-->NP\\VP}` is drawn as a cyclic graph where edges are labelled with the axes relating pairs of nodes.

The difficulty implementing the scoping operator (like the loop operator) in path-based languages such as $\mathcal{X}$ (or caterpillars) is that they have no memory of previous steps. In general, it is not possible to return to a particular node unless every node in the tree is uniquely labelled. This is clearly not the case for linguistic trees. In order to transform a 'scoped' expression into a $\mathcal{X}$ expression we need to convert cyclic queries into a disjunction of acyclic ones. An algorithm that does exactly this for the positive fragment of $\mathcal{X}$ has been presented by Gottlob et al. (2004). Positive Core XPath, or Positive $\mathcal{X}$, is the set of $\mathcal{X}$ expressions that do not include negation in filter expressions. This allows us to remove the scoping operator from positive $\mathcal{X}$. However, note that positive $\mathcal{X}$ cannot express the edge alignment operators.

Lemma 5.1. *The scoping operator adds no expressiveness to Positive $\mathcal{X}$.*

Proof. This follows directly from the result of Gottlob et al. (2004) mentioned above. Let L be an $\mathcal{L}$ expression that uses the scoping operator. We simply draw the query graph of L adding *descendant* labelled edges between scoping and scoped nodes. Now, the algorithm of Gottlob et al. (2004) results in a disjunction, D of acyclic query trees that is equivalent to the original (cyclic) query. Each query tree in the disjunction has a node x that represents the target node set of of the original expression L. Thus each query tree in D is equivalent to a set of filter expressions, $F = \{F_i\}_{1<i<k}, k \in \mathbf{N}$, based at x. Thus L is equivalent to a $\mathcal{X}$ expression of the form `//_[`A`]` where $A \equiv \bigvee F_i$. □

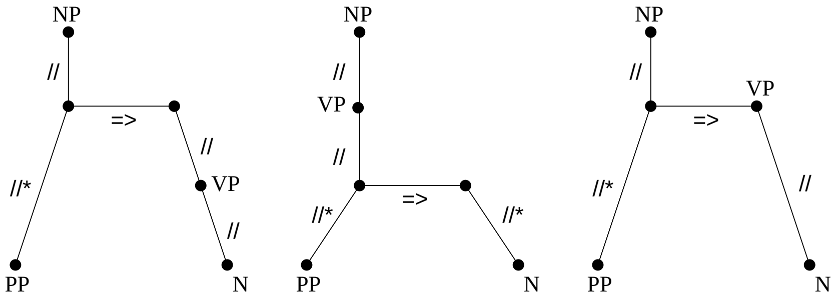


Figure 5.7: Acyclic version of `NP{//PP-->N \\VP}`

The result of applying transformation on the $\mathcal{L}$ expression `NP{//PP-->N\\VP}` is shown in Figure 5.7. The equivalent $\mathcal{X}$ expression is as follows.

```
//N[\\VP\\_<=\\NP[//*PP] or \\*_<=_[//*PP]\\VP\\NP or \\VP<=_[//*PP]\\NP]
```

Unfortunately, this technique does not extend to $\mathcal{X}$ expressions with negation. Negation within the scoping braces only holds inside in the scoped subtree. This is not a problem for negated paths that do not involve the *ancestor* axis because such paths cannot escape the scoped subtree. However, the effects of negation and scoping on the *ancestor* axis give the following lemma.

Lemma 5.2. *$\mathcal{X}_{\{\}}$ is strictly more expressive than $\mathcal{X}$.*

Proof. Consider the $\mathcal{L}$ expression `//B/A{//A[not (\\_[not .A])]}`. This is a `/`-path conforming to the regular expression $\mathrm{B}^{+}\mathrm{A}$. Now, Marx and de Rijke (2004) has shown that all $\mathcal{X}$ query can be expressed at most two variables in FO_{tree} extended with *child* and *immediate following sibling*. However, the regular expression above cannot be expressed in this signature in less than three variables (as noted in Marx (2005b) using the result of Kamp (1968)). □

A similar linguistic example is the $\mathcal{L}$ query `NP{//VP[not \\PP]}`. This expression selects VPs that are dominated by NPs with no PP occurring in between. We can express this using a conditional axis as `NP(/_[not PP])*/VP`. However, as we have seen, this closure cannot be expressed in $\mathcal{X}$.

The other additions of $\mathcal{L}$ to $\mathcal{X}$ are the one step horizontal axes: *immediate following sibling*, *immediate following* and their converses. Note, the full names of these axes will be used interchangeably with their short $\mathcal{L}$ form (eg. `->`, `=>` respectively) from now on.

Lemma 5.3. *The* immediate following sibling *and* immediate following *axes, and their converses, cannot be expressed in $\mathcal{X}$.*

Proof. This follows from Marx (2005b). Again, $\mathcal{X}$ answer sets are definable as formulae in FO^2_{tree} (with child) whereas these two axes require three variables. □

Thus, $\mathcal{L}$ is more expressiveness than $\mathcal{X}$.

We proceed to incrementally add $\mathcal{L}$ operators to $\mathcal{X}$ as primitives to gain a further idea of the expressiveness they provide. Consider $\mathcal{X}_{\text{->}}$. Since edge alignment can be expressed in $\mathcal{X}$, $\mathcal{X}_{\text{->}}$ is equivalent to $\mathcal{L}$ without the scoping operator. To start with we can extend the result in relating $\mathcal{X}$ to FO^2_{tree} (Marx, 2005b) so that it applies to $\mathcal{X}_{\text{->}}$.

Lemma 5.4. *The filter expressions, hence answer sets, of $\mathcal{X}_{\texttt{->}}$ expressions are exactly the sets definable by first-order formulae $\varphi(x)$ in one free variable and at most two variables in signature $\tau_l = \{\texttt{/}, \texttt{//}, \texttt{->}, \texttt{==>}, P_i\}$ where P_i is a countable set of unary predicates.*

Proof. (Sketch) We can basically take a direct copy of the proof sketched in Marx (2005b) for $\mathcal{X}$. The mapping to first-order logic over the signature τ_l follows easily from the standard translation from modal logic (Blackburn et al., 2001).

The proof of the reverse direction, is basically copied from Etessami et al. (1997, Theorem 1) who proved a similar result for unary linear temporal logic (unary-TL). This proof works recursively on the structure of the formula. If $\varphi(x)$ is atomic we can simply output the corresponding axis or test. If $\varphi(x)$ is composite, Etessami et al. (1997) show that we can recursively build an equivalent unary-TL (in our case $\mathcal{X}_{\texttt{->}}$) formula. This is done by reformulating $\varphi(x)$ as a propositional formula over atomic relations and unary τ_l formula with smaller quantifier depth than $\varphi(x)$. Recursion the proceeds over subformulae of $\varphi(x)$ by introducing a case distinction determining when subformulae hold. A number of mutually exclusive cases, order types, need to be distinguished. Let χ be an order type and $\psi(y)$ a $FO^2_{\tau_l}$ formula. Let ψ' denote the translation of $\psi(y)$ in $\mathcal{X}_{\texttt{->}}$. The translation then boils down to the fact that we can translate formulae of the form $\exists y(\chi(x,y) \wedge \psi(y)$ as follows:

$\chi(x,y)$	$\exists y(\chi(x,y) \wedge \psi(y))$
$x = y$	ψ'
$xR(\texttt{/})y$	`/_[`ψ'`]`
$yR(\texttt{/})x$	`\_[`ψ'`]`
$xR(\texttt{->})y$	`->_[`ψ'`]`
$yR(\texttt{<-})x$	`<-_[`ψ'`]`
$xR(\texttt{//})y \wedge \neg xR(\texttt{/})y$	`//_[`ψ'`]`
$yR(\texttt{\\})x \wedge \neg xR(\texttt{\}})y$	`\\_[`ψ'`]`
$xR(\texttt{==>})y \wedge \neg xR(\texttt{->})y$	`==>_[`ψ'`]`
$yR(\texttt{<==})x \wedge \neg xR(\texttt{<-})y$	`<==_[`ψ'`]`

□

One point to consider is whether or not the scoping operator is redundant in $\mathcal{X}_{\texttt{->}}$. That is, whether $\mathcal{X}_{\texttt{->}}$ is expressively equivalent to $\mathcal{L}$. The proof of Lemma 5.2 immediate implies that this is not the case.

Lemma 5.5. *$\mathcal{L}$ is strictly more expressive than $\mathcal{X}_{\texttt{->}}$.*

Proof. (Sketch) The additional axes (`=>` and `->`) express sequential relations and so do not give $\mathcal{X}_{\texttt{->}}$ any more ability to express queries of the form `//B/A{//A[not (\\_[not .A])]}`. Thus, the scoping operator is not expressible in $\mathcal{X}_{\texttt{->}}$. □

In fact, the interaction of $\mathcal{L}$ operators results in queries that require other closures that are inexpressible in $\mathcal{X}$. First, consider the interaction of the scoping and edge alignment operators. As noted previously, this allows us to express subtree edge alignment. The query `//S{[//^NP-->VP$]}` is equivalent to constraining the `NP` (`VP`) to be a leftmost (rightmost) descendant of the `S`. This requires us to be able to state that every node on the `/`-path between the `S` and `NP` has no left sibling. That is, the conditional axis which is inexpressible in $\mathcal{X}$.

Second, consider the scoping operator and the *immediate following* axis. This axis allows the current context node to move outside of the scoped subtree. This is demonstrated in Figure 5.8. Consider a situation where we wish to find verb phrases (VP) containing a noun phrase

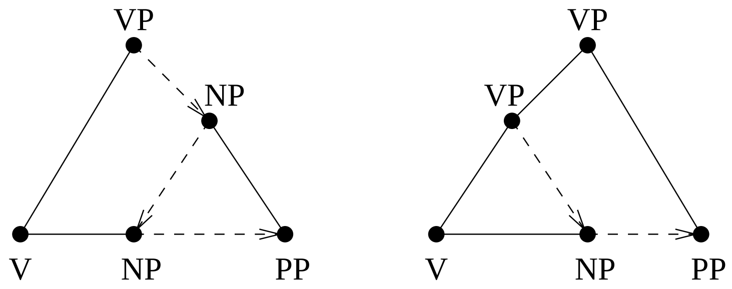


Figure 5.8: Scoping and immediate following. Dashed lines indicate the path of the query.

(NP) immediately followed by a prepositional phrase (PP). That is `VP{//NP->PP}`. From the point of view of the NP labelled node there is no way to tell if an immediate following PP is dominated by the same VP originally being tested. In order to constrain `->` to be within a subtree, we need to phrase this constraint using other axes.

Note, this is can be done for the *following* axis, `-->`. This axis can be alternatively defined as:

`-->`$t[F] \equiv$ `\\*_==>_//*`$t[F]$.

Now, the only chance that we may leave the scope is if the 'ancestor' part of the expression takes us above the scoping node. Thus as long as we constrain how far up the ancestor is chosen, we are assured of staying within the scope. As noted previously, the cycle-removing algorithm of Gottlob et al. (2004) basically enumerates the possible positions such an ancestor can take.

Although `->` has a similar form to its closure `-->` it requires further constraints that are inexpressible in $\mathcal{X}$. Specifically we need to be able to identify ancestors that are rightmost and descendants that are leftmost. This is much the same as subtree edge alignment. As in the previous example, these constraints cannot be expressed in $\mathcal{X}$. Importantly, this means that the only way we can represent the immediate following relation is with the primitive. Without some sort of memory device there is no way to force this primitive to stay within a scope. In a first-order formulae such a memory device would come in the form of extra variables. Thus the scoping operator adds variables to $\mathcal{X}$ query in a restricted form.

Putting all this together gives a clear picture of the expressiveness required to implement $\mathcal{L}$ operators using members of the XPath family of languages surveyed in Section 3. It is clear that the scoping, the immediate following, and immediate following sibling axes are more than syntactic sugar in the context of $\mathcal{X}$. The interaction between all three $\mathcal{L}$ operators as well as negation indicate that $\mathcal{L}$ requires some of expressiveness of the conditional axes. The next section looks at the effect of these operators in the setting of Conditional XPath.

5.3 LPath operators and Conditional XPath

The first thing to notice in moving to Conditional XPath ($\mathcal{X}^{c*}$) is that the immediate following relation is expressible.

`->` $\equiv$ `([not(=> _)]\)* => (/[not(<= _)])*`

Since $\mathcal{X}$ is contained in $\mathcal{X}^{c*}$, the definitions of edge alignment operators carry over from $\mathcal{X}$. Once again, the scoping operator requires more work. However, the ability to express this follows immediately from the completeness of conditional XPath path queries by Marx (2005b) noted in Theorem 3.8. Consider now $\mathcal{X}^{c*}$ with the scoping operator added to its syntax, $\mathcal{X}^{c*}_{\{\}}$.

Lemma 5.6. *$\mathcal{X}^{c*}$ is as expressive as $\mathcal{X}^{c*}_{\{\}}$.*

Proof. Any $\mathcal{X}^{c*}_{\{\}}$ with scoping braces deleted is just a $\mathcal{X}^{c*}$ expression. Therefore, we can write any $\mathcal{X}^{c*}_{\{\}}$ expression ignoring any scoping braces into a FO_{tree} formula $\phi(x, y)$. Let z be the variable we wish to intepreted as the scoping node and let $w_0, \ldots, w_k$ be variables to be witnessed by nodes in the scoped location path. That is, $z, w_0, \ldots, w_k$ occur in $\phi(x, y)$. Moreover, if they are free variables they must be one of x or y. For each w_i we conjoin the clause $descendant(z, w_i)$ in the appropriate variable scope. Since this does not change the number of free variables this has an equivalent $\mathcal{X}^{c*}$ expression. □

Thus all the $\mathcal{L}$ operators are expressible in $\mathcal{X}^{c*}$. Moreover, the first-order completeness of $\mathcal{X}^{c*}$ means that the interactions between $\mathcal{L}$ operators add no extra expressiveness. However, there is no Kleene star in $\mathcal{L}$. As noted previously, we can still express some downward transitive closures by virtue of the scoping operator. However, none of the $\mathcal{L}$ operators promote sequential closures so the reverse case is not true. This gives us the following theorem.

Theorem 5.7. *$\mathcal{L}$ is strictly contained in $\mathcal{X}^{c*}$. As a result, $\mathcal{L}$ is less expressive than FO_{tree}.*

That is, the expressiveness of LPath ($\mathcal{L}$) lies strictly between Core XPath ($\mathcal{X}$) and Conditional XPath ($\mathcal{X}^{c*}$). Thus $\mathcal{L}$ is a new member of the XPath family of languages, and not a notational variant of one of the existing languages.

5.4 The Expressiveness of Conditional LPath

The obvious question now is whether $\mathcal{L}$ with conditional axis, $\mathcal{L}^{c*}$, is any more expressive than $\mathcal{X}^{c*}$. The main point of difference between the two languages is the status of the the *immediate following* (`->`) axis. This is elevated to rank of primitive axis in $\mathcal{L}$. Unfortunately, treating `->` as a primitive axis does not necessarily give it the same properties as the one step axes of $\mathcal{X}^{c*}$. Consider the relations R_i where $i \in \{$`=>`, `<=`, `/`, `\`$\}$. If $(x, y) \in R_i^*$ there is a unique i-path between x and y. This is not the case for the immediate following axis. The `->` is a many-to-many mapping and its converse, the *immediate preceding* relation `<-`, is as well.

Notice that, the `/` relation is also one-to-many but the the converse, `\`, is one-to-one. This one-to-many property of `<-` that even though, `-->` is the closure of `->`, we cannot use the usual translation of conditional axis to FO_{tree}. For example, consider the $\mathcal{L}^{c*}$ expression `B(->A)+`. We might try to express this (incorrectly) in FO_{tree} as

$$\begin{aligned} &following(x, y) \wedge B(x) \wedge A(y) \wedge \\ &\qquad \forall z(following(x, z) \wedge following(z, y) \rightarrow A(z)). \end{aligned}$$

The possibility of multiple `->`-paths between x and y makes this formula too strong a statement. The original $\mathcal{L}^{c*}$ expression only requires the *existence* of an `->`-path between nodes x and y where x has label B and the other nodes are labelled A. The formula above requires all `->`-paths to have this property.

However, we can derive a FO_{tree} formula that is equivalent to the set defined by `B(->A)+` as follows. Let x and y be nodes such that $xR_{\texttt{-->}}y$, and x and y are labelled B and A respectively. Let v be the first common ancestor of x and y and denote the subtree rooted at v as T_v. We also need the following to hold. For each leaf l between x and y there is at least one node z

labelled A on the each \-path from l to v. This set of z nodes gives us the required ->-path. A first-order formula that expresses this is as follows,

$$\begin{aligned} imf_{BA^+}(x,y) \equiv & following(x,y) \wedge B(x) \wedge A(y) \wedge \\ & \forall z((following(x,z) \wedge following(z,y) \wedge leaf(z) \\ & \rightarrow \exists w((z = w \wedge A(w)) \\ & \vee (ancestor(z,w) \wedge A(w) \wedge \neg ancestor(w,x)))). \end{aligned}$$

We can easily let A and B represent location paths instead of labels in the the formula above. So this formula can easily be modified to deal with the conditional -> axis in general. This means that all $\mathcal{L}^{c*}$ expressions without scoping braces can be expressed in FO_{tree}. As in Lemma 5.6, we can add the scoping operator without leaving FO_{tree}. $\mathcal{L}^{c*}$ clearly contains $\mathcal{X}^{c*}$ so we have the following equivalence:

Theorem 5.8. *$\mathcal{L}^{c*}$ has the same expressiveness as $\mathcal{X}^{c*}$. As a result, $\mathcal{L}^{c*}$ path sets are first-order complete over the signature FO_{tree}*

In fact, we can find an equivalent $\mathcal{X}^{c*}$ expression for the conditional immediate following axis using the fact that $\mathcal{X}^{c*}$ is closed under intersection and complementation (Marx, 2005b). First note that the formula $imf_{BA^+}(x,y)$, hence the $\mathcal{L}^{c*}$ query `//B(->A)+`, is equivalent to the following:

$$\begin{aligned} imf_{BA^+}(x,y) \equiv & following(x,y) \wedge B(x) \wedge A(y) \wedge \\ & \neg\exists z((following(x,z) \wedge following(z,y) \wedge \\ & leaf(z) \wedge \neg\exists w((z = w \wedge A(w)) \\ & \vee (ancestor(z,w) \wedge A(w) \wedge \neg ancestor(w,x)))) \end{aligned}$$

We can write this using intersections and complements of $\mathcal{X}^{c*}$ path well-formed formulae. Let

$$\phi(x,y) \equiv (?B/ancestor/(child?\neg A)^{+}/?leaf) \cap following.$$

Now we can write an expression equivalent to `//B(->A)+`

$$imf_{BA^+}(x,y) \equiv (?B/following?A) \cap \overline{\phi/following}.$$

Along with the proof, Marx (2005b, Theorem 2) provides a method for finding the complement of any $\mathcal{X}^{c*}$ path set (cf. the discussion in Section 3.10. Thus, we have a concrete method for translating $\mathcal{L}^{c*}$ expressions into $\mathcal{X}^{c*}$.

5.5 Conditional LPath as Linguistic Tree Query Language

$\mathcal{L}^{c*}$ is capable of expressing a large range of linguistic tree queries. It can certainly express all of the basic subtree matching queries outlined in the Chapter 2's requirements analysis (cf. Section 2.5.1). $\mathcal{L}^{c*}$'s relationship to $\mathcal{X}^{c*}$ gives it a sound theoretical basis. $\mathcal{L}^{c*}$ path-sets are first-order complete with respect to FO_{tree}. $\mathcal{L}^{c*}$, again through $\mathcal{X}^{c*}$, has a clear relationship with the propositional tense logic for trees, $\mathcal{L}_P$ (cf. Section 3.2). This allows reasoning about the query language in the framework of both PDL and temporal logic. The relationship to PDL and its model checkers immediately shows that the data complexity of both $\mathcal{L}$ and $\mathcal{L}^{c*}$ is linear in the size of the tree. This gives also us powerful set of tools for reasoning about expressiveness as well as a base in linguistic theory.

The only other current linguistic treebank query language with this level of expressiveness is fsq (cf. Section 2.1.3. However, fsq only allows boolean queries. Moreover, $\mathcal{L}^{c*}$'s path-based syntax is much more intuitive and more closely aligned to actual descriptions of structure in the linguistics literature (Palm, 1999). However, there is still a price to pay for choosing this path-based variable-free approach over the variables and predicates of classical first-order logic.

The major advantage of the classical approach of fsq is that variables can be used to identify specific nodes throughout a query. The scoping operator accounts for cases where there is a need to identify the root of a particular subtree, the scoping node, at every step in a path expression. However, although $\mathcal{L}^{c*}$ is first-order complete, it is not always clear how a first-order formula can be converted into a (variable-free) $\mathcal{L}^{c*}$ expression.

First-order completeness tells us that the following query is expressible: *Find the first common ancestor of noun phrases immediately followed by a verb phrase.* This is a variant on sample query 5 previously seen in the linguistic query language survey (Section 2.1). This can be expressed as a unary FO_{tree} query, $\varphi(x)$ as follows.

$$\begin{aligned}\varphi(x) =& \exists y\, \exists z\, (descendant(x, y) \wedge descendant(x, z) \wedge NP(y) \wedge VP(z) \\ & \wedge following(y, z) \\ & \wedge \neg\exists w\, (following(y, w) \wedge following(w, z)) \\ & \wedge \neg\exists z'\, (descendant(x, z') \wedge descendant(z', y) \wedge descendant(z', z))).\end{aligned}$$

However, even with the *immediate following* axis and the scoping operator it is not very obvious how this can be expressed in $\mathcal{L}^{c*}$. Note that the following query is incorrect,

```
//_[{//NP->VP} and not(//_{//NP->VP})].
```

This is because each `NP` (or `VP`) may refer to completely different nodes. We can express this query by using the $\mathcal{X}^{c*}$ definition of the *immediate following* relation rather than the primitive `->` axis.

```
//_[/_[(NP or (/_[not(=>_)])*/NP[not(=>_)) and
        => (VP or (/_[not(<=_)])*/VP[not(<=_)])]
```

However, this is a very different approach to representing this sort of query than suggested by the FO_{tree} formula above.

These problems are inherent to path-based, variable-free languages. To a great extent, they are separate from the problem of determining the types of closures needed for linguistic tree query. As discussed previously, the restricted kleene star used in $\mathcal{L}^{c*}$ has already been found sufficient for defining principles of Government and Binding (cf. Section 3.3). The limited form of closure available in $\mathcal{L}^{c*}$ is also adequate for expressing closures discussed in Section 2.5.2.

As an example of how sequential closures are expressed consider the following query: *Find words consisting of consonant-vowel-consonant sequences.*[2] Let words, consonants and vowels be represented by the labels `W`, `C`, and `V` respectively. We can express this query in $\mathcal{L}^{c*}$ as follows.

```
//W{[/^C(->C)*(->V)+(->C)+_$]}
```

Here, the `->` axis allows us to capture the case where where the consonants and vowels may not all be at the same depth. Moreover, the scoping operator provides a convenient way of specifying subtree edge alignment. This allows us to specify completely what a node dominates.

A more hierarchical closure was also suggested in Section 2.5.2. We can describe `NP` labelled nodes that conform to the simple grammar fragment, `NP` $\rightarrow$ `Adj NP`, `NP` $\rightarrow$ `N` as

[2]This is slightly harder version of the query in Cassidy (2002)

```
/NP[({/^Adj=>NP$})*/N]
```

In general, we can represent sets of context-free production rules where rules are not mutually recursive. For example, consider the following production rules

$$\begin{aligned} \mathtt{A} &\rightarrow \mathtt{C\ B\ C} \\ \mathtt{B} &\rightarrow \mathtt{C\ A\ C} \end{aligned}$$

This basically requires us to represent a path of alternating B and A labelled nodes of unbounded length. This type of closure requires at least the expressive power of Regular XPath, $\mathcal{X}^*$.

The addition of the *immediate following* and *immediate following sibling* axes completes the set of $\mathcal{X}$ axes for navigating trees. In $\mathcal{L}^{c*}$ each axis has a corresponding one-step axis. Rogers (1994) notes that there are only a small number of relations that make sense in trees. The $\mathcal{L}^{c*}$ axis set accounts for both hierarchical, sequential and sibling orderings on unranked ordered trees. Moreover, the non-counting principle in linguistics (cf. Section 3.3) suggests that there is no linguistic motivation for including bounded relations of more than distance one. As such, there do not appear to be any such (unconditional) relations that do not appear in the $\mathcal{L}^{c*}$ axis set. Thus, $\mathcal{L}^{c*}$ appears to have the complete set of axes necessary for linguistic *tree* query.

However, linguistic structures are often not true trees at all. For example, both the Penn and the TIGER treebanks make use of secondary edges (cf. Section 1.2). These are not accounted for in $\mathcal{L}^{c*}$. However, they could be supported with additional axes. This amounts to allowing nodes to have more than one parent. This relaxation of tree axioms also allows querying of intersecting hierarchies. However, this will almost certainly require the addition of a type system like that already used in the Emu query language and NiteQL.

The crossing branches that appear in the TIGER treebank cause more of a problem especially for interpreting the *immediately following* axis. However, this problem could be solved by replacing current sequential axes with separate axes for structural and temporal precedence. An important topic for further investigation is how well theoretical results for trees generalize to these more general structures.

5.6 LPath Operators and Regular XPath

The next logical step would be to extend $\mathcal{L}$ with full regular expressions over axes (Regular LPath, $\mathcal{L}^*$). Such a query language would, of course, be capable of doing all the things $\mathcal{L}^{c*}$ can do. It would also allow us to nest Kleene stars. For example, we can define words consisting of consonant/vowel sequences satisfying the the regular expression $\mathtt{(CV^+C)^+}$ as follows.

```
//W{[/^C(->V)+->C(->C(->V)->C)*_$]}
```

Similarly, we can define regular paths from the root to the leaves.

However, it is very unlikely that a *linguistic* tree query language needs this much expressiveness. $\mathcal{L}^*$ filter expressions (hence $\mathcal{L}^*$ answer sets) are at least as expressive as the language $\mathcal{L}_K$ (PDL over trees, cf. Section 3.2). PDL allows counting. On the other hand, the non-counting nature of linguistic structure has already been discussed in Section 3.3.

It is still interesting to look at the affect of adding $\mathcal{L}$ operators to Regular XPath, $\mathcal{X}^*$. Clearly, $\mathcal{X}^*$ contains $\mathcal{L}^{c*}$ due to the equivalence of $\mathcal{L}^{c*}$ equivalence with $\mathcal{X}^{c*}$. This means tree edge alignment and the immediate following axis (and its converse) are expressible. However, it is not clear how the scoping operator can be expressed. $\mathcal{X}^*$ suffers from the same lack of memory as the fragments we have already looked at.

The unrestrained kleene star can appear within scoping braces. That is, theoretically an unbounded number of nodes must be accounted for within a scope. This means we cannot use algorithms like that presented by Gottlob et al. (2004) for removing cycles in queries. As mentioned previously (cf. Section 3.2), there is still no logical characterization of the $\mathcal{L}_K$ with respect to classical logics. This means we cannot use an equivalent logic to provide theoretical results like that of Lemma 5.6 for $\mathcal{X}^{c*}$.

On the other hand, the trees that we deal with are always finite. Given the maximum number of nodes in a tree we can remove the kleene star. This can be done by transforming kleene star expressions into a finite disjunction of paths. For example, if the number of nodes in a tree is 6:

$$\texttt{(/a/b)+} \equiv \texttt{/a/b} \cup \texttt{/a/b/a/b} \cup \texttt{/a/b/a/b/a/b}$$

Once the kleene star is removed we have reduced the original $\mathcal{L}^*$ expression to a materially equivalent $\mathcal{L}^{c*}$ expression whence we can express the scoping operator.

5.7 Caterpillars, Looping and Linguistic Query

If Regular XPath ($\mathcal{X}^*$) provides too much expressiveness for linguistic tree query, then caterpillar expressions certainly do as well. However, they do give us an example of how linguistic query differs from the general tree query problem. Caterpillar expressions were developed to aid typesetters not linguists. Thus, they seem to reflect a different set of priorities than the other query languages discussed so far. For example, an ability to express even or odd path lengths is important for document layout (Brüggemann-Klein and Wood, 1999). These types of queries suggest the full power of the kleene star. Once again it is not clear that this is necessary for our purposes.

In terms of LPath ($\mathcal{L}$) operators, caterpillar expressions for the *immediate following* and tree edge alignment are much the same as for $\mathcal{X}^{c*}$. For example, $(up/isLast)^*/root$ expresses right alignment. Although by the time this takes us to the root of the tree, there is no way to know what node we started at. However, as in $\mathcal{X}^*$, the extra expressiveness of the unrestrained Kleene star means a translation of the scoping operator into this formalism is unforthcoming.

Looping caterpillars contain FO_{tree} (Gorris and Marx, 2005, Theorem 2.9) and so are strictly more expressive than $\mathcal{L}^{c*}$. Hence tree edge alignment is expressible as a test. In fact the loop operator seems very similar to the scoping operator. As we have seen, the scoping operator implicitly adds cycles to queries. The loop operator also gives us a way of expressing cyclic queries. This suggests the loop operator as a solution to the scoping problem. In some instances the loop operator can act as a scoping device. For example, `S[{//NP-->VP}]` can be successfully translated to

$$(S/child^+/NP/following/VP/(up)^+)^{loop}.$$

However, consider the $\mathcal{L}$ expression `S[{//NP-->VP<--PP}]`. The following looping caterpillar expression might seem adequate,

$$(S/child^+/NP/following/VP/preceding/PP/(up)^+)^{loop}.$$

However, since only the last node is required to be in the scope, this translation is incorrect. Of course, we can rearrange this query so that there is no chance of jumping out of and into a scope. However, this still does not allow us to express queries where the scoped expression is not a node test. Thus, scoping is as difficult operator to implement here as in any of path based formalisms seen so far. At this stage, it is unclear whether $\mathcal{L}$'s scoping operator adds any expressiveness to (looping) caterpillar expressions.

τ_{md}	$\mathcal{L}$
root	`._[not \]`
leaf	`._[not /]`
σ	$.\sigma$
firstchild	`/_[not <=]`
nextsibling	`=>`
lastsibling	`_[not =>]`

Figure 5.9: Relations of τ_{md} and equivalent $\mathcal{L}$ expressions.

Looping caterpillars present a powerful algebraic formalism that includes Conditional LPath ($\mathcal{L}^{c*}$). However, while looping caterpillars are certainly expressive enough to express the subtree match requirements set out in Section 2.5, as with $\mathcal{X}^*$, it is unlikely that this level of expressiveness is required in a linguistic tree query language. Moreover, unlike $\mathcal{X}^*$, this extra expressiveness is offset by extra difficulty expressing negation and node tests. The language of caterpillar expressions is a one-sorted. As a result, the intuitive representations of node tests and negation of two-sorted languages (such as XPath) are missing. Although looping caterpillars present a useful and well-founded general query formalism, it is not the best choice for querying linguistic trees. However, its strong connection to tree walking automata may still provide insights into the computational and expressive properties of linguistic tree query languages.

5.8 LPath Operators in Monadic Datalog

The following sections consider monadic datalog as a linguistic tree query language. Recall that monadic datalog programs are defined over the following signature τ_{md}.

$$\tau_{md} = \langle root,\ leaf,\ \sigma \in \Sigma,\ firstchild,\ nextsibling,\ lastsibling\rangle,$$

where Σ is a countable set of node labels. That is, τ_{md} defines the set of extensional (EDB) predicates available for use. Equivalent $\mathcal{L}$ expressions for each of the relations in $\tau_m d$ is shown in Figure 5.9.

Monadic datalog is expressively equivalent to unary MSO queries over the signature τ (Gottlob and Koch, 2004). This makes monadic datalog a much more expressive query language than any of the XPath variants previously examined. The association with MSO gives us a starting point for showing that LPath operators can be expressed in monadic datalog.

Lemma 5.9. *LPath operators are expressible in monadic datalog.*

Proof. We only need to show that LPath operators are definable in MSO over τ_{md}. Now, both *descendant*, `//`, and *following sibling*, `==>` axes are definable as binary relations in MSO. For example, `==>` is definable as the closure of *nextsibling*. This means, all relations in the signature of FO_{tree} are MSO definable. Thus, all FO_{tree} definable unary queries are expressible in MSO. Both edge alignmnent and the *immediate following* relation are definable FO_{tree} and so are MSO-definable.

Scoping can be expressed in MSO as follows. Consider an MSO expression of φ. φ describes a substructure in a tree. We can define φ to be within the scope of a node z in the same manner as Lemma 5.6. That is, by expressing a descendant relation between each first-order and set variable occuring in φ. This clearly does not require any constructs outside of MSO. $\square$

$$\text{leftalign}(x) \leftarrow \text{root}(x).$$
$$\text{leftalign}(x) \leftarrow \text{leftalign}(x_0), \text{firstchild}(x_0, x).$$

$$\text{rightalign}(x) \leftarrow \text{root}(x).$$
$$\text{rightalign}(x) \leftarrow \text{aux\_rightalign}(x), \text{lastsibling}(x).$$

$$\text{aux\_rightalign}(x) \leftarrow \text{rightalign}(x_0), \text{firstchild}(x_0, x).$$
$$\text{aux\_rightalign}(x) \leftarrow \text{aux\_rightalign}(x_0), \text{nextsibling}(x_0, x).$$

Figure 5.10: Edge alignment in monadic datalog.

This theoretical result does not really give us any practical way of implementing these operators in monadic datalog. However, it is not too hard to write programs that express left and right tree edge alignment. These are shown in Figure 5.10

Gottlob and Koch (2002) present a linear time translation from Core XPath into monadic datalog. This is done via an encoding of Core XPath axes as caterpillar expressions over τ_{md} (cf. Figure 5.11). This connection to caterpillars is used to show that Core XPath expressions have equivalent monadic datalog programs. We can also add the *immediate following* axis in the same manner as the other Core XPath axes. The following τ_{md}-caterpillar expression defines the immediate following axis,

$$\texttt{->P} \equiv (\text{lastsibling} \,.\, \text{nextsibling}^{-1*} \,.\, \text{firstchild}^{-1})^{*} .\, \text{nextsibling} \,.\, \text{firstchild}^{*}$$

As seen previously, these caterpillar expressions can be represented by non-deterministic finite automata. These in turn can be used to generate equivalent monadic datalog programs. For example Figure 5.12 shows an NFA equivalent to the LPath program `//P->_` ('immediately follows a `P`') and the resulting monadic datalog program.

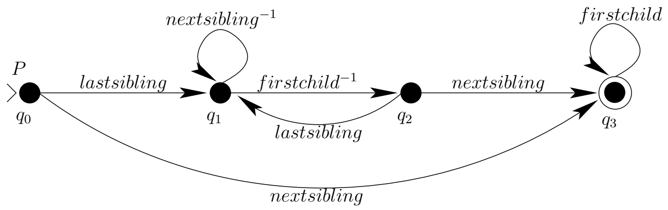
However, scoping is still difficult to express in monadic datalog. This is because each rule in a monadic datalog program can only express local properties. Relations with unbounded distance are expressed using recursion and evaluated with the fixpoint operator. The query evaluation process is much like the labelling algorithm used for PDL model-checking. Once a node has been found to satisfy a particular unary predicate we no longer remember what other nodes where involved in that satisfaction. As previously discussed in Section 4.11 this gives monadic datalog queries something of a path-based, memoryless, flavour. As such, we face much the same problems as were encountered in adding the scoping operator to the XPath family of languages.

5.9 Scoping Regular Expressions Over Cuts

As previously noted, a problem arises when we add regular expressions within the scope braces. This means that an arbitrary number of nodes must be accounted for under the scoping node. This section shows how a restricted form of scoping can be implemented in monadic datalog where unrestricted Kleene star is allowed but only the *immediate following* relation appears

$$\begin{aligned}
\text{self} &:= \epsilon \\
\text{child} &:= \text{firstchild} \,.\, \text{nextsibling}^* \\
\text{parent} &:= \text{child}^{-1} \\
\text{descendant} &:= \text{child}^+ \\
\text{ancestor} &:= \text{descendant}^{-1} \\
\text{descendant_or_self} &:= \text{child}^* \\
\text{ancestor_or_self} &:= \text{descendant_or_self}^{-1} \\
\text{following_sibling} &:= \text{nextsibling}^* \\
\text{preceding_sibling} &:= \text{following_sibling}^{-1} \\
\text{following} &:= \text{ancestor_or_self} \,.\, \text{nextsibling}^+ \,.\, \text{descendant_or_self} \\
\text{preceding} &:= \text{following}^{-1}
\end{aligned}$$

Figure 5.11: Core XPath axes as caterpillar expressions.



$$\begin{aligned}
& imf_P \equiv q_3 \\
& q_3(x) \leftarrow q_3(x_0), \text{firstchild}(x_0, x). \\
& q_3(x) \leftarrow q_2(x_0), \text{nextsibling}(x_0, x). \\
& q_3(x) \leftarrow q_0(x_0), \text{nextsibling}(x_0, x). \\
& q_2(x) \leftarrow q_1(x_0), \text{firstchild}(x, x_0). \\
& q_1(x) \leftarrow q_2(x), \text{lastsibling}(x). \\
& q_1(x) \leftarrow q_1(x_0), \text{nextsibling}(x, x_0). \\
& q_1(x) \leftarrow q_0(x), \text{lastsibling}(x). \\
& q_0(x) \leftarrow P(x),
\end{aligned}$$

Figure 5.12: Immediate following in monadic datalog

within the scoping braces. This is the problem of finding sequences in a cut of the tree that conform to a regular expression. The frontier of the tree is one such cut that linguists may be interested in.

The problem can be tackled by using something along the lines of a DCG parser. We eliminate the use of the *immediate following* axis, which eliminates one way of escaping a scope. This solution uses the fact that we can label nodes with intensional (IDB) predicates. This gives us a form of memory which we can use to progressively build a picture of structure in the tree. This distributed memory demonstrates a key difference between monadic datalog and path-based formalisms like XPath. However, as we will see, it requires us to be able to state new facts.

In general, consider the filter expression {//r} to be evaluated on tree $\mathcal{T}$, where r is a regular expression over the node label alphabet Σ. Assume that the relation between adjacent node labels is *immediate following*. Now, r can be represented with by a NFA, $\mathcal{A}_r$. Let $\mathcal{A}_r = \langle Q, \Sigma, F, q_0, \delta \rangle$, where $Q = q_0, \ldots, q_k$ are the states of $\mathcal{A}_r$. The input alphabet Σ is finite. Let F be the set of final states, q_0 the start state and δ is the transition function. Let $Reach(q)$ be the set of states reachable from state $q \in Q$ in $\mathcal{A}_r$. Let $\mathcal{A}_r(q_i, q_j)$ be the induced NFA that has q_i as start state and q_j as final state. The states of $\mathcal{A}_r(q_i, q_j)$ are defined as $Q_{i,j} = \{q \in Q | q \in Reach(q_i), q_j \in Reach(q)\}$. The transition function $\delta_{i,j}$ is induced by $Q_{i,j}$ and the original transition function δ. Let $L(q_i, q_j)$ be the set of strings generated by $\mathcal{A}_r(q_i, q_j)$.

For each leaf $x \in T$, the domain of $\mathcal{T}$, add $\langle q_i, q_j, 0, 0\rangle(x)$ as a fact in the program, $\mathcal{P}$, if there is some transition, $\delta(q_i, lab(x)) = q_j$. If there is no such transition add $N(x)$ as a fact. Note, this step requires us to do some sort of preprocessing. If node x is not a leaf then, let the children of x be $x_0, \ldots, x_n$. Now, we wish to define a predicate $\langle q_i, q_j, 0, 0\rangle(x)$ that is true at x if,

1. $\langle q_i, q_r, 0, 0\rangle(x_0)$ and $\langle q_s, q_j, 0, 0\rangle(x_n)$ for some $q_i, q_j, q_r, q_s \in Q$. and,

2. If $\langle q_k, q_l, 0, 0\rangle(x_i)$, then $\langle q_l, q_m, 0, 0\rangle(x_{i+1})$ for $0 \leq i < n$, $q_k, q_l, q_m \in Q$.

That is, the subtree rooted at x contains a cut that conforms to some string $l \in L(q_i, q_j)$ for some $q_i, q_j \in Q$. If a node satisfies $\langle q_0, q_F, 0, 0\rangle(x)$ then that string is in $L(\mathcal{A}_R)$ which is what we are looking for.

We must also consider cases where the cut may contain the required string but also other elements on either side. Let $q_F \in F$ be a final state. Define $\langle q_0, q_F, 1, 1\rangle(x)$ to be true if there is a cut in the subtree rooted at x, $\mathcal{T}_x$ that forms a string where ulv where $l \in L(\mathcal{A}_R)$ and $u, v \notin L(\mathcal{A}_R)$ set. Now, define $\langle q_0, q_j, 1, 0\rangle(x)$ to be true at x if there is some cut of T_x of the form lv where $l \in L(q_0, q_j)$ for some $q_j \in Q$. $\langle q_i, q_F, 0, 1\rangle(x)$ is defined similarly. We wish to define a program $\mathcal{P}$ that contains the following rules for each $q_i \in Q$, for all $q_j \in Reach(q_i)$ and $q_k \in Reach(q_i)$ such that $q_j \in Reach(q_k), k \neq j$.

$$\begin{aligned}
\langle q_i, q_j, 0, 0\rangle(x) &\leftarrow \text{firstchild}(x, x_0), \langle q_i, q_j\rangle(x_0)\\
\langle q_i, q_j\rangle(x) &\leftarrow \langle q_i, q_k, 0, 0\rangle(x), \text{nextsibling}(x, x_0), \langle q_k, q_j\rangle(x_0),\\
\langle q_i, q_j\rangle(x) &\leftarrow \langle q_i, q_j, 0, 0\rangle(x), \text{lastsibling}(x).
\end{aligned}$$

$$\begin{aligned}
\langle q_0, q_j, 1, 0\rangle(x) &\leftarrow \text{firstchild}(x, x_0), \mathrm{L}\langle q_0, q_j, 1, 0\rangle(x_0)\\
\mathrm{L}\langle q_0, q_j, 1, 0\rangle(x) &\leftarrow \mathrm{N}(x), \text{nextsibling}(x, x_0), \mathrm{L}\langle q_0, q_j, 1, 0\rangle(x_0)\\
\mathrm{L}\langle q_0, q_j, 1, 0\rangle(x) &\leftarrow \langle q_0, q_j\rangle(x_0)
\end{aligned}$$

$$\begin{aligned}
\langle q_i, q_F, 0, 1\rangle(x) &\leftarrow \text{firstchild}(x, x_0), \mathrm{R}\langle q_i, q_F\rangle(x_0)\\
\mathrm{R}\langle q_i, q_F\rangle(x_0) &\leftarrow \langle q_i, q_k, 0, 0\rangle(x), \text{nextsibling}(x, x_0), \mathrm{R}\langle q_k, q_F\rangle(x_0)\\
\mathrm{R}\langle q_i, q_F\rangle(x_0) &\leftarrow \langle q_i, q_F, 0, 0\rangle(x), \text{nextsibling}(x, x_0), \mathrm{N}(x)
\end{aligned}$$

$$\begin{aligned}
\langle q_0, q_F, 1, 1\rangle(x) &\leftarrow \text{firstchild}(x, x_0), \mathrm{LR}\langle q_0, q_F, 1, 1\rangle(x_0)\\
\mathrm{LR}\langle q_0, q_F, 1, 1\rangle(x) &\leftarrow \mathrm{N}(x), \text{nextsibling}(x, x_0), \mathrm{LR}\langle q_0, q_F, 1, 1\rangle(x_0)\\
\mathrm{LR}\langle q_0, q_F, 1, 1\rangle(x) &\leftarrow \mathrm{R}\langle q_0, q_F\rangle(x_0)
\end{aligned}$$

$$\begin{aligned}
\mathrm{N}(x) &\leftarrow \text{firstchild}(x, x_0), \text{aux_N}(x_0).\\
\text{aux_N}(x) &\leftarrow \mathrm{N}(x), \text{nextsibling}(x, x_0), \text{aux_N}(x_0).\\
\text{aux_N}(x) &\leftarrow \mathrm{N}(x), \text{lastsibling}(x).
\end{aligned}$$

$\langle q_0, q_F, 0, 0\rangle(x), \langle q_0, q_F, 1, 1\rangle(x), \langle q_0, q_F, 0, 1\rangle(x), \langle q_0, q_F, 1, 0\rangle(x)$ are predicates that define nodes that scope the *immediate following* regular expression r.

Note that this does only find the scoping nodes and so is equivalent to expressions where the scoping operator is inside a filter. For example, `S[{//`r`}]`.

5.10 Monadic Datalog an Linguistic Tree Query

Many arguments have already been made that first-order expressiveness is all that is needed in a linguistic tree query language. That is, MSO is too expressive. These arguments apply to monadic datalog as well. On the other hand, monadic datalog presents a very expressive query language with good evaluation complexity. Its equivalence to MSO on trees also provides a strong connection to beautiful results from formal language theory. It is also the most straightforward of the MSO equivalent formalisms to actually write queries. If expressiveness beyond FO_{tree} is found to be required, monadic datalog has much to recommend it.

Part of the reason why monadic datalog is easy to use is it is easy to formulate programs that specify paths. The equivalence of monadic datalog and TMNF-caterpillar expressions means that monadic datalog programs can be viewed as caterpillar expressions with very expressive tests. This makes it reasonably easy to describe, for example, the immediate following relation and tree edge alignment. This connection to caterpillar expressions also highlights some of the drawbacks of monadic datalog. First, negation is not explicit, nor is it straightforward to describe. However, it is clear that negative descriptions of structure are important in linguistic queries. Linguists are unlikely to have strong logic programming backgrounds so this remains a significant disincentive against monadic datalog. Second, monadic datalog programs, like languages in the XPath family, forget where they have been easily. This is due to the path-based nature languages and means that queries requiring subtree scoping are difficult to formulate.

On the other hand, monadic datalog of capable of defining and remembering sets of nodes. The fixpoint evaluation can be seen as a node labelling process. This provides a form of memory distributed throughout the nodes of the tree. This approach was used in the previous section to implement scoped regular expressions over the *immediate following* axis. In fact, many model-checking algorithms take this node labelling approach. For example, the PDL model checkers outlined in Section 3.4 and the boolean attribute grammars in Section 4.9.

Monadic datalog has efficient evaluation mechanism in Arb (Koch, 2003). Moreover Arb has been geared to the problem of streaming tree query and only requires two passes of each tree. Thus, although monadic datalog may be too expressive for our linguistic tree query requirements it may still serve as an intermediary for evaluating a language like Conditional LPath. In this case, an important problem to investigate is the blow-up in program size for linguistic tree queries.

5.11 Conclusion

This chapter investigated the query languages surveyed in the two preceding chapters as linguistic tree query languages. This was done by examining the relationship between these languages and and the linguistically motivated tree query language LPath. Specifically, we examined how the immediate following relation, tree edge alignment and subtree scoping operators of LPath could be expressed in each of these languages. This provided a way of evaluating the ability of these formalisms to express linguistic queries. It also provided a formal foundation for LPath which allowed a number of results on the expressiveness of this language to be derived.

LPath was conceived as the navigational component of XPath (ie. Core XPath) extended with the tree operators above. The analysis of LPath operators shows that they are more than just syntactic sugar. In fact, LPath takes up a new rung on the expressiveness hierarchy strictly between Core and Conditional XPath. Moreover, the comparison with LPath shows that Core XPath is not equipped fulfill all the requirements for linguistic tree query.

On the other hand, all LPath operators can be expressed in Conditional XPath. Conditional LPath, LPath extended with the conditional axis, has the same expressiveness as Conditional XPath. Thus, Conditional XPath provides a fixpoint in expressiveness with respect to LPath. This also provides further evidence that the closures required for linguistic query can be restricted to conditional paths. Moreover, this supports the argument that first-order logic (FO_{tree}) provides enough expressiveness for our linguistic tree query needs. In terms of expressiveness, Conditional LPath appears to be the tree query language we have been looking for.

A number of other formalisms with greater expressiveness were also considered. Regular XPath, looping caterpillars and monadic datalog are all strictly more expressive than Conditional LPath. These languages can certainly specify all the linguistic queries that Conditional LPath can. However, there is no compelling evidence that their extra expressiveness would be used in linguistic queries. Moreover, characterizations of the expressiveness of Regular XPath and looping caterpillars with respect to monadic second-order logic are lacking. As such, it is extra unclear what extra expressiveness these languages actually bring.

On the other hand, monadic datalog can be used to dynamically label nodes. This gives it a form of memory absent from the XPath family and caterpillars that is useful for formulating, for example, scoping queries . It also has a close association with streaming tree query. Thus, monadic datalog may lead the way to efficient implementations of evaluating Conditional LPath. In general, the real efficiency of these languages needs to be investigated further.

Chapter 6

Conclusion and Future Work

The analysis of human communication, in all its forms, increasingly depends on large collections of texts and transcribed recordings. These collections, or corpora, are often richly annotated with structural information. These data sets are extremely large so manual techniques are only successful up to a point. As such, significant effort has recently been invested in automatic techniques for extracting and analyzing these massive data sets. However, further progress on analytical tools is confronted by three major challenges. First, we need the right data model. Second, we need to understand the theoretical foundations of query languages on that data model. Finally, we need to know the expressive requirements for general purpose query language with respect to linguistics. This thesis has address all three of these issues.

Specifically, this thesis has studied formalisms used by linguists and database theorists to describe tree structured data. These are Propositional Dynamic Logic (PDL) and Monadic Second-Order Logic (MSO). These formalisms have been used to reason about a number of tree querying languages and their applicability to the linguistic tree query problem.

The contribution of this thesis has been to identify a set of linguistic tree query requirements and to identify the level of expressiveness needed to implement them. The analysis of the expressiveness of the LPath query language showed that Core XPath is not expressive enough to fulfil basic navigational requirements such as subtree scoping and immediate precedence. On the other hand, the linguistically motivated operators of LPath are all expressible in Conditional XPath. Thus, we have found LPath to be a distinct new language in the hierarchy of XPath languages, expressively between Core and Conditional XPath. Inclusion in this hierarchy provides the formal foundation in which to reason about LPath.

The fact that LPath operators are theoretically expressible in Conditional XPath does not relegate them to syntactic sugar. For example, it is still not clear how scoping queries involving negation can actually be expressed in Conditional XPath. This suggests that Conditional LPath as an extension of both LPath and Conditional XPath that is a better solution to the linguistic tree query problem.

A main focus of this thesis has been how complex closures can be expressed and evaluated in a linguistic tree query language. We have presented strong evidence that the types of closures needed are those expressible with conditional paths. These are exactly the types of closures expressible in Conditional LPath. Thus, in Conditional LPath we have identified a path-based, linguistically oriented language that appears to have the right level of expressiveness for a node-selecting linguistic tree query language. Moreover, we know Conditional LPath answer sets are first-order complete.

Thus, the main result of this study is that the first-order language over trees, FO_{tree} has the required level of expressiveness for a linguistic tree query. This immediately means that linguistic tree query languages should be amenable to efficient implementation via relational database technologies. This also means that the extra precision exhibited by Regular XPath, looping caterpillars, monadic datalog, and MSO in general are not required for linguistic tree query.

Although Conditional LPath does appear to be the tree query language we have been looking for in terms of expressiveness, questions remain about the efficiency of query evaluation. More generally we would expect to take advantage of fact that Conditional LPath is expressively equivalent to Conditional XPath. Conditional XPath can be embedded in a number of other formalisms.

For example, Afanasiev (2003) gives a linear time translation from Conditional XPath to Computational Tree Logic. Queries can then be evaluated using the NuSMV model-checker. Initial investigations with respect to Core XPath have shown it to be competitive with state-of-the art XML Core XPath engines although many optimization still need to be implemented (Afanasiev, 2004). However, it is unclear how exactly the scoping operator can be translated to Conditional XPath. More generally, we need to determine whether or not there is a linear time translation between the two languages.

There are many other approaches to evaluation that should also be investigated. One approach would be to use monadic datalog and Arb. That is, selecting tree automata. We also need to investigate how the current SQL based implementation of LPath can be extended. This is clearly possible given the first-order nature of Conditional LPath.

We also need to consider the fact that the structures that appear in annotated corpora are not always trees. Cross-serial dependencies (scrambling) and secondary edges are not expressible in MSO over trees (ie. $\mathcal{L}^2_{K,P}$) let alone the first-order language FO_{tree}. This is the next major challenge in expressiveness for the problem of querying linguistically annotated corpora.

Secondary edges can be be represented simply by adding a new axis to Conditional LPath (hence a new binary relation to FO_{tree}). This is indeed the approach taken by current treebank query languages such as fsq and TIGERSearch. It is easy enough to extend the signature of a language with a new construct. It remains to be seen how this will effect the formal properties of languages like Conditional LPath. In general, further investigation is necessary into how results for structures with bounded tree-width for MSO extend to a modal approach. Scrambling poses more of a problem as it requires a different definition of immediate precedence. This problem has been investigated by model-theoretic syntacticians (Rogers, 2004; Mönnich et al., 2001). Thus, a next step is to see how mathematical results on these structures can be applied to linguistic tree query languages.

The other type of non-tree structure a linguistic query language needs to be able to deal with are multiple trees. Queries that span multiple trees generally come in two forms: ordered forests and intersecting hierarchies. Neither of these types of queries have been considered in detail in this thesis. However, a solution for ordered forests has already been presented in the fsq query tool. Here, the basic sequential relation between nodes is linear precedence rather than sibling precedence. The precedence relation can easily be extended to the whole treebank rather than just individual trees, allowing ordered forest queries.

One approach to querying intersecting hierarchies is to relax the ‘one parent’ axiom inherent to trees. Thus, querying would be done over ordered acyclic graphs with multiple nodes labelled ‘root’. A problem with this approach with that wildcards may bind to nodes in the wrong tree. However, types can be used to ensure queries are evaluated over the appropriate tree. Types are used in NiteQL and the Emu query language for exactly this purpose. These amount to labelling nodes with multiple features in the formalisms we have reviewed. However, it is not clear how this approach can be incorporated into tree based query evaluation mechanisms (eg. tree automata).

Finally, it is clear that an update language would be of great use in creating and maintaining annotated corpora. Query languages for XML, such as XQuery and XSLT, have traditionally been used to transform tree structure. A similar approach could be taken in the linguistic tree context. In this approach, queries consist of separate pattern and a constructor clauses. We can view the node selecting query languages discussed in this thesis as describing pattern clauses. The next step is to develop a convenient syntax for specifying constructor clauses. We also need efficient algorithms for implementing transformations. A formal understanding of tree transformations should provide great help in this task.

The overriding lesson of this thesis is that a solid theoretical foundation is vital for understanding the linguistic tree query problem. We have found mathematical approaches to linguistics and tree query to be extremely useful for understanding the problem and its possible solutions. Viewing this problem through the lens of well understood formalisms has enabled us to determine the proper level of expressiveness for linguistic tree query. It has also opened the door to a variety of tractable evaluation algorithms. This theoretical approach has resulted in a convergence between two seemingly disparate fields of study. Further work in the intersection of linguistics and database theory should also pave the way for theoretically well-founded future work in this area. This, in turn, will lead to better tools for linguistic analysis and data management, and more comprehensive theories of human language.

Bibliography

Abiteboul, S. (1997). Querying semi-structured data. In *ICDT*, volume 6, pages 1–18.

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. (1997). The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88.

Abiteboul, S. and Vianu, V. (1999). Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3):428–452.

Afanasiev, L. (2003). XML query evaluation via CTL model checking. Master's thesis, ILLC Scientific Publications, MoL-2003-07.

Afanasiev, L. (2004). XML query evaluation via CTL symbolic model checking. In *Proceedings of ESSLLI Student Session*.

Afanasiev, L., Blackburn, P., Dimitriou, I., Gaiffe, B., Goris, E., Marx, M., and de Rijke, M. (2005). PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135.

Alechina, N. and Immerman, N. (2000). Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337.

Arnborg, S., Lagergren, J., and Seese, D. (1991). Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340.

Backofen, R., Rogers, J., and Vijay-Shanker, K. (1995). A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information*, 4:5–39.

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. (1994). The World-Wide Web. *Communications of the ACM*, 37(8):76–82.

Berwick, R. C. and Weinberg, A. S. (1984). *The Grammatical Basis of Linguistic Performance : Language Use and Acquisition*, volume 11 of *Current studies in linguistics*. MIT Press, Cambridge, Mass.

Bird, S., Chen, Y., Davidson, S., Lee, H., and Zheng, Y. (2004). LPath: A path language for linguistic trees. Unpublished manuscript.

Bird, S., Chen, Y., Davidson, S., Lee, H., and Zheng, Y. (2005). Extending XPath to support linguistic queries. In *Proceedings of Programming Language Technologies for XML (PLANX)*, pages 35–46, Long Beach, California.

Bird, S. and Liberman, M. (2001). A Formal Framework for Linguistic Annotation. *Speech Communication*, 33(1,2):23–60.

Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal logic.* Cambridge University Press, New York, NY, USA.

Blackburn, P., Gaiffe, B., and Marx, M. (2003). Variable free reasoning on finite trees. In *Proceedings of Mathematics of Language: MOL 8*, Bloomington, Indiana, USA.

Blackburn, P., Meyer-Viol, W., and de Rijke, M. (1996). A proof system for finite trees. In Büning, H. K., editor, *Computer Science Logic*, volume 1092 of *Lecturn Notes in Computer Science*, pages 86–105. Springer.

Bojanczyk, M. and Colcombet, T. (2005). Tree-walking automata do not recognize all regular languages. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 234–243, New York, NY, USA. ACM Press.

Brants, S., Dipper, S., Hansen, S., Lezius, W., and Smith, G. (2002). The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories Sozopol.*

Brüggemann-Klein, A. and Wood, D. (1999). Caterpillars, context, tree automata and tree pattern matching. In Rozenberg, G. and Thomas, W., editors, *Developments in Language Theory, Foundations, Applications, and Perspectives*, pages 270–285. World Scientific.

Buneman, P., Fernandez, M. F., and Suciu, D. (2000). Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110.

Carletta, J., McKelvie, D., Isard, A., Mengel, A., Klein, M., and M
oller, M. (2004). A generic approach to software support for linguistic annotation using XML. In Sampson, G. and McCarthy, D., editors, *Corpus Linguistics: Readings in a Widening Discipline*. Continuum International, London and NY.

Cassidy, S. (2002). XQuery as an Annotation Query Language: a Use Case Analysis. In *Proceedings of LREC 2002, Las Palmas, Spain, May.*

Cassidy, S. and Harrington, J. (2001). Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1-2):61–77.

Chomsky, N. (1963). Formal properties of grammars. In Luce, D., Bush, R., and Galanter, E., editors, *Handbook of Mathematical Psychology*, volume 2, pages 323–418. New York: Wiley and Sons.

Chomsky, N. (1981). *Lectures on Government and Binding.* Foris, Dordecht.

Clark, J. and DeRose, S. (1999). XML Path language (XPath). http://www.w3.org/TR/xpath.

Cleaveland, R. and Steffen, B. (1993). A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147.

Cotton, S. and Bird, S. (2002). An Integrated Framework for Treebanks and Multilayer Annotations. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 1670–1677. ELRA.

Courcelle, B. (1990). Graph rewriting: an algebraic and logic approach. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 193–242. MIT Press.

Cowart, W. (1997). *Experimental Syntax: Applying Objective Methods to Sentence Judgements.* SAGE Publications, London.

Derwing, B. L. (1979). Against autonomous linguistics. In Perry, T. A., editor, *Evidence and Argumentation in Linguistics*, pages 163–189. Walter de Gruyter, Berlin.

Emerson, E. A. (1996). Model checking and the mu-calculus. In Immerman, N. and Kolaitis, P. G., editors, *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop, January 14-17, 1996, Princeton University*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society.

Etessami, K., Vardi, M. Y., and Wilke, T. (1997). First-order logic with two variables and unary temporal logic. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 228–235.

Fagin, R. (1974). Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In *Complexity of Computation, SIAM-AMS Proc. 7*, pages 27–41.

Fernandez, M., Simeon, J., and Wadler, P. (2000). XML Query Languages: Experiences and Exemplars. http://www.w3.org/1999/09/ql/docs/xquery.html.

Frick, M., Grohe, M., and Koch, C. (2003). Query evaluation on compressed trees (extended abstract). In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 188–200. IEEE Computer Society.

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., Pallett, D. S., and Dahlgren, N. L. (1986). *The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus CDROM*. NIST.

Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985). *Generalized Phrase Structure Grammar*. Havard University Press, Cambridge, Ma.

Gécseg, F. and Steinby, M. (1997). Tree languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages: Volume 3 - Beyond Words*, pages 1–68. Springer-Verlag, Berlin.

Gorris, E. and Marx, M. (2005). Looping caterpillars. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, USA, Proceedings*. IEEE Computer Society.

Gottlob, G. and Koch, C. (2002). Monadic queries over tree-structured data. In *LICS '02: Proceedings of the 17tg Annual IEEE Symposium on Logic in Computer Science*, pages 189–202.

Gottlob, G. and Koch, C. (2004). Monadic datalog and the expressive power of languages for web information extraction. *Journal of the ACM*, 51(1):74–113.

Gottlob, G., Koch, C., and Pichler, R. (2003). The complexity of XPath query evaluation. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS*, pages 179–190, San Diego, CA, USA. ACM.

Gottlob, G., Koch, C., and Schulz, K. U. (2004). Conjunctive queries over trees. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database System*, pages 189–200, Paris, France. ACM.

Grädel, E. (2001). Why are modal logics so robustly decidable? In Paun, G., Rozenberg, G., and Salomaa, A., editors, *Current Trends in Theoretical Computer Science, Entering the 21th Century*, pages 393–408. World Scientific.

Grädel, E. (2005). Finite model theory and descriptive complexity. In Grädel, E., Kolaitis, P., Libkin, L., Marx, M., Spencer, J., Vardi, M., Venema, Y., and Weinstein, S., editors, *Finite Model Theory and Its Applications*. Springer-Verlag. To appear.

Graff, D. and Bird, S. (2000). Many uses, many annotations for large speech corpora: Switchboard and TDT as case studies. *CoRR*, cs.CL/0007024.

Harel, D., Kozen, D., and Tiuryn, J. (2002). Dynamic logic. In Gabbay, D. and F.Guenthner, editors, *Handbook of Philosophical Logic, Vol 4., 2nd Edition*, pages 99–217. Kluwer Academic Publishers, Dordrecht.

Heid, U., Voormann, H., Milde, J.-T., Gut, U., Erk, K., and Pado, S. (2004). Querying both time-aligned and hierarchical corpora with NXT search. In *Fourth Language Resources and Evaluation Conference, Lisbon, Portugal*.

Hinrichs, E. W., Bartels, J., Kawata, Y., and Kordoni, V. (2000). The VERBMOBIL Treebanks. In *KONVENS 2000 Sprachkommunikation, ITG-Fachbericht 161*, pages 107–112. VDE Verlag.

Hoeksema, J. and Janda, R. D. (1988). Implications of process-morphology for categorial grammar. In Oehrle, R. T., Bach, E., and Wheeler, D., editors, *Categorial Grammars and Natural Language Structures*. D. Reidel, Dordrecht.

Kamp, J. (1968). *Tense logic and the theory of linear order*. PhD thesis, University of California, Los Angeles.

Kepser, S. (2003). Finite structure query: A tool for querying syntactically annotated corpora. In *EACL 2003: The 10th Conference of the European Chapter of the Association for Computational Linguistics*, pages 179–186.

Kepser, S. (2004). Querying linguistic treebanks with monadic second-order logic in linear time. *Journal of Logic, Language and Information*, 13(4):457–470.

Koch, C. (2003). Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *Proc. VLDB*.

König, E. and Lezius, W. (2001). The TIGER language - a description language for syntax graphs. part 1: User's guidelines. Technical report, University of Stuttgart, Stuttgart, Germany.

Kracht, M. (1997). *Inessential Features*, volume 1328 of *Lecture Notes in Artificial Intelligence*, pages 43–62. Springer.

Kupferman, O., Vardi, M. Y., and Wolper, P. (2000). An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360.

Lai, C. and Bird, S. (2004). Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*, pages 139–146.

Lange, M. (2002). *Games for Modal and Temporal Logics*. PhD thesis, University of Edinburgh.

Libkin, L. (1998). *Elements of Finite Model Theory*. Springer-Verlag.

Libkin, L. and Neven, F. (2003). Logical definability and query languages over unranked trees. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 178–187. IEEE Computer Society.

Manning, C. D. (2003). Probabilistic syntax. In Bod, R., Hay, J., and Jannedy, S., editors, *Probabilistic Linguistics*, chapter 8, pages 289–342. Massachusetts Institute of Technology, Cambridge, Massachusetts.

Marcus, M., Kim, G., Marcinkiewicz, M., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The penn treebank: Annotating predicate argument structure. In *ARPA Human Language Technology Workshop*.

Marx, M. (2004a). Conditional XPath, the first order complete XPath dialect. In Deutsch, A., editor, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 13–22. ACM.

Marx, M. (2004b). XPath with conditional axis relations. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 477–494, Heraklion, Crete, Greece. Springer.

Marx, M. (2005a). Conditional XPath. http://www.science.uva.nl/marx/pub/recent/cxpath.pdf.

Marx, M. (2005b). First order paths in ordered trees. In Eiter, T. and Libkin, L., editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 114–128. Springer.

Marx, M. and de Rijke, M. (2004). Semantic characterization of navigational XPath. In *Proceedings of TDM'04 Workshop on XML Databases and Information retrieval*, Twente, The Netherlands.

McKelvie, D., Isard, A., Mengel, A., Moller, M. B., Gross, M., and Klein, M. (2001). The MATE workbench — an annotation tool for XML coded speech corpora. *Speech Communication*, 33(1-2):97–112.

Mönnich, U., Morawietz, F., and Kepser, S. (2001). A regular query for context-sensitive relations. In *IRCS Workshop Linguistic Databases 2001*, pages 187–195.

Murata, M. (2001). Extended path expressions of XML. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 126–137. ACM Press.

Neven, F. (2002). Automata, logic, and XML. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer.

Neven, F. (2005). Attribute grammars for unranked trees as a query language for structured documents. *J. Comput. Syst. Sci.*, 70(2):221–257.

Neven, F. and Schwentick, T. (2000). Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'00), May 15-17, 2000, Dallas, Texas, USA*, pages 145–156. ACM.

Neven, F. and Schwentick, T. (2001). Automata-and logic-based pattern languages for tree-structured data. In *Semantics in Databases, Second International Workshop*, pages 160–178.

Neven, F. and Schwentick, T. (2002). Query automata over finite trees. *Theorertical Computer Science*, 275(1-2):633–674.

Neven, F. and van den Bussche, J. (2002). Expressiveness of structured document query languages based on attribute grammars. *J. ACM*, 49(1):56–100.

Palm, A. (1999). Propostional tense logic for trees. In *Proceedings of the Sixth Meeting on Mathematics of Language: MOL6*, University of Central Florida, Orlando, Florida.

Papadimitriou, C. (2004). *Computational Complexity*. Addison-Wesley.

Prasad, R., Miltsakaki, E., Joshi, A., and Webber, B. (2004). Annotation and data mining of the penn discourse treebank. In *Proceedings of the ACL Workshop on Discourse Annotation Barcelona, Spain*.

Rogers, J. (1994). Studies in the logic of trees with applications to grammar formalisms. Technical Report 95-04, Department of Computer & Information Sciences, University of Delaware, Newark, Delaware.

Rogers, J. (1996). A model-theoretic framework for theories of syntax. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*.

Rogers, J. (2004). On scrambling, another perspective. In *Seventh Workshop on Tree-Adjoining Grammars and Related Formalisms (TAG+7)*.

Rohde, D. (2001). Tgrep2 user manual. http://tedlab.mit.edu/ dr/Tgrep2/tgrep2.pdf.

Sampson, G. (1996). From central embedding to corpus linguistics. In Thomas, J. and Short, M., editors, *Using Corpora for Language Research:Studies in Honour of Geoffrey Leech*, chapter 2, pages 14–26. Longman, London.

Steiner, I. and Kallmeyer, L. (2002). VIQTORYA – a visual query tool for syntactically annotated corpora. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002)*, pages 1704–1711. ELRA.

Suciu, D. (1998). Semistructured data and xml. In *The 5th International Conference of Foundations of Data Organization (FODO'98)*, pages 1–12, Kobe, Japan.

Tarski, A. and Givant, S. (1987). *A formalization of set theory without variables*, volume 41 of *AMS Colloquium Publications*. Amer. Math. Soc., Providence, Rhode Island, USA.

Tiede, H.-J. (2005). Inessential features, ineliminable features, and modal logics for model theoretic sytax. In *FG-MOL 2005: 10th Conference on Formal Grammar and 9th Meeting on Mathematics of Language*. CSLI publications.

Vardi, M. Y. (1996). Why is modal logic so robustly decidable? In Immerman, N. and Kolaitis, P. G., editors, *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop, January 14-17, 1996, Princeton University*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–184. American Mathematical Society.

Vogel, C., Hahn, U., and Branigan, H. (1996). Cross-serial dependencies are not hard to process. In *Proceedings of the 16th conference on Computational linguistics*, pages 157–162, Morristown, NJ, USA. Association for Computational Linguistics.