Natural Language Interface for Geographical Information Systems

Sela Mador-Haim
Natural Language Interface for Geographical Information Systems

Research Thesis

Submitted In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Sela Mador-Haim

Submitted to the Senate of the Technion - Israel Institute of Technology

Iyar 5766 Haifa December 2006
The Research Thesis Was Done Under The Supervision of Assoc. Prof. Yoad Winter in the Department of Computer Science.

Acknowledgment

The generous financial help of the Technion is gratefully acknowledged.
## Contents

1 Introduction .......................................................... 4  
1.1 Motivation .......................................................... 4  
1.2 Contribution ......................................................... 5  
1.3 Previous work ....................................................... 7  
   1.3.1 Geographic Information Systems ............................ 7  
   1.3.2 NLIs to GISs .................................................. 8  
1.4 Structure of this work .............................................. 9

2 A compositional approach for building SQL queries ............ 11  
2.1 Categorial Grammar ................................................ 12  
2.2 $\lambda SQL$ ......................................................... 16  
2.3 Safe $\lambda SQL$ ..................................................... 18  
2.4 From safe $\lambda SQL$ to SQL .................................... 20  
2.5 Safe lexicon ........................................................ 23

3 Semantic issues ......................................................... 25  
3.1 Eigenspace vs. Existential semantics .............................. 25  
3.2 Semantics of $\textit{between}$ ........................................ 27

4 Lexicon architecture ................................................... 29  
4.1 Non-spatial lexical items .......................................... 29  
4.2 Spatial lexical items .............................................. 33  
4.3 Data-dependent lexical items ..................................... 34  
4.4 Type shifting ....................................................... 36

5 Implementation ......................................................... 38  
5.1 Lexicon definition file ............................................ 38  
5.2 Categorial Grammar parser architecture ........................ 39  
5.3 Link to a GIS ....................................................... 40  
5.4 Query optimizations ................................................ 41  
5.5 Example .............................................................. 42
List of Figures

3.1 Examples for between ........................................ 27
4.1 buildings that have most floors .......................... 32
5.1 Query result in QGIS ........................................ 43
Abstract

Geographical Information Systems (GISs) are information systems for processing of data that pertain to spatial or geographic coordinates. Even though GISs are enjoying a rapidly growing users community, the current systems are often difficult to use or require a long learning process. In the GIS literature, it has been well-acknowledged that natural language interfaces (NLIs) would significantly enhance the exploitation of the more complex features of GISs, yet despite the potential value of NLIs for GISs, the work on this subject has so far been rather limited. To the best of our knowledge, existing NLIs for GISs are limited in scope and expressive power and lack the ability to express complex relationships over spatial entities.

In general, the design of NLIs to databases is regarded as a difficult problem since human interaction is often vague, ambiguous or highly contextualized. The approach we take in this work is to avoid many of these problems by designing a system that uses a controlled language for GIS queries. Such controlled languages, which are based on fragments of English, can be designed in a way that minimizes the use of vague, ambiguous and context-dependent expressions, while maintaining the ability to express very complex queries in a language that is a subset of English. We benefit from the fact that GISs are a closed, well-defined domain, which enables us to focus on data independent parts of the language. We show that the addition of data dependent portions can be done semi-automatically and requires very low effort.

Our implementation of an NLI for GISs involves three major tasks: First, we develop a suitable semantic representation for GIS queries, which we call λSQL, and a method to translate natural-language queries via λSQL into spatially-enabled SQL. Second, we define the data independent lexicon, which is based on an extension of the simple applicative categorial grammar (Ajdukiewicz-Bar-Hillel calculus) and λSQL expressions. The definition of λSQL expressions that represent the semantics of spatial relations (especially prepositions) in accordance with the intuitive understanding of such relations involved tackling certain aspects of spatial prepositions that where never dealt with before. The third task is the development of methods to add
the data dependent portion of the lexicon with minimal effort, including an automatic tool that generates lexical entries from the actual geographical database in use.

Furthermore, we implemented the NLI presented in this work as a fully working demo prototype that accepts natural language queries and presents the results graphically using a GIS called QGIS.
List of abbreviations

CCG    Combinatorial Categorial Grammar
CG     Categorial Grammar
DB     Database
DBMS   Database Management System
FOL    First Order Logic
GIS    Geographical Information System
GQT    Generalized Quantifier Theory
GUI    Graphical User Interface
NL     Natural Language
NLDBI  Natural Language Database Interface
NLI    Natural Language Interface
NP     Noun Phrase
OGC    Open Geospatial Consortium
Chapter 1

Introduction

1.1 Motivation

A Geographical Information System (GIS) is an information system for processing of data that pertain to spatial or geographic coordinates. A GIS consists of a database with specific attributes for spatial data, as well as a set of operations for using the data [30]. GISs are widely used to access and analyze spatial data in a wide variety of applications. While the community of GIS users is growing rapidly, the current systems are often difficult to use or require a long learning process [29].

A GIS has two major query interfaces: a graphical user interface (GUI) and a text-based query interface. A GUI consists of an interactive spatial selection interface, where one defines the selection condition by pointing at or drawing spatial objects on the screen display, after having indicated the spatial data layer(s) from which to select features [8]. Although the graphical interface is very useful, in many situations the textual interface is the more suitable or even the only choice of expressing GIS queries [36]. Currently, most of the GIS query languages are human-defined formal languages (e.g. subsets or variations of SQL [36]). Such formal query languages cause difficulties in learning and using the language for most typical users of such systems, which are not experts in computer-science. As a result, GIS users often interact with the system in a cumbersome and non-efficient way. In order to bypass the complications arising from the use of available SQL-based query languages of GISs, a form-based interface [2] is introduced into most systems in order to allow the users to invoke prepared queries by filling-in values in a form. Such interfaces allow the user to invoke a pre-identified set of commonly used queries, but limit the user only to a small set of prepared queries.
In the GIS literature [36, 37, 13, 21] it has been well-acknowledged that natural language interfaces (NLIs) would significantly enhance the exploitation of the more complex features of GISs. Yet, despite the potential value of NLIs for GISs, the work on this subject has so far been rather limited [37]. To the best of our knowledge, existing NLIs for GISs are limited in scope and expressive power and lack the ability to express complex relationships over spatial entities.

In general, the design of NLIs to databases is regarded as a difficult problem since human interaction is often vague, ambiguous or highly contextualized [36, 1]. The approach we take in this thesis is to avoid many of these problems by designing a system that uses a controlled language for GIS queries. Such controlled languages [24, 27] can be designed in a way that minimizes the use of vague, ambiguous and context-dependent expressions, while maintaining the ability to express very complex queries in a language that is a subset of English. We benefit from the fact that GISs are a closed, well-defined domain, which enables us to focus on data independent parts of the language, especially English prepositions and other spatial relations. We show that the addition of data dependent expressions (e.g. proper names) can be done semi-automatically and requires very low effort.

Our implementation of an NLI for GISs involves three major tasks: First, we develop a suitable semantic representation for GIS queries, which we call \( \lambda \text{SQL} \), and a method to translate natural-language queries via \( \lambda \text{SQL} \) into spatially-enabled SQL. Second, the definition the data independent lexicon, which was based on simple applicative categorial grammar (Ajdukiewicz-Bar-Hillel calculus) and \( \lambda \text{SQL} \) expressions. The definition of \( \lambda \text{SQL} \) expressions that represent the semantics of spatial relations (especially prepositions) in accordance with the intuitive understanding of such relations involved tackling certain aspects of spatial prepositions that where never dealt with before. The third task is the development of methods to add the data dependent portion of the lexicon with minimal effort, including an automatic tool that generates lexical entries from the actual geographical database in use.

1.2 Contribution

This thesis introduces an NLI to GISs that receives queries in a controlled language that is a fragment of English, translates these queries into spatial extension of SQL based on an industry standard, and presents the results graphically using existing GIS software. On a theoretical level, this work presents applicative semantics for spatial expressions in GIS context and deals with several semantic issues that where highlighted by this work. The
main contributions are:

**An application-oriented semantics for spatial expressions**

- A technique for constructing SQL queries compositionally, using a new intermediate representation called $\lambda$SQL. The use of $\lambda$SQL enables the NLI to handle complex queries using an extensible controlled language.

- Development of a controlled language for GIS queries, based on Ajdukiewicz-Bar-Hillel calculus and $\lambda$SQL, that enables expression of wide range of queries in a fragment of English, including definition of a lexicon for the data-independent core of the language.

- An examination of the behavior of prepositions in the context of sets of objects, which leads to a new semantic interpretation for spatial prepositions, which we call *eigenspace semantics*.

- An account of the meaning of some spatial expressions in the case of non-convex objects.

**The design of an NLI for GISs**

- A fully working prototype demo, allowing users to query existing GISs using NL and viewing the results graphically.

- Implementation of a parser for NL queries in C++, based on Ajdukiewicz-Bar-Hillel calculus and $\lambda$SQL.

- Implementation of a translation scheme from $\lambda$SQL into SQL.

- A template mechanism, enabling semi-automatic treatment of the data-independent portion of the lexicon, which allows porting the NLI to new sets of data with minimal effort.

- Optimization of the queries generated by the NLI, in order to enable the database engine to complete queries in reasonable time.

- Linking with an actual GIS, based on an industry standard set by the Open Geospatial Consortium (OGC).
1.3 Previous work

In this section, we give a brief survey of the background literature relevant to the work developed in this thesis. We overview Geographical Information Systems and introduce basic concepts in this field relevant to our work. Further, we survey existing works related to NLIs to GISs.

1.3.1 Geographic Information Systems

Geographic information systems serve as repositories of observations humans make about spatially related objects and their properties [11]. Therefore, GIS data modeling emphasizes spatial representations of the real world [26]. Such representations of geographic phenomena are either discrete or continuous. Discrete phenomena are spatially homogeneous entities with distinct locations and boundaries, such as power poles, highways and buildings. In contrast, continuous phenomena are distributed continuously across space with undetermined boundaries. They are modeled as distributions of single-value geographic variables (called fields), such as temperature, terrain, and soil type [40].

Data models used to represent these geographic phenomena are of two types [8]:

1. **Raster**: raster representations use a set of small and regular grids (or pixels). Each cell contains a single value and every location corresponds to a cell. Raster representations are very useful for describing continuous phenomena.

2. **Vector**: Vector data models use line segments or points, represented by their Cartesian coordinates to define locations. The vector model is extremely useful for describing discrete features, but less useful for describing continuous phenomena.

This work focuses on discrete phenomena represented using vector data model, which is the predominant data model in current GISs.

Spatial data is organized in layers, also known as *feature classes* in GIS literature [23]. Feature classes are defined through their thematic aspects. Each class has a class name or a class label and a list of attributes that gives the thematic attributes of the terrain features belonging to this class. As an example [23] we can define different classes of line features, such as: roads, railroads and rivers. The road may have the attributes: road type, pavement, traffic density, last maintenance operation, responsible authority. The railroads may have the attributes: number of tracks, maximal velocity,
traffic density, etc. The rivers may have the attributes: width, depth, current, maximal ship size, etc.

Database management systems (DBMSs) play a central role in GISs, because they liberate GIS designers from building and maintaining a substantial and complex part of a large software system. By using a DBMS, all the tasks of low-level data management are removed from the programmer’s and end user’s responsibility [11]. Spatial DBMSs are specialized versions or extensions of database management systems, which provide the underlying database technology for GISs and other applications [16]. Most current spatial DBMSs are based on an integrated architecture, extending existing DBMSs with spatial data types (point, poly-line, polygon) and function (overlap, distance, area, length) [35].

A standard for spatial databases was developed by an industry consortium called the Open Geospatial Consortium (OGC) [35]. The OpenGIS standard, developed by OGC, defines an extension to SQL [6] that adds the ability to query regarding spatial entities in the database.

1.3.2 NLIs to GISs

Designing a NLI for GISs is part of the larger problem of designing natural language interfaces to database (NLDBIs). A lot of research have been done, especially in the eighties, in this general field [15, 33, 17], and several commercial NLDBIs have appeared [1]. Due to the development of alternative graphical and form-based interfaces, NLDBIs are much less widespread than once predicted [1]. However, while form-based queries may be sufficient for most simple applications, they do not satisfy the needs of GIS users. Due to the complex and versatile nature of GIS queries, the improvement that NLDBIs may offer to workers with GISs may be higher than with other kinds of DB-based systems in the past. Unfortunately, by the time GISs gained popularity, the interest in NLDBIs has already dwindled down, and as a result, despite the potential value of NLIs for GISs, the work on this subject has been very limited so far [37].

Some works have demonstrated NLIs using a database that contain geographically related data. The databases used in these works, however, lack any actual spatial information (e.g. geometric polygons representing states) and are based on purely tabular data and thus may be regarded as a GIS only in a very loose sense. Zelle and Mooney [41] developed a system called CHILL that implemented a technique for learning shift-reduce parsers from an annotated corpus. In order to showcase CHILL, they created a demo called Geoquery (http://www.cs.utexas.edu/users/ml/geo.html) which enable queries over a knowledge base of 800 facts represented as prolog assertions, such as
“What states border states through which the Mississippi runs”, where the relations runs through and border are represented directly as prolog assertions in the database. This system does not handle spatially represented information and its syntax is limited by the choice of a shift-reduce parser.

Rohil [28] developed and NLI to a GIS of India. Like the previous work, this GIS contained only simple tabular data, such as a list of rivers and the countries they run through. Furthermore, the expressive power of this system is very limited due to a simplistic model of only two types of syntactic categories: entities and relations.

Minock [22] developed a system called STEP that focuses on a technology for describing alternative interpretations for ambiguous queries by paraphrasing those alternatives as unambiguous natural language sentences. A demo for STEP is available on-line (http://www.cs.umu.se/mjm/step/), and enables the user to ask queries concerning geographical entities. Like the two other systems mentioned above, however, this system also lacks any actual spatial information.

GIS literature discusses the use of multi-modal interfaces that incorporate natural language into a GIS user interface. For example, sketch-and-talk [29] processed speech and direct sketch on a map display for querying spatial databases. Likewise, DAVE_G supports speech/gesture interactions with large-screen map displays driven by GIS. All of those interfaces, however, are limited to simple commands and their parameters. GeoDialogue [3] is a dialog-based system that allows human-machine interaction. However, its syntactic analysis is limited to identification of predefined phrases within a sentence, without any actual syntactic analysis.

One additional work on NLI to GISs worth mentioning is a system developed by Xu [39], which is, as far as we know, the only existing NLI that interacts with a general-purpose, commercially-available GIS, namely ArcView. This system uses an intensional knowledge representation system called SNePS in order to analyze user queries and send them to ArcView. However, the interface is limited to perform really basic queries on one single map (i.e. layer) and one single attribute, and therefore does not support general spatial relations.

1.4 Structure of this work

This thesis is structured as follows. In Chapter 2 we discuss our compositional approach for building SQL queries and present \( \lambda SQL \), the intermediate language used in our NLI. In Chapter 3 we discuss semantic issues related to spatial prepositions that are crucial for interfaces to spatial databases.
In Chapter 4 we describe the lexicon of the controlled language developed in this thesis. In Chapter 5 we discuss the implementation of the NLI and Chapter 6 concludes.
Chapter 2

A compositional approach for building SQL queries

SQL is a recursive language in the sense that it allows using an SQL query as a part of an expression inside another SQL query. However, due to its complex syntax, the construction of an SQL query in a compositional way from a query in natural language is not a straightforward task. One way to tackle this problem is by using an intermediate representation [12, 24]. While such an intermediate language avoids the complications of composing SQL queries directly, its downsides are the additional translation phase it requires and the fact that such intermediate languages are usually not as expressive as the target language.

To overcome this dilemma, we introduce an intermediate representation language that we call λSQL, which only adds the necessary “compositional glue” to SQL. As a result, only a simple translation process is necessary to convert λSQL queries into normal SQL syntax. Thus, λSQL is basically the simply typed λ Calculus with the addition of syntactic sugar for SQL-like syntax.

The lexicon used in this thesis is based on simple applicative categorial grammar (Ajdukiewicz-Bar-Hillel calculus [38]). A natural way to derive the semantics of an expression based on the categorial lexicon is by using the λ calculus, which is the main motivation for an intermediate language that combines some first order logic expressions with λ operators. Section 2.1 introduces the types and categories used in this thesis, section 2.2 defines λSQL, and section 2.4 describes the translation scheme from λSQL to SQL.
2.1 Categorial Grammar

In categorial grammar (CG), nearly all grammatical information is contained within the lexicon. Words are assigned categories that may combine with others in a semantically transparent manner, through a small set of rules. Categories may be either atomic elements or complex categories. Complex categories represent semantic functions that specify the canonical linear direction in which they seek their arguments.

The choice of atomic categories in our system is based on a compromise between two, sometime contradictory, considerations. First, we would like to stay as close as possible to standard category systems based on Montague Grammar [9]. On the other hand, pragmatic considerations lead us to stay close to the established ontology of the subject domain, which may sometimes force us to divert a little bit from the standard. The set of primitive categories in our lexicon are:

1. $S$ - Sentence
2. $N$ - Noun. Represents a set of entries in the database
3. $NP$ - Noun phrase. Represents an entry in the database
4. $R$ - Real number
5. $Rs$ - Predicate over real numbers
6. $STR$ - String
7. $G$ - Geometric entity
8. $Gs$ - Predicate over geometric entities

As seen from the list of atomic categories above, there are several non-standard grammatical categories representing different types of entities in our model: $STR$, $R$, $Rs$, $G$ and $Gs$, in addition to the standard categories $S$, $N$ and $NP$. The motivation behind this set of atomic categories will become clearer once we introduce the type system.

Complex categories are constructed from basic categories using the standard slash (/) and backslash (\) operators. Given $A$, a finite set of atomic categories, the set of categories $CAT$ is the smallest set such that:

1. $A \subseteq CAT$
2. $(X/Y), (X\ Y) \in CAT$ if $X, Y \in CAT$
Categories may combine through the two directed elimination rules presented below:

1. $A/B \Rightarrow A$
2. $B \Rightarrow A$

Complex categories can be viewed as functions with the addition of directionality. We can regard a complex syntactic category as a function that receives arguments according to the order in which the elimination rule can be applied and returns a primitive category. For example, the category: $(N \setminus N)/(S \setminus NP)$ can be viewed as a function that receives $(S \setminus NP)$ as its first argument (on its right) and $N$ as a second argument (on its left), and returns the category $N$.

Expressions in $\lambda SQL$ refer to an existing ontology of the GIS domain. Therefore, the semantic types in our system should correspond to such an ontology. As discussed in Section 1.3.1, we use a discrete representation of geographic phenomena, where each discrete object is represented as an entry in the database. Each object features thematic attributes that describe various properties of an object such as road type, street name or a number of floors.

In our system, we use a many-sorted type system, which is based on a set of basic types that reflects the different values for thematic attributes and GIS objects in our system. The types in our system correspond to data types in the GIS database and thus help to enforce the type-correctness of SQL queries that are generated from $\lambda SQL$ expressions. The basic types used in this work are therefore:

- $t$ - truth value.
- $r$ - a set of real numbers (used for both numerical attribute values and measured values such as distance).
- $s$ - string (a value of a textual attribute).
- $g$ - geometrical shape (a set of points in space).
- $e$ - entry in the database (a GIS object).

Each of these primitive types represents another usage within common GIS systems. Except for the $t$ type for the two truth values, the actual values of each type may depend on the implementation details of the GIS. The domain of $r$ is the set of finite precision real numbers of an actual
precision that depends on GIS implementation. The domain of $s$ is the set of all ASCII strings up the maximal length of strings allowed in the GIS implementation. The domain of $g$ is the set of geometrical shapes that are representable in the GIS, for example a set of set of polygons in a vector data model or a set of a raster images, depending on data model used for representing two-dimensional shapes. The domain of $e$ is the set of all GIS entities in all layers in the relevant GIS, that is the set of all houses, lakes, roads etc. in a given GIS.

There is a very tight connection between the CG categories and their types. We define a $Typ$ operator that maps a category into a type. $Typ$ is defined as:

- $Typ(S) = t$
- $Typ(NP) = e$
- $Typ(N) = (et)$
- $Typ(R) = r$
- $Typ(Rs) = (rt)$
- $Typ(STR) = s$
- $Typ(G) = g$
- $Typ(Gs) = (gt)$
- $Typ(A/B) = Typ(A\B) = (Typ(A) Typ(B))$

Note that the type of $NP$ in the context of this work is $e$ and not $(et)t$, which is the usual type for noun phrases according to Generalized Quantifier Theory (GQT) [14]. Essentially, this is a design decision that is based on the intent to keep lexical definitions simple, as demonstrated in Chapter 4. Furthermore, this apparent diversion from GQT is merely a matter of terminology. Later in this thesis (Section 4.4) we introduce the category $GQ$, which is an abbreviation for the complex category $S/(S\NP)$. This complex category fulfills the usual role of $NPs$ in GQT.

So far we merely described the types and categories in our lexicon, based on simple applicative CG. Our lexicon, however, makes use of three extensions to CG: the use of feature structures to eliminate spurious ambiguity and block the formation of unsafe $\lambda$SQL expressions\(^1\), the introduction of

\(^1\)Safe $\lambda$SQL expressions are expressions that could be translated into SQL queries using the translation algorithm defined in this work. Safe $\lambda$SQL is discussed in Section 2.3
empty lexical entries used for type shifting, and finally augmenting the system with other rules in addition to the elimination rules defined above. There are several existing formalisms that provide those features along with many additional features, such as Lambek’s Type Logical Grammar (TLG) and unification grammars. The main motivation for preferring an extended version of simple applicative CG over the use of an alternative, more powerful formalism is that each of the alternative formalisms, while being more powerful, offer features we do not need for queries in our system and come with some significant drawbacks. The main drawback of TLG is that parsing algorithms for TLG are of higher complexity than simple applicative CG, while its additional features would not help us to significantly reduce the number of type shifting operators, as discussed in Section 4.4. The main drawback of unification grammars is a less elegant lexicon.

Feature structures are helpful for enforcing agreement. The traditional method for enforcing agreement is to split primitive categories into subcategories, whereas each subcategory represents a different combination of case, number etc. Such splitting, however, might inflate the lexicon into unmanageable size and hence it is impractical. Unification grammars [19] provide a much more convenient mechanism to handle agreement. Several unification grammars such as Categorial Unification Grammar (CUG) [34] and Unification Categorial Grammar (UCG) [4], combine features of CG with those of unification grammars. Both grammars are quite powerful and flexible, but they mainly offer additional features we do not need in our fragment (such as the ability to handle anaphoric expressions) and on the other hand the need to represent lexical entries as feature structures means the representation of the lexicon would involve complications that are not necessary to tackle with for the purpose of our system.

Instead, the approach we use is the one described in [31], where atomic categories are themselves feature structures that are used to support syntactic phenomena such as case and number agreement, while complex categories and elimination rules are the same as in CG. The feature structures are flat, consisting of a set of key and value pairs. Two atomic categories of the same type unify only if there is no attribute mismatch, i.e. they do not feature different values for the same attribute. Attribute values can be either constants or variables, as in: \( Rs_t=\#u/R_t=\#u \) (The prefix \( \# \) denotes a variable). In case the elimination rule is invoked with \( Rs_t=\#u/R_t=\#u \) and \( R_t=n \), during unification, \( \#u \) is instantiated as \( n \) and the result is \( R_{s,t=n} \).

The main advantage for this approach is that such feature structures can be viewed as merely a more compact representation for subcategory splitting, and thus our grammar is still simple applicative categorial grammar.

In order to describe different semantic behavior of expressions in different
contexts, we employ empty lexical entries, which have a syntactic and semantic contribution, but no lexical content. Such empty lexical entries introduce type shifting operators that are used in our lexicon for several purposes. Section 4.4 will describe some of those type-shifting operators.

The three more rules our system supports in addition to the directional elimination rules are:

- **Forward harmonic composition**: \( X/Y \ Y/Z \Rightarrow X/Z \)
- **Backward harmonic composition**: \( Y\backslash Z \ X\backslash Y \Rightarrow X\backslash Z \)
- **Exchange**: \( (X/Y)\backslash Z \Leftrightarrow (X\backslash Z)/Y \)

The above three rules are subsumed by Lambek’s Type-Logical Grammar [5]. However, in order to keep the parser as efficient as possible and avoid spurious ambiguities as a result of over-generation, we chose to add to the parser only the features that are required for the controlled language introduced in this work, as discussed above.

### 2.2 \(\lambda\)SQL

The intermediate language used in our NLI, which we call \(\lambda\)SQL, consists of First Order Logic (FOL) expressions, with the addition of \(\lambda\) operators. The FOL portion of \(\lambda\)SQL is augmented with the necessary syntactic sugar that gives it an SQL-like syntax, including unary and binary operators as well as dot expressions. Both operators and dot expressions are semantically redundant constructs that are necessary in order to make \(\lambda\)SQL expressions compatible with SQL syntax.

\(\lambda\)SQL is a simply-typed language, which is formally defined below:

**Definition 1.** Given \(T\) - the set of types over the basic types: \(\{t, e, r, s, g\}\), \(V\) - a set of typed variables, \(C\) - a set of typed constant identifiers, \(STR\) - a set of constant strings and \(R\) - a set of real number identifiers, we can define our intermediate language \(L\) in the following way:

Assuming \(\alpha,\beta,\gamma,\delta \in T\) and \(a^\alpha, b^\beta, c^\gamma \in L\), a string \(p^\delta\) is a term in \(L\) of type \(\delta\) if and only if one of the following is true:

1. \(p^\delta \in V\) or \(p^\delta \in C\)
2. \(p^\delta \in STR\) and \(\delta = s\)
3. \(p^\delta \in R\) and \(\delta = r\)
4. \( p^\delta \in \{ o^{a_0} a^\alpha, a^\alpha o^{a_1(b_\beta)} b^\beta \} \) where \( o \in C \)

5. \( p^\delta = f^{a_1(o_2(\ldots o_{a_\delta}))}(a_1^{a_1}, a_2^{a_2}, \ldots a_n^{a_n}) \) where \( f \in V \) or \( f \in C \)

6. \( p^\delta = a^c.d^\delta \) where \( d \in C \) and \( \delta \in t, r, s, g \) (\( d \) is a field name in a DB entry \( a \), viewed as a function from DB entries to field values)

7. \( p^t = (a = b) \) (comparison is allowed between terms of any type. Result is always Boolean)

8. \( p^t \in \{ \exists x^e.a^t, \forall x^e.a^t \} \) where \( x \in V \)

9. \( p^{\gamma x} = \lambda x^\gamma.a^\alpha \) where \( x \in V \)

The set of constants in \( \lambda SQL \) include all SQL operator symbols, SQL built-in functions and names of fields in the database. The set of operators in \( C \) include all SQL unary and binary operators such as comparators (<, >, <=, >=, !=) of type \( r(t) \), Boolean operators \( (AND, OR, NOT) \) and arithmetic operators (+, -, *, /). One additional important operator in \( \lambda SQL \) is the dot operator, as in \( var.fieldname \), where \( var \) is of type \( e \) and \( fieldname \) is a constant function from entries in the database to entities of a basic type (i.e. it is of type \( et, er \) or \( es \)). A dot expression is equivalent to \( fieldname(var) \), a function that returns the value of a field of a given entry.

In the lexicon, each lexical item consists of a syntactic category \( C \) and a corresponding \( \lambda SQL \) expression \( exp \), denoted \( C : exp \). During parsing of natural language queries, the derivation process involves the reduction of a sequence of such \( C : exp \) pairs into a single category and a corresponding \( \lambda SQL \) expression by consecutively applying the rules described in section 2.1 over adjacent items in the sequence. Each time an elimination rule is used during parsing, it results in a function application between the corresponding \( \lambda SQL \) expressions. The composition rules imply a function composition between the two \( \lambda SQL \) expressions, and exchange replaces the order of lambda operators.

1. **Forward elimination**: \( \frac{(A/B) : exp1 B : exp2}{A : exp1(exp2)} \)

2. **Backward elimination**: \( \frac{B : exp2 (A\setminus B) : exp1}{A : exp1(exp2)} \)

3. **Forward harmonic composition**: \( \frac{(A/B) : exp1 (B/C) : \lambda x.exp2(x)}{(A/C) : \lambda x.exp1(exp2(x))} \)

\(^2\)Function application is defined in \( \lambda SQL \) only for identifiers in order to make it easier to keep \( \lambda SQL \) safe, as explained in Section 2.5. We use comma separated list of parameters instead of Currying in order to make the syntax compatible with SQL syntax.
4. **Backward harmonic composition:** \((B\backslash C) : \lambda x.\text{exp2}(x) (A\backslash B) : \text{exp1} \Rightarrow (A\backslash C) : \lambda x.\text{exp1}(\text{exp2}(x))\)

5. **Exchange:** \(((A/B)\backslash C) : \lambda x.\lambda y.\text{exp} \Leftrightarrow ((A\backslash C)/B) : \lambda y.\lambda x.\text{exp}\)

In our framework, we normalize \(\lambda\text{SQL}\) expressions using \(\beta\)-reductions so that \((\lambda x.\text{exp1})(\text{exp2})\) reduces to \(\text{exp1}[x := \text{exp2}]\) (every free occurrence of \(x\) in \(\text{exp1}\) is substituted with \(\text{exp2}\)).

### 2.3 Safe \(\lambda\text{SQL}\)

Not every \(\lambda\text{SQL}\) formula evaluates to an effective SQL query. For example, the expression \(\lambda x.\lambda y.\lambda z.(x * x * x + y * y * y = z * z * z)\) can not be translated into an SQL query. In order to guarantee that we can translate a \(\lambda\text{SQL}\) expression into an SQL query, we define a subset of \(\lambda\text{SQL}\) that we call safe-\(\lambda\text{SQL}\). Our definition of safe-\(\lambda\text{SQL}\) is based upon syntactic constraints of \(\lambda\text{SQL}\). The translation process from \(\lambda\text{SQL}\) into SQL involves translation of \(\lambda\) operators and quantification operators into \(\text{SELECT}\) statements of SQL. Intuitively, an expression such as \(\lambda x^\alpha.\text{exp}\) describes a set of elements of type \(\alpha\) and therefore should be translated into an SQL \(\text{SELECT}\) statement that returns such a set. However, not every such expression can be translated into an effective SQL query (for example, the expressions \(\lambda a.\text{a} > 100\) and \(\lambda x^e.\text{x}\)). Several considerations that affect the definition of safe-\(\lambda\text{SQL}\) are:

1. SQL queries can not return a set of high-level entities such as functions. We only deal with queries that return a set of database entries and queries that perform an aggregated computation over such sets (such as counting the elements in a set or returning the maximal value). Therefore, safe \(\lambda\text{SQL}\) can only feature \(\lambda\) operators over variables of basic types, and the expression following a \(\lambda\) operator must be of type \(t\).

2. A \(\text{SELECT}\) statement in SQL must specify a name of a database table to query. In order to be able to translate an expression such as \(\lambda x^\alpha.\text{exp}\) into an effective SQL query, the expression \(\text{exp}\) must constrain the variable \(x\) to one single table. Such constraints are provided in the form of \(x.\text{layer} = \text{"table - name"}\), where \(\text{layer}\) is used in \(\lambda\text{SQL}\) as a virtual attribute that specifies the name of the table for each database entry.

3. In general, an expression \(\lambda x^\alpha.\text{exp}\) is not guaranteed to be translatable into an SQL query when \(\alpha\) is any type other than \(e\). However, in
case \( x \) is constrained by an expression of the form \( x = \text{exp}_1 \), as in: 
\[
\lambda x^\alpha.\exists y^\beta.(x = \text{exp}_1) \text{ AND exp}_2,
\]
then instead of going over possible values for \( x \), we can go over the existentially quantified variables following \( \lambda x \) (the variable \( y \) in this example) and project them into the values of \( x \) using \( \text{exp}_1 \). Since projection is supported in SQL, such expressions can be handled.

The constraints over \( \lambda SQL \) that define safe-\( \lambda SQL \) reflect the above considerations. In order to define safe-\( \lambda SQL \) formally, we need a definition that would enable us to test whether a variable, \( v \) is constrained by predicate \( c \) in expression \( \text{exp} \).

**Definition 2.** Given two expressions \( \text{exp} \) and \( c \) of type \( t \) and a variable \( v^\alpha \), we say that \( c \) constrains \( v \) in \( \text{exp} \), and denote \( v, c \triangleright \text{exp} \), if and only if:

- \( \text{exp} = c \)
- \( \text{exp} = \text{exp}_1 \text{ AND exp}_2 \) and either \( v, c \triangleright \text{exp}_1 \) or \( v, c \triangleright \text{exp}_2 \)
- \( \text{exp} = \exists x.\text{exp}_1 \) or \( \text{exp} = \forall x.\text{exp}_1 \) and \( v, c \triangleright \text{exp}_1 \) where \( x \neq v \)

Another definition we need is that of sub-expressions of a \( \lambda SQL \) expression.

**Definition 3.** Let \( p \) be a \( \lambda SQL \) expression. \( \text{Sub}(p) \), the set of the sub-expressions of \( p \), is the smallest set such that:

- \( p \in \text{Sub}(p) \)
- If \( (a \circ b) \in \text{Sub}(p) \) and \( a \in C \) then \( a \in \text{Sub}(p) \) and \( b \in \text{Sub}(p) \). If \( (o \ a) \in \text{Sub}(p) \) then \( a \in \text{Sub}(p) \)
- If \( f(a_1, a_2, \ldots a_n) \in \text{Sub}(p) \) where \( f \in V \) or \( f \in C \) then \( \{a_1 \ldots a_n\} \subseteq \text{Sub}(p) \) (note that \( f \) is not a sub-expression)
- If \( (a = b) \in \text{Sub}(p) \) then \( a \in \text{Sub}(p) \) and \( b \in \text{Sub}(p) \)
- If either \( \lambda x.a \) or \( \exists x.a \) or \( \forall x.a \) are in \( \text{Sub}(p) \) where \( x \in V \) then \( a \in \text{Sub}(p) \)

We can now define the safe \( \lambda SQL \) subset:

**Definition 4.** A \( \lambda SQL \) expression \( q \) is safe if and only if \( q \) is of type \( (et) \), and for every \( p^\gamma \) in \( \text{Sub}(q) \) the following conditions hold:

**Safety 1** \( \gamma \in \{e, t, r, s, g, et, tt, rt, st, gt\} \)
Safety2 If $\gamma = (\alpha t)$ then $p = \lambda v^e . \text{exp}$ for some $v \in V$

Safety3 If $p = \lambda x^e . \text{exp}$ or $p = \exists x^e . \text{exp}$ then $x, (x \text{.layer} = \text{str}) \triangleright \text{exp}$ holds for some $\text{str} \in \text{STR}$ and if $x, (x \text{.layer} = \text{str1}) \triangleright \text{exp}$ holds then $\text{str1} = \text{str}$.

Safety4 If $p = \lambda x^e . \text{exp}$ and $\gamma \neq e$ then $\text{exp} = \exists y^e . \text{exp1}$ and $x, (x = \text{exp2}) \triangleright \text{exp1}$ holds for some $\text{exp2}$.

### 2.4 From safe $\lambda$SQL to SQL

In this section we describe an algorithm that translates safe-$\lambda$SQL expressions into SQL queries. The correctness of the algorithm is proved in Appendix B.

The typical syntax of a select SQL-command for querying a database is:

```
SELECT < selectlist > FROM < tablelist > WHERE < whereclause >;
```

The `selectlist` parameter is usually a list of fields to be displayed, but it also allows other expressions such as aggregate functions (e.g. field summation). The `tablelist` parameter is a list of tables to query and `whereclause` is a Boolean expression that restricts the rows in the query.

The theoretical foundation of SQL queries is relational algebra [25]. Three out of the six basic operations defined in relational algebra are relevant for $\lambda$SQL: selection, projection and join (also known as Cartesian product). An additional operation which is part of a latter extension to the relational model [25] that we support as well is aggregation. In relational algebra, a database table, which is called a relation, is described as a set of tuples, where each tuple consists of the fields in that table.

We start by explaining the four operations and their equivalents in SQL and $\lambda$SQL, and then provide an algorithm for translating safe $\lambda$SQL expressions to SQL queries.

A selection is an operation that selects a subset from a set of tuples using a predicate. Selection is expressed in SQL using a `SELECT` command, as in: `SELECT x.* FROM table as x WHERE exp`, where `exp` is the selection predicate. In $\lambda$SQL, the expressions that express selection are of the form $\lambda x^e . \text{exp}$. However, while such expressions can be interpreted as a set of entries, we cannot perform an actual query without knowing which table to

---

As a matter of fact, SQL violates some of the constraints of the relational algebra model [7], but this is not relevant for our discussion.
query. The attribute \textit{layer} is used in $\lambda$SQL expressions to encompass this information, so that in case $x$, $(x.layer = \text{table}) \triangleright exp$ holds we can translate the above $\lambda$SQL expression to a SELECT command over table \textit{table}.

A \textit{projection} is an operation that projects a set of tuples into a set of domain values or another set of tuples. Normally, projection is performed immediately following a selection. Such a combined selection-projection can be expressed using the SQL command: 

\begin{verbatim}
SELECT proj(x) FROM table as x
WHERE exp.
\end{verbatim}

In $\lambda$SQL we can express such a projection using the following pattern: $\lambda y.\exists x.e.(y = proj(x)) \ AND \ exp$.

A \textit{join} is a Cartesian product between two database tables. The result of join between a table $S^{et}$ and a table $Q^{et}$ is the Cartesian product $S \times Q$ of type $e(\{et\})$. A join of two tables can be expressed by the $\lambda$SQL expression $\lambda x.\lambda y.x.layer = S \ AND \ y.layer = Q$. In SQL, a join is expressed by listing more than one table in \textit{tablelist}. In this work, we are interested in the join operation in one particular context: a join of two tables following a projection to the domain of the first table, as in the statement: 

\begin{verbatim}
SELECT x.* FROM table1 as x,table2 as y WHERE exp(x,y).
\end{verbatim}

This SELECT statement finds all the pairs of $x$ and $y$ where $exp(x, y)$ is true and then projects all those pairs to the domain of $x$ (returns the values of $x$ in all the $(x, y)$ pairs), which is equivalent to the $\lambda$SQL expression $\lambda x.\exists y.exp(x, y)$.

An \textit{aggregate} function is a function from a set of values of some basic type to a single value of a (possible different) basic type. An aggregate function is therefore of the type $(\gamma t)\delta$. Aggregation is expressed using an SQL command such as 

\begin{verbatim}
SELECT aggr(exp1(x)) FROM table as x WHERE exp2.
\end{verbatim}

In $\lambda$SQL, aggregate functions look like: $aggr(\lambda x.exp)$, which is usually combined with a projection, as in: $\ max^{(rt)}(\lambda x.\exists y.(x = \ exp1(y)) \ AND \ exp2)$.

The translation algorithm, which is given in full in Appendix A, involves two phases. In the first phase the $\lambda$SQL expression $p$ is normalized by pushing existential quantification operators outwards and eliminating universal quantification operators. Existential quantification operators are pushed by applying the transformation from $exp1 \circ \exists x.exp2$ to $\exists x.(exp1 \circ exp2)$ where $\circ$ is either $\ AND$ or $\ OR$. This transformation is safe due to the use of renaming mechanism that gives each bound variable a unique identifier. As a result, the variable $x$ in $exp1$ does not appear in those expressions. Universal quantification operators are eliminated by converting them into existential quantification operators using the transformation from $\forall x.exp$ to $NOT \ \exists x.(NOT \ exp)$.

In the second phase, the translation algorithm traverses the resulting $\lambda$SQL expression recursively, while applying the following transformations:

\begin{itemize}
  \item **P1** The pattern $\lambda x.(exp \ AND \ (x.layer = \text{table}))$ is translated into $SE-
LECT \( x \cdot \) FROM table WHERE \( \text{exp} \).

**P2** The pattern \( \lambda x \cdot \exists y. (\text{exp AND} (x = \text{exp1}) \ \text{AND} \ (y.\text{layer} = \text{table})) \) is translated into \( \text{SELECT \ exp1 FROM \ table \ WHERE \ \text{exp}} \).

**P3** The pattern \( \text{SELECT \ selection \ FROM \ tablelist \ WHERE} \ \exists x. (\text{exp AND} (x.\text{layer} = \text{table})) \) is transformed into: \( \text{SELECT \ selection \ FROM \ tablelist, \ table \ WHERE \ \text{exp}} \).

**P4** The pattern \( \text{aggr(\text{SELECT \ exp1 \ FROM \ tablelist \ WHERE} \ \text{exp2})} \) is transformed into \( \text{SELECT \ aggr(\text{exp1}) \ FROM \ tablelist, \ table \ WHERE \ \text{exp2}} \).

**P5** The pattern \( \exists x. (\text{exp AND} (x.\text{layer} = \text{table})) \) is transformed into \( \text{EXISTS (SELECT \ x \cdot \ FROM \ table \ WHERE \ \text{exp}}) \).

The \( \lambda \text{SQL} \) expression is successfully translated into a legal SQL command if the topmost operation after translation is a \( \text{SELECT} \) command with no \( \lambda \) or \( \exists \) operators. Furthermore, aggregate functions should be placed in the \( \text{selectlist} \) and are not allowed to receive a \( \text{SELECT} \) query as a parameter. In case a \( \lambda \text{SQL} \) expression is a safe-\( \lambda \text{SQL} \) expression, successful translation is guaranteed, as proved in Appendix B.

As an example for \( \lambda \text{SQL} \), consider the following fragment from our lexicon:

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>( \lambda \text{SQL} ) expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildings</td>
<td>N</td>
<td>( \lambda x. (x.\text{layer} = \text{&quot;building&quot;}) )</td>
</tr>
<tr>
<td>with</td>
<td>N\N\N</td>
<td>( \lambda n1^e. \lambda n2^e. \lambda x^e. (n1(x) \ \text{AND} \ n2(x)) )</td>
</tr>
<tr>
<td>more than</td>
<td>Rs/R</td>
<td>( \lambda n^e. \lambda x^e. (x &gt; n) )</td>
</tr>
<tr>
<td>two</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>floors</td>
<td>N\Rs</td>
<td>( \lambda p^e. \lambda x^e. p(x.\text{floors}^e) )</td>
</tr>
<tr>
<td>highest</td>
<td>N\N</td>
<td>( \lambda n^e. \lambda x^e. (n(x) \ \text{AND} \ (x.\text{height}^e = \text{max}(r) \ (\lambda r^e. \exists y^e. (n(y) \ \text{AND} \ r = y.\text{height}^e)))) )</td>
</tr>
</tbody>
</table>

The natural language expression “buildings with more than two floors” will be parsed into the \( \lambda \text{SQL} \) expression: \( \lambda x^e. (x.\text{layer}^e = \text{"building"} \ \text{AND} \ x.\text{floors}^e > 2) \), which is translated according to P1 into the SQL query:

\[
\text{SELECT} \ x \cdot \ \text{FROM building AS} \ x \ \text{WHERE} \ x.\text{floors} > 2
\]

---

\( ^4 \)In terms of completeness, the pattern P3 is redundant since P5 could be used to handle all \( \exists \) operators. However, it is used to generate a more optimized SQL.

\( ^5 \)The pattern P5 is not triggered in case it is under the immediate scope of a \( \lambda \) or \( \exists \) operator in order to allow P2 and P3 to take precedence over P5.
A bit more complex example is the query “highest buildings”, which is translated into: 

\[ \lambda x^e. (x.\text{layer}^e = \text{“building” AND } x.\text{height}^e = \max(r^e.r.\exists y^e.(y.\text{layer}^e = \text{“building” AND } r = y.\text{height}^e)))) \]

The translation mechanism starts by transforming the sub-expression \( \lambda r^e. \exists y^e. (y.\text{layer}^e = \text{“building” AND } r = y.\text{height}^e) \) according to P2. Next, \( \max \) will trigger P4, and the last step is to apply P1 over the whole expression, which results in:

\[
\text{SELECT } x.* \text{ FROM building AS } x \text{ WHERE } x.\text{floors} = (\text{SELECT max}(y.\text{floors}) \text{ FROM building})
\]

### 2.5 Safe lexicon

The translation process from an English query into \( \lambda SQL \) involves replacing each lexical item in the English query with \( C : exp \) where \( C \) is a syntactic category and \( exp \) is the corresponding \( \lambda SQL \) expression in the lexicon. The sequence \( C_1 : exp_1...C_n : exp_n \) is then simplified into a single expression by repeated invocations of rules of inference. In a controlled query language that is based on \( \lambda SQL \), it is necessary to constrain the lexicon in a way that will guarantee that any well-formed query will be translated into an expression in the safe subset of \( \lambda SQL \). In this section we give certain criteria that are sufficient in order to show that our lexicon is safe.

The requirement that a safe \( \lambda SQL \) expression is of type \((et)\) can be satisfied by making sure that the target syntactic category for queries correspond to type \((et)\). In our case, the corresponding category is \( N \). This is usually sufficient for describing the set of objects we want to select, as in Buildings with more than one floor, which corresponds to category \( N \). Adding English imperative sentences such as Show me buildings with more than one floor is a straightforward task, and it is possible to extend the system further to support interrogative sentences such as Which buildings have more than one floor.

A target syntactic category \( N \), however, is not sufficient in order to guarantee that the query can be translated into an SQL query. For example, roof is a noun. However, assuming we do not have a layer of roofs but only use an attribute that specifies whether certain objects such as buildings have a roof, roofs could not be translated into an SQL query since we could not select roofs directly. On the other hand buildings can be accepted as a query (assuming there is a “buildings” layer), as well as buildings with a roof. In order to make a distinction between nouns that specify layers and nouns that are merely an attribute name, we add the attribute \( t \) to the noun.
category, so that the former case would be \( N_{t=n} \) and the latter \( N_{t=u} \). We accept only queries that correspond to the category \( N_{t=n} \). Note that in case the database is modified so that “roofs” becomes a layer, the category for \( \text{roofs} \) would change to \( N_{t=n} \) and hence a change in the database can affect the syntax.

In order to comply with the safety rules provided in Section 2.2, we need to make several restrictions over the \( \lambda SQL \) expressions in the lexicon. A list of such restrictions that are enforced in our lexicon is provided in Appendix C. In this thesis I will not be able to prove that this set of rules guarantees the safety of the lexicon and I leave this claim open as a hypothesis for further research.
Chapter 3

Semantic issues

While some progress was made in semantic theories of prepositional phrases in recent years [42, 20], certain aspects of spatial linguistic phenomena have not been treated as extensively in the semantic literature, but are nevertheless crucial for interfaces to spatial databases. Two such aspects are treated in our system and are discussed below.

3.1 Eigenspace vs. Existential semantics

While previous work on prepositional semantics mainly dealt with relationships between two distinct objects, GIS queries often correspond to relationships between sets of objects. Consider the following query: “Buildings that are up to 200m from a lake”. In case there is more than one lake, we expect our query to return any building such that there is at least one lake up to 200m from it. In other words, it appears like the query existentially quantifies over the lakes. On the other hand, if we change our query to “Buildings that are at least 200m from a lake”, we would expect the query to return buildings that are over 200m away from all the lakes. The query “buildings that are between 200m and 500m from a lake” has a yet more complex semantics, and should result in any building such that there is at least one lake less than 500m from it and there is no lake less than 200m from it.

The semantics of the above three queries becomes much clearer, however, when instead of interpreting the indefinite “a lake” as an existential quantifier over the lakes in the database, “a lake” is interpreted as the set of all lakes, and distance is measured with respect to the space taken by the union of all lakes. We refer to this kind of interpretation for indefinites as eigenspace semantics. In SQL, the eigenspace of a set of objects is evaluated by using the aggregate function GeomUnion over a set of objects, as in:
SELECT geomunion(x.the_geom) FROM lake AS x;

In our framework, eigenspace semantics is treated by enabling a type-shifting from an indefinite noun-phrase into a special category $G$ used for representing the eigenspace. The $\lambda$SQL expression for this $G/N$ type-shifting is: 

$$\lambda n.\text{geomunion}(\lambda g.\exists x.(n(x) \text{ AND } g = x\text{.the_geom}))$$

where the $\text{the_geom}$ is the attribute for the geometrical data of an object in GIS database. The $\lambda$SQL expression for $\text{from } ((\text{N}\setminus\text{N})\setminus\text{RS})/G$ is then defined as: 

$$\lambda g.\lambda p.\lambda n.\lambda x.((n(\text{x}) \text{ AND } p(\text{distance}(x\text{.the_geom}, g))))$$

Note that while eigenspace semantics is used for spatial prepositions, in the case of other spatial relations that are not expressed using prepositions, such as contains and intersects, an indefinite is treated in the usual way, as an existential quantifier. For example, if we ask about “towns that contain a building with more than 10 floors”, the eigenspace semantics would mean finding a town that contains all buildings with more than one floor, whereas we expect to get any town that contains at least one building with more than 10 floors. One way to achieve the correct semantics in this case is to generate a $\lambda$SQL expression for contains that existentially quantifies over the set of contained objects: 

$$\lambda n_1.\lambda n_2.\lambda x.\exists y.(n_1(y) \text{ AND } n_2(y) \text{ AND } \text{contains(x.the_geom, y.the_geom))}$$

Not every preposition generates eigenspace semantics. For example, in the case of a lake with an island, for the preposition with, eigenspace semantics (a lake that contains the eigenspace of all islands) is clearly not a correct interpretation. The explanation in this case is that with is not a truly locative preposition. On the other hand, it appears like any locative preposition does generate eigenspace semantics. This is true both for prepositions that accept modifiers and prepositions that do not accept modifiers. For example, in the case of the preposition near, a man that is near a house is usually understood not to be inside another house, while the existential interpretation does allow a man to be near one house and inside another.

1The above expression is not the actual lexical item for contains. While it does give us the correct semantics in the case of indefinites, such lexical item would not enable our system to handle other quantifiers, as in contains all buildings. Therefore, the actual lexical entry for verbs such as contains does not provide the existential quantification. The $\lambda$SQL expression for contains is $\lambda y^e.\lambda x^e.\text{(contains(x.the_geom, y.the_geom))}$. However, a type shifting operator, which is described more thoroughly in Section 4.4, converts it into an expression that accepts a generalized quantifier: $\lambda q_1^{(x^e)t}.\lambda x^e.q_1(\lambda y^e.\text{contains (x.the_geom, y.the_geom))}$, and another type shifting operator converts the noun that follows contains into existential quantifier: $\lambda n_1^{(x^e)t}.\lambda n_2^{(x^e)t}.\exists x^e.(n_1(x) \text{ AND } n_2(x))$. 

26
3.2 Semantics of \textit{between}

An additional aspect of spatial relations that was so far ignored in the semantic literature is relations between non-convex objects. A fundamental spatial relation that is quite problematic in the context of non-convex objects is the 3-place relation \textit{between}.

Zwarts and Winter [42] suggest the following definition for \textit{between}: \( A \) is between \( B \) and \( C \) if \( A \subseteq \text{convexHull}(B \cup C) \setminus B \setminus C \), for convex objects in \( A, B \) and \( C \). The problem is that many objects we deal with in the context of GISs are not convex. For example, it could be quite handy to talk about objects between two streets. However, streets are often non-convex shapes. As can be seen in figure 3.1, the convex hull for two streets represented by the solid lines includes areas that do not agree with our understanding of the expression \textit{between the two streets}. In order to overcome this problem, we suggest the following definition:

\textbf{Definition 5.} Let \( X, Y \) and \( Z \) be sets of points. We say that \( X \) is between \( Y \) and \( Z \) if either there is a point \( y \) on the border of \( Y \) such that the shortest line connecting \( y \) to \( Z \) crosses \( X \), but does not cross \( Y \), or there is a point \( z \) on the border of \( Z \) such that the shortest line connecting \( z \) to \( Y \) crosses \( X \), but does not cross \( Z \).

The areas between the streets according to Definition 1 are marked by stripes. As can be seen from the illustration, the new definition is more in agreement with our intuitive understanding of \textit{between}. Note that while the above is a strictly spatial definition of \textit{between}, in some contexts people may
use *between* in sloppier ways (e.g. Beit Shemesh is between Jerusalem and Tel-aviv). In our system, however, we wish to avoid the vagueness of such non-strict usage.

Since the primitives in existing OpenGIS systems are not sufficient to implement Definition 5, however, the actual implementation is still based upon the convex hull definition and not on Definition 5.
Chapter 4

Lexicon architecture

The data independent part of the lexicon is the core of our controlled language. This is the part of the lexicon that involves general logical and spatial operators that do not depend on the actual GIS. By carefully selecting the data-independent lexical items, we are able to express very complex queries while avoiding vagueness and ambiguity problems that often undermine the usability of NLIs.

An important part of our work is the ability to express spatial relations between GIS objects. However, non-spatial lexical items are an important part of the lexicon as well. In the first part of this chapter we describe the non-spatial items in the lexicon. In Section 4.2 we review the spatially-related lexical items. In Section 4.3 we present classes of data-dependent lexical items, and finally in Section 4.4, we review type shifting operators in the lexicon.

4.1 Non-spatial lexical items

Non-spatial lexical items can be partitioned into the following groups:

- Units, such as meters, kilometers, miles, acres. The lexical definitions of these items convert any unit into metric units.

- Numerical relations, such as less than, at least, between n and m. Numerical relations represent a set of real numbers.

- Superlatives: biggest, smallest, most, least. The words most and least can be used to refer to the maximal or minimal value of any numerical field in the database. Other words such as largest and longest are used as abbreviation for “most area” and “most length”.

29
• Boolean connectives: and, or, not

• Determiners: some, every, not

• Directional or case marking prepositions: of (as in length of, area of), from, to

• Other lexical entries: that, which, is, are, with, without, have.

Some examples for the non-spatial lexical items are given below:

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>λSQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>kilometer</td>
<td>$R_{t=dist}/R_{t=num}$</td>
<td>$\lambda x^t.(x \ast 1000)$</td>
</tr>
<tr>
<td>less than</td>
<td>$R_{s=num}/R$</td>
<td>$\lambda n^t.\lambda x^t.(x &lt; n)$</td>
</tr>
<tr>
<td>some</td>
<td>$(S/(S/NP))/N$</td>
<td>$\lambda n_1^t.\lambda n_2^t.(\exists x^e.(n_1(x) \text{ AND } n_2(x)))$</td>
</tr>
<tr>
<td>that</td>
<td>$(N\backslash N)/(S\backslash N)$</td>
<td>$\lambda n_1^t.\lambda n_2^t.\lambda x^e.(n_1(x) \text{ AND } n_2(x))$</td>
</tr>
<tr>
<td>a</td>
<td>$N/N$</td>
<td>$\lambda n^t.\lambda x^e.n(x)$</td>
</tr>
<tr>
<td>have</td>
<td>$(S\backslash NP)/N_{t=u}$</td>
<td>$\lambda n^t.\lambda x^e.n(x)$</td>
</tr>
<tr>
<td>is</td>
<td>$(S\backslash NP)/(N\backslash N)$</td>
<td>$\lambda \text{mod}^{(et)}(et).\text{mod}(\lambda x^e.\text{true})$</td>
</tr>
<tr>
<td>most</td>
<td>$(N\backslash(N/N_{t=u}))/((N_{t=u}\backslash Rs)$</td>
<td>$\lambda \text{attr}^{(et)}(et).\lambda n_{t=0}^{(et)}.\lambda x^e.(\lambda y^e.\text{true}, n) \text{ AND } \text{attr}(\lambda n^t.(n = \max(\lambda n_2^t, \exists y^e.(\lambda z^e.\text{true}, y) \text{ AND } \text{attr}(\lambda n_2^t.(n_2 = n_3)), y))), x))$</td>
</tr>
</tbody>
</table>

Some comments regarding non-spatial lexical items:

• The feature $t$ of categories $R$ and $Rs$ is used to distinguish between pure numbers and measure phrases (distance and area), as is apparent in the lexical entry for kilometer.

• Determiners, such as some and every, get the standard treatment in Generalized Quantifier Theory. The use of category $(S/(S\backslash NP))/N$ will be explained in Section 4.4.

• The article a is defined here as a lexical item that receives a noun and returns a noun, as opposed to the standard treatment of indefinite articles as existential determiners. Such treatment is necessary to allow the interpretation of indefinites using eigenspace semantics, as explained in Section 3.1.

• The dauntingly complex definition for most is an ad-hoc solution to the problem of context in superlatives. For example A building that has the largest antenna involves two stages: first, find the largest antenna in the context of all buildings, and then find a building that features such antenna. A blue building that has the largest antenna, however,
requires finding the largest antenna in the context of blue buildings, which may be smaller than the largest antenna for all buildings. Our ad-hoc solution is to allow superlatives such as most or largest to receive the noun to its left as a parameter, even when the superlative is inside a relative clause.

As an example for most, consider the query the building that have the most floors. The lexical entry for most expects an attribute such as floors, which has the λSQL expression λp^rt.λx^e.p(x.floors) of type (rt)(et) (a function from a set of numeric values into a set of database entries), to its right. The string on its left: the buildings that have the, is parsed into the λSQL λn^et.λx^e.(x.layer =" buildings" AND n(x)), which is of type (et)(et). Finding the building with the most floors involves two queries: first, we need to find the maximal number of floors, n, and then we need to find buildings with n floors. The expression: max(λn_2.∃y^e.(a(λz^e.true, y) AND attr((λn_3.(n_2 = n_3)), y))) is the inner query that finds the maximal number of floors. a(λz^e.true, y) becomes y.layer =" buildings" after substituting a with the corresponding expression, and attr((λn_3.(n_2 = n_3)), y) becomes y.floors = n_2 after substituting attr with the expression for floors, and therefore the inner expression reduces to max(λn_2.∃y^e. (y.layer =" buildings" AND y.floors = n_2)). The complete λSQL expression is λx^e.(x.layer = "buildings" AND x.floors = max(λn_2.∃y^e. (y.layer =" buildings" AND y.floors = n_2))). The full derivation is shown in Figure 4.1.
Figure 4.1: buildings that have most floors

\[
\begin{align*}
&\overset{\text{most}}{\frac{N\setminus(N/N)}{(N\setminus Rs)}} \\
&\lambda x.\lambda x.(a(\lambda y.\text{true}, x)&) \\
&\text{attr}(\lambda n.(n = \max(\lambda n_2.\exists y.(a(\lambda z.\text{true}, y)& \\
&\text{attr}(\lambda n_3.(n_2 = n_3)), y)), x)) \\
&\overset{\text{floors}}{\frac{N\setminus Rs}{N\setminus Rs:}} \\
&\lambda p.\lambda x.p(x.\text{floors}) \\
&\overset{\text{have}}{\frac{(S\setminus NP)/N}{(S\setminus NP)/N:}} \\
&\lambda a.\lambda x.(a(\lambda y.\text{true}, x)&x.\text{floors} = \\
&\overset{\text{FC}}{\max(\lambda n_2.\exists y.(a(\lambda z.\text{true}, y)&(n_2 = y.\text{floors}))))} \\
&\overset{\text{buildings}}{\frac{N\setminus(N/N)}{(N\setminus(N/N)/(S\setminus NP):}} \\
&\lambda n_1.n_1 \\
&\overset{\text{that}}{\frac{(N\setminus(N/N)/(S\setminus NP):}} \\
&\lambda n_1.\lambda n_2.\lambda x.n_1(x)&n_2(x) \\
&\overset{\text{EX}}{(N\setminus(N/N)/(N\setminus(N/N))} \\
&\lambda n_1.\lambda n_2.\lambda x.n_1(x)&n_2(x) \\
&\overset{\text{BE}}{(N\setminus(N/N))} \\
&\lambda n_1.\lambda n_2.\lambda x.n_1(x)&n_2(x) \\
&\overset{\text{BE}}{(N\setminus(N/N))} \\
&\lambda n_1.\lambda n_2.\lambda x.n_1(x)&n_2(x) \\
&\overset{\text{BE}}{(N\setminus(N/N))} \\
&\lambda x.(x.\text{layer} = \text{buildings}) \\
&\overset{\text{BE}}{\lambda n_1.\lambda x.n_1(x)&x.\text{layer} = \text{buildings}} \\
&\overset{\text{BE}}{\lambda x.(x.\text{layer} = \text{buildings}&x.\text{floors} = \\
&\overset{\text{BE}}{\max(\lambda n_2.\exists y.(y.\text{layer} = \text{buildings}&(n_2 = y.\text{floors}))))}}
\end{align*}
\]
4.2 Spatial lexical items

As mentioned before, we wish to select a controlled language that avoids the pitfalls of vagueness and ambiguity. In order to satisfy this requirement, we need to avoid vague qualitative relations such as near, far, almost etc. Another type of relations that need to be avoided are projective relations such as in front of, behind, left and right. The meaning of these prepositions involves context-dependent [18] elements that are hard to handle within a controlled language.

The following groups of spatial relations are included in the lexicon:

- Topological relations, following Egenhofer’s 9-intersection model [10]: in, outside of, borders, overlaps, crosses, contains and intersects. Note that only the first two expressions are prepositions, while the others are verbs.

- Distance: the word from when used to specify exact distance, as in “200m from a lake”.

- Constructors: intersection of, border of and center of. These words are used to refer to spatial entities that do not exist in the database, but can be derived from existing objects. For example, assuming “42nd Street” and “Broadway” are objects in the database, “the intersection of 42nd street and Broadway” can be constructed by intersecting the geometrical representations of the two streets.

- Relative location: north of, south east of and the 3-place relation between are all used to describe the position of one object relative to another object (or objects, as in the case of between).

- Superlatives: closest and furthest are spatially-related superlatives.

Some examples for the spatial lexical items:

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>λSQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>(N\N)/G</td>
<td>λgr^g.λn^et.λx^e.(n(x) AND within(x.the_geom, gr))</td>
</tr>
<tr>
<td>contain</td>
<td>(S\NP)/NP</td>
<td>λy^e.λx^e.(contains(x.the_geom, y.the_geom))</td>
</tr>
<tr>
<td>from</td>
<td>((N\N)/G)\Rs_{t=dist}</td>
<td>λp^et.λgr^g.λn^et.λx^e.(n(x) AND p(distance(x.the_geom, gr)))</td>
</tr>
<tr>
<td>intersection</td>
<td>(Gs\N_{p=and})/N_{p=of}</td>
<td>λ\l_1^et.λ\l_2^et.λgr^g.∃x.∃y.(n_1(x) AND n_2(y) AND intersects(x.the_geom, y.the_geom) AND (gr = intersection(x.the_geom, y.the_geom)))</td>
</tr>
<tr>
<td>north</td>
<td>(N\N)/G_{p=of}</td>
<td>λgr^g.λn^et.λx^e.(n(x) AND (x.the_geom &gt;&gt; gr))</td>
</tr>
</tbody>
</table>

Comments regarding spatial relations:

33
As seen from the above example, prepositions such as in and from are treated differently than verbs such as contain. The former are defined using eigenspace semantics and receives category $G$ (the eigenspace of the noun succeeds it) while the latter acts on $NPs$. The existential semantics of indefinites in the latter case is achieved using type-shifting operators as explained in Section 4.4.

The constructor intersection receives two nouns and returns the set of all possible intersections between entities represented by those two nouns.

The operator $\triangleright\triangleright$ used for north is true if the bounding box of the left operand is above the bounding box of the right operand.

### 4.3 Data-dependent lexical items

Data dependent lexical items are lexical items that refer to specific data inside the database and therefore change from one data set to another. GIS data are divided into separate thematic feature classes or layers, whereby each layer consists of one type of geometrical entities such as buildings, streets or utility poles. For each layer there is usually an associated set of attributes that represent non-spatial data attached to real world geometric objects. These may be Boolean data, numeric data or strings. Examples for such attributes are the number of floors in a building or a street name. A query may refer directly to string values such as street names and therefore such string values should be part of the lexicon as well.

In this work we use templates to handle data-dependent lexical items. The template mechanism enables us to provide both syntactic and semantic information for data-dependent lexical items that share the same syntactic category and similar $\lambda$SQL expression. A template is defined in the lexicon just like any other lexical item. However, as a convention, the lexical name for a template begins with the # sign. Furthermore, identifiers within the $\lambda$SQL expression for a template that begins with the # symbol are template parameters, and are replaced by actual data during template instantiation. Possible template parameters are #layer, which is the name of the relevant layer, #attr, which is the name of the relevant attribute and the lexical name of the template itself (such as #real or #strval), which is replaced with the relevant string.

Templates defined in our lexicon are given below:
<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>λSQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>#layer</td>
<td>(N_t=#layer)</td>
<td>(\lambda x^e.(x.\text{layer} = #\text{layer}))</td>
</tr>
<tr>
<td>#real</td>
<td>(N_t=x, t=#layer \mid R_s)</td>
<td>(\lambda p^r, \lambda x^e.p(x.#\text{real}))</td>
</tr>
<tr>
<td>#realId</td>
<td>(N_t=#layer/R)</td>
<td>(\lambda n^r, \lambda x^e.(x.\text{layer} = #\text{layer})) AND (x.#\text{realId} = n r))</td>
</tr>
<tr>
<td>#area</td>
<td>(N_t=u, t=#layer \mid R_s t=\text{area})</td>
<td>(\lambda p^r, \lambda x^e.p(x.#\text{area}))</td>
</tr>
<tr>
<td>#length</td>
<td>(N_t=u, t=#layer \mid R_s t=\text{dist})</td>
<td>(\lambda p^r, \lambda x^e.p(x.#\text{length}))</td>
</tr>
<tr>
<td>#strval</td>
<td>(N/N{l = #\text{layer}})</td>
<td>(\lambda n, \lambda x.(n(x) \text{ AND } x.#\text{attr like } #\text{strval}))</td>
</tr>
<tr>
<td>#uniqstr</td>
<td>(N)</td>
<td>(\lambda x^e.(x.\text{layer} = #\text{layer AND } (x.#\text{attr like } #\text{uniqstr})))</td>
</tr>
</tbody>
</table>

Two of the templates are used in a fully automatic way:

- The \#strval template is used in lexical items that refer to strings inside the database. Whenever a string in a query is not found in the lexicon, the lexical analyzer searches that string in the database and instantiates the above template with the relevant layer name, attribute name and string value.

- The \#uniqstr template is instantiated for a string that is found only in one layer. For example, if Herzel is a street name, and it does not appear in any other layer, it is not necessary to write Herzel street and simply Herzel will suffice.

The rest of the templates are used in a definition file that instantiates them for the relevant layers and attributes within a layer. No knowledge in λSQL is required in order to edit the template definition file. Examples for such templates:

- The template \#layer is used for lexical items that describe a layer, such as building or streets.

- The template \#real is used for attributes that contain numerical value, such as floors or high

- The template \#realId is used for attributes that contain numerical value which is a unique key, and therefore sufficient to identify the entry, such as zip code or quarter number.
4.4 Type shifting

In addition to lexical entries that are related to English words, our lexicon contains several empty lexical items that are used to convert one syntactic category to another when necessary, while performing the required λSQL operation for such conversions. In most cases the use of type-shifting could have been avoided by supplying several duplicate categories for each lexical entity. However, the use of type-shifting enables us to maintain a more compact lexicon.

The type shifting operators defined in our lexicon are:

<table>
<thead>
<tr>
<th>T/S</th>
<th>Category</th>
<th>λSQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>G/N</td>
<td>(\lambda_{et}.geomunion(\lambda g^g.\exists x^e.((g = x.the_geom) AND n(x))))</td>
</tr>
<tr>
<td>T2</td>
<td>G/Gs</td>
<td>(\lambda_{gs^{gs}}.geomunion(gs))</td>
</tr>
<tr>
<td>T3</td>
<td>(N_{i=}/(N_{i=}\backslash Rs))</td>
<td>(\lambdaattr^{(rt)}(et).\lambda x^e.attr((\lambda nr^{rt}.(nr! = 0)), x))</td>
</tr>
<tr>
<td>T4</td>
<td>GQ/N</td>
<td>(\lambda_{n_1^{et}}.\lambda_{n_2^{et}}.((\exists x^e.((n_1(x) AND n_2(x))))</td>
</tr>
<tr>
<td>T5</td>
<td>((S\backslash NP)/GQ)/((S\backslash NP)/NP))</td>
<td>(\lambda rel^{(et)}(et).\lambda d^{et}.\lambda x^e.(q(\lambda y^e.rel(y, x))))</td>
</tr>
<tr>
<td>T6</td>
<td>((GQ/N)/Rs)</td>
<td>(\lambda pr^{et}.\lambda_{n_1^{et}}.\lambda_{n_2^{et}}.p(count(\lambda x^e.(n_1(x) AND n_2(x)))))</td>
</tr>
<tr>
<td>T7</td>
<td>((N\backslash N)/GQ)/((N\backslash N)/G))</td>
<td>(\lambda_{pps^{(et)}(et)}.\lambda d^{et}.\lambda_{n_1^{et}}.\lambda x^e.q(\lambda y^e.pps(y.the_geom, n, x))))</td>
</tr>
</tbody>
</table>

The type shift operator T1 was discussed already in Section 3.1. The operator T2 receives a set of geometries and returns one geometry by performing geomunion over them. It is used when constructors such as center are used. Operator T3 is used in the case of attributes such as businesses. Recall that while a business is a noun, it is not an entity in our data model, but rather an attribute of a building. Usually we use it with a numeric predicate, as in a building with two businesses. We use T3 to interpret with businesses as with more than zero businesses.

The operator T4 is used to convert a noun into an existential quantifier. We did not want to interpret the determiner a as an existential quantifier due to eigenspace semantics, and therefore we need a type-shift operator that would convert indefinites into existential quantifiers when necessary. The category GQ is not a primitive category but merely an abbreviation for \(S/(S\backslash NP)\). Since the category NP is of type e in our lexicon, we need a more complex category to represent quantifiers.

The operators T5 and T7 are used to enable verbs and prepositions to work with quantifiers. The operator T6 is used to allow numbers to be used as quantifiers as well, and supports queries such as Streets that are north of at least 60 lakes.
Note that the only type shift which is used for type lifting is $T4$, which may have been unnecessary in case Lambek’s Type-Logical Grammar was used. All of the other type shifts introduce new SQL functions into the expression and thus could not become redundant by the use of a more powerful grammar. Therefore, fully supporting Lambek’s calculus would not save much work in this system.
Chapter 5

Implementation

The NLI presented in this thesis was implemented in C++. The implementation includes a parser for a lexicon definition file that provides a convenient mechanism for defining and modifying the lexicon, as well as a bottom-up, right-to-left Categorial Grammar parser, the translation mechanism from \( \lambda SQL \) to SQL and a link to an actual spatial DBMS which is used both for sending queries and for searching strings for template instantiation. The demo software works under both Linux and Windows operating systems.

5.1 Lexicon definition file

The lexicon definition file contains all the definitions that are relevant for the lexicon, including types, categories, constants, variables and the lexicon itself. The parser for the lexicon definition file is implemented using bison/flex.

The definition file includes the following constructs:

- **Type typename** defines a new basic type
- **Type typename = typeexp** defines a complex type, using type expressions such as \( (e, (e, t)) \)
- **cat category(typeexp)** defines a new atomic category
- **cat category = categoryexp** defines a name that abbreviates a complex category. The **categoryexp** looks like \( S/(S\setminus NP) \)
- **var varname(typeexp)** defines a variable of type **typeexp**. Similarly, **const varname(typeexp)** defines a constant. Constant name and type should correspond to constant or function in SQL
• name := λSQL exp defines a λSQL expression. The syntax of λSQL in the definition file uses backslash for λ operators, \E for \exists, \A for \forall, & for AND and || for OR.

• “word” category λSQL exp defines a lexical item

Another definition file used by the NLI is data.conf, which includes definitions for the data dependent lexicon. The data definition file uses a much simpler syntax. It includes a section for each layer, where every section looks like:

```
layer building : building, buildings
{
real floors_n floor, floors;
real num_entr_n entry, entries;
real num_apts_n apartment, apartments;
real num_busn_n business, businesses;
area area area;
max area biggest;
}
```

The header of each section consists of the keyword “layer” followed by the name of the table containing the layer in the database and a list of words which is used to refer to this layer. Inside the curly brackets there is a list of template-name/attribute-name/word-list triplets that are supported for that layer.

### 5.2 Categorial Grammar parser architecture

A categorial grammar parser was implemented as part of this thesis as well. It is written completely in C++ and works fairly fast, which means that on a modern computer the parsing process of a query is fast enough to appear as immediate to the user.

Currently, there is a number of existing Categorial Grammar parsers that where considered for this system. Grail (http://www.labri.fr/perso/moot/grail.html) is a Type Logical Grammar parser written in prolog. It supports Lambek’s calculus and some of its extensions. However, it is relatively slow and its time complexity in the worst case is exponential. Bob Carpenter’s TLG theorem prover (http://www.colloquial.com/tlg/index.html) is written in Prolog as well, but runs much faster than Grail. However, it does not support fully automatic hypothetical reasoning since hypothetical categories should
be supplied manually. As a result, support for rules such as harmonic composition and exchange requires manual intervention. Another existing parser is OpenCCG (http://openccg.sourceforge.net/) which is written in Java and support Combinatory Categorial Grammar [32]. It does not support the exchange rule, and it uses an XML based format to specify the lexicon, which results in a less readable lexicon relatively to other systems.

There were several considerations that led to the decision to build a new parser, instead of using any of the existing ones:

- The choice of a parser for simple applicative categorial grammar over parsers that work on more powerful grammars such as Lambek calculus, which are typically slower and more complex to use.

- The ease of modifying the parser in order to support λSQL, optimizations, templates and other mechanisms which are not present in standard parsers.

- The ease of communicating with a DBMS that features a c/c++ API

The parser is based on a bottom-up right-to-left tabular algorithm, and is built using a software architecture that should enable an easy expansion of it. The base class hierarchy includes Type classes to represent types, Category classes that represent categories, Expr classes that represent λSQL expressions and a Parser and Lexer classes that implement the NL parser and lexical analyzer. The lexical analyzer supports the automatic instantiation mechanism described in Section 4.3. The parser supports all the features described in section 2.1, and implements several optimization techniques such as identifying duplicate parse results, including λSQL expressions that are isomorphic under variable renaming.

The class SQLTranslate performs the translation from λSQL to SQL by recursive traversal of the expression, while applying the translation rules explained in Section 2.4.

5.3 Link to a GIS

In order to test the system and verify that the NLI developed in this thesis is both correct and usable, it was important to link the NLI with an actual GIS. We decided to link the NLI to software which is both free and standard compliant so that the NLI developed here would be easy to port to other GISs.

The spatial DBMS we used as a back-end for our system is PostGIS (http://postgis.refractions.net/). PostGIS is an open-source GIS extension
to the PostgreSQL database engine, which implements the OpenGIS “Simple features specification for SQL” standard [6]. PostGIS basically supplies a set of functions that operate on vector representations, such as a function that calculates distance between polygons. The SQL queries are sent to PostGIS, which generates the result in a the form of a table which is loaded into a GIS front-end that supports PostGIS, such as QGIS (http://www.qgis.org), which is an open-source GIS, available on both Linux and Windows operating systems.

PostGIS is capable of porting .shp files, which is an industry-standard GIS file-format, which enabled us to test the demo system on real life data that represents buildings, street, water bodies and neighborhoods in the city of Haifa.

5.4 Query optimizations

The NLI developed in this thesis supports the use of quite complex NL queries that might be translated into huge SQL queries with several levels of nesting of SELECT commands. In order to make such NLI usable, it may be necessary to optimize the produced SQL queries, so that a query will complete in reasonable time. One way to optimize the queries is by taking advantage of spatial indexing.

The OpenGIS standard [6] specifies a mechanism that allows to index geometric data according to their bounding box. One issue with this indexing mechanism, however, is that the DBMS cannot take advantage of those spatial indexes unless special “bounding box” operators are used within the SQL query. One such operator is the overlapping bounding box operator: &&. This operator returns true if the bounding box of its left operand overlaps the bounding box of its right operand. When the && operator is used within a query, the DBMS is capable of using the spatial indexing mechanism to filter any object whose bounding box does not overlap a given bounding box. It is possible to speed up queries by editing the lexicon and adding the && operator to various spatial relations. For example, the λSQL expression for within becomes \[ \lambda \text{g}.\lambda \text{n}.\lambda \text{x}.(n(x) \text{ AND } x\text{.the_geom}\&\&\text{g}) \text{ AND within}(x\text{.the_geom},g)) \].

While adding the && operator to topological relations is quite trivial, finding the bounding box in the case of from is a little bit trickier. For example, in case we want to find buildings that are between 100m to 500m from a lake, we need to take the bounding box of the lake and expand it in every direction so that all objects up to 500m from the lake would overlap this bounding box. The built-in PostGIS function expand enables us to do such
expansion. However, we encountered a technical difficulty here: while the expand command expects a single numerical value, the distance is provided in the form of a predicate that defines a set of possible values. It was necessary to find the largest distance allowed by the query in order to provide it as a parameter to the `expand` function. We solve this problem by adding a new built-in operator to λSQL called `UBOUND` that gets an expression of type `$rt$` and returns its upper bound by invoking a simple symbolic arithmetic solver that is capable of finding the upper bound of a set of inequalities. For example, the string between 100m and 5km would be parsed into the λSQL expression \( \lambda x. x >= 100 \text{ AND } x/1000 <= 5 \). Applying `UBOUND` to this expression would return the value 5000.

Using the above operators, we can redefine the λSQL expression for `from` as:

\[
\lambda p. \lambda g. \lambda n. \lambda e. (n(x) \text{ AND } (x.the_geom \&\& \text{expand}(g, \text{UBOUND } p)) \text{ AND } p(distance(x.the_geom, g)))
\]

which is modified by adding \((x.the_geom \&\& \text{expand}(g, \text{UBOUND } p))\) to the expression. The new addition enables PostGIS to filter the entries in \( x \) to entries that overlap the bounding box of \( \text{expand}(g, \text{UBOUND } p) \) (a bounding box to \( g \), which is expanded according to the upper bound of \( p \) in every direction).

### 5.5 Example

As an example, the query *Buildings that are up to 500m from the intersection of Elm street and Oak street* is converted into the λSQL expression:

\[
\lambda x_1. (\text{distance}(x_1.the_geom, \text{GeomUnion}(\lambda g. (\exists x_2. (\exists y. (x_2.layer = 'street' \text{ AND } x_2.street_name LIKE 'Elm' \text{ AND } y.layer = 'street' \text{ AND } y.street_name LIKE 'Oak' \text{ AND } \text{intersects(x2.the_geom, y.the_geom))))) <= 500 \text{ AND } x_1.layer = 'building')
\]

Which is translated into the SQL command:

```sql
(SELECT x1.*
FROM building AS x1
WHERE distance(x1.the_geom, (SELECT GeomUnion(intersection(x2.the_geom,y.the_geom)) FROM street AS x2,street AS y
WHERE x2.street_name LIKE 'Elm' AND y.street_name LIKE 'Oak'
AND intersects(x2.the_geom,y.the_geom)) )<=500)
```

The above query generates the result in figure 5.1.
Figure 5.1: Query result in QGIS
Chapter 6

Conclusion

In this thesis we have addressed the problem of developing an interface to GISs that is based on a controlled natural language. We have developed a controlled language for GIS queries that enables expression of a wide range of queries in a fragment of English, developed a technique for constructing SQL queries compositionally using a new intermediate representation called \( \lambda SQL \). We developed a fully working prototype demo that allows users to query existing GISs using NL. Furthermore, we provided a mechanism that allows the NLI to be easily portable to different data sets and provided a mechanism that generates optimized queries. This thesis proved the feasibility of building a usable NLI to GISs with a substantial potential to simplify human-machine interaction in this context.

On a theoretical level, this work exposed several interesting semantic issues concerning spatial expressions, such as the eigenspace semantics of spatial prepositions and the semantics of relations with non-convex shapes in the case of relations such as \textit{between}.

Future work on this subject can be done on several different levels. One future direction is expanding the lexicon further by adding additional constructs such as comparison between attributes, anaphoric expressions and multiple types of questions. Furthermore, empirical study is necessary to evaluate how usable such interfaces are for actual GIS users of varying skills and needs and get some additional data on missing features in the current controlled language. More thorough theoretical study is required regarding semantic issues such as eigenspace and \textit{between} presented here, and finally, the use of \( \lambda SQL \) as an intermediate representation that enables construction of SQL queries compositionally requires further study in order to find such a language that would enable representation of a well defined, substantial subset of SQL.
Appendix A

Translation algorithm from safe $\lambda$SQL to SQL

The translation algorithm works in two phases. In the first phase, the function $\text{norm}$ pushes the quantifiers outwards and eliminates universal quantifiers. In the second phase, the function $\text{l2s}$ maps the resulting safe $\lambda$SQL expression into an SQL query.

Before presenting the translation functions themselves, we need to introduce three functions used in $\text{norm}$ and $\text{l2s}$. The first, $\text{match}(\text{exp}, \text{pattern}, \{p_1,..p_n\})$ is a pattern matching function that receives an expression $\text{exp}$, a pattern $\text{pat}$ and a set of parameters $\{p_1,..p_n\}$. The pattern $\text{pat}$ matches $\text{exp}$ if and only if $\exists p_1...\exists p_n. \text{exp} = \text{pat}(p_1,..p_n)$. In case of a match, the matching values is assigned into parameters $p_1,..p_n$.

The function $\text{applysub}(\text{exp}, \text{func})$ is used to apply program function $\text{func}$ to sub-expressions of $\text{exp}$.

**Definition 6.** Given an expression $\text{exp}$ and program function $\text{func}$, $\text{applysub}(\text{exp}, \text{func})$ is defined by:

- $\text{applysub}((a \circ b), \text{func}) = \text{func}(a) \circ \text{func}(b)$
- $\text{applysub}((o \ a), \text{func}) = o \ \text{func}(a)$
- $\text{applysub}(f(a_1,..a_n), \text{func}) = f(\text{func}(a_1),..\text{func}(a_n))$
- $\text{applysub}((a = b), \text{func}) = (\text{func}(a) = \text{func}(b))$
- $\text{applysub}(\lambda x. a, \text{func}) = \lambda x. \text{func}(a)$
- $\text{applysub}(\exists x. a, \text{func}) = \exists x. \text{func}(a)$
- $\text{applysub}(\forall x. a, \text{func}) = \forall x. \text{func}(a)$
The function \( \text{findremove}(\exp, \pat, \{p_1, \ldots, p_n\}) \) is similar to the operator \( \triangleright \) defined in Section 2.3, but in the case of \( \text{findremove} \), once a pattern \( \pat \) is found inside an \( \exp \), it is removed from \( \exp \) and replaced with \( \true \).

**Algorithm 1** The function \( \text{findremove}(\exp, \pat, \{p_1, \ldots, p_n\}) \)

```plaintext
if match(\exp, \pat, \{p_1, \ldots, p_n\}) then
  return \true
else if match(\exp, \exp_1 \AND \exp_2, \{\exp_1, \exp_2\}) then
  return \text{findremove}(\exp_1, \pat, \{p_1, \ldots, p_n\}) \AND \text{findremove}(\exp_2, \pat, \{p_1, \ldots, p_n\})
else if match(\exp, \exists x.\exp_1, \{x, \exp_1\}) then
  return \exists x.\text{findremove}(\exp_1, \pat, \{p_1, \ldots, p_n\})
else
  return \exp
end if
```

And now we can present the algorithms for \( \text{norm} \) and \( \ell_2s \). The latter function uses the transformations P1-P5 from Section 2.4.

**Algorithm 2** The function \( \text{norm}(\exp) \)

```plaintext
\exp \Leftarrow \text{applysub}(\exp, \text{norm}) \{\text{Recursive traversal into } \exp\}

if match(\exp, \exp_1 \o \exp_2, \{o, \exp_1, \exp_2\}) and (o = \OR \text{ or } o = \AND) then
  while match(\exp, (\exists x.\exp_1) \o \exp_2, \{o, x, \exp_1, \exp_2\}) \text{ or } match(\exp, \exp_2 \o (\exists x.\exp_1), \{o, x, \exp_1, \exp_2\}) do
    \exp \Leftarrow \exists x.(\exp_1 \o \exp_2)
  end while
end if
return \exp
```
Algorithm 3 The function $l2s(\text{exp})$

Require: $\text{exp}$ - a safe $\lambda$SQL expression

\begin{verbatim}
if match($\text{exp}, \lambda x^e. \text{exp1}\{x, \text{exp1}\}$) then \{P1\}
  \text{exp1} \leftarrow \text{findremove}(\text{exp1}, x.\text{layer} = \text{table}, \{\text{layer}, \text{table}\})
  \text{exp} \leftarrow \text{SELECT} x.* \text{FROM table WHERE} \text{exp1}
else if match($\text{exp}, \lambda x^e. \exists y. \text{exp1}\{x, y, \text{exp1}\}$) then \{P2\}
  \text{exp1} \leftarrow \text{findremove}(\text{exp1}, (x = \text{exp2}), \{\text{exp2}\})
  \text{exp} \leftarrow \text{SELECT} \text{exp2} \text{FROM table WHERE} \text{exp1}
else if match($\text{exp}, \exists x. \text{exp1}\{\text{exp1}\}$) then \{P5\}
  \text{exp1} \leftarrow \text{findremove}(\text{exp1}, x.\text{layer} = \text{table}, \{\text{table}\})
  \text{exp} \leftarrow \text{EXISTS} (\text{SELECT} x.* \text{FROM table WHERE} \text{exp1})
end if

while match($\text{exp}, \text{SELECT} \text{selection} \text{FROM tablelist WHERE} \exists x. \text{exp1}\{\text{selection}, \text{tablelist}, x, \text{exp1}\}$) do \{P3\}
  \text{exp1} \leftarrow \text{findremove}(\text{exp1}, x.\text{layer} = \text{table}, \{\text{table}\})
  \text{exp} \leftarrow \text{SELECT} \text{selection} \text{FROM tablelist,table WHERE} \text{exp1}
end while

\text{exp} \leftarrow \text{applysub}(\text{exp}, l2s) \{\text{Recursive traversal into} \ \text{exp}\}

if match($\text{exp}, \text{aggr} (\text{SELECT} \text{exp1} \text{FROM tablelist WHERE} \text{exp2})\{\text{aggr}, \text{exp1}, \text{tablelist, exp2}\}$) then \{P4\}
  \text{exp} \leftarrow \text{SELECT} \text{aggr}(\text{exp1}) \text{FROM tablelist WHERE} \text{exp2}
end if

return $\text{exp}$
\end{verbatim}

Appendix B

Correctness of translation from safe-$\lambda$SQL to SQL

A $\lambda$SQL expression is translated into a legal SQL query if:

C1 The topmost operator in the string is a SELECT command. In an SQL query, the entire expression must be inside the scope of a SELECT.

C2 No $\lambda$ or $\exists$ operator are allowed in SQL.

C3 An aggregate function (a function that receives a set of entries and returns a single value) is placed inside the selectlist and should not receive a SELECT query as a parameter. This is just a peculiarity of the SQL syntax, which is handled by P4.

A safe-$\lambda$SQL expression must be an expression of type (et) and therefore, according to Safety2, the $\lambda$SQL expression is of the form $\lambda x^e.exp$. Safety3 requires $x,(x.layer = str) \triangleright exp$ to hold, and hence P1 successfully translates the topmost $\lambda$ operator into a SELECT command, which satisfies C1.

In order to show that C2 is satisfied, we need to show that any sub-expression of the form $\lambda x^a.exp$ or $\exists x^e.exp$ will be successfully translated into a legal SQL expression by one of the translation patterns:

- Assuming a sub-expression $q$ is of the form $\exists x^e.exp$, Safety3 requires $x,(x.layer = str) \triangleright exp$ to hold for some string str. Therefore, any existential quantifier is handled by either P3 or P5.

- Assuming a sub-expression $q$ is of the form $\lambda x^a.exp$, due to the $\lambda$ operator, the type of $q$ must be a complex type ($\alpha \beta$). According to Safety1, $\alpha \in \{e,r,s,t,g\}$ and $\beta = t$. 

48
• In case a sub-expression $q$ is of the form $\lambda x^e.exp$, Safety1 makes P1 applicable to $q$.

• Otherwise, in case $q$ is of the form $\lambda x^\alpha.exp$ and $\alpha \neq e$, Safety4 guarantees that P3 is applicable.
Appendix C

Safe lexicon rules

Before we present the constraint that are required for a safe lexicon, we need to discuss the properties of $\lambda$ operators in $\lambda$SQL. It would be handy to know in advance which $\lambda$ operators in the lexicon will be eliminated (due to $\beta$ reductions) in the final $\lambda$SQL expression. We can purpose a set of lexical safety rules that are less restrictive than the safe-$\lambda$SQL rules by enforcing those rules only for $\lambda$ operators that are not guaranteed to be eliminated. For example, Safety3 requires that if a sub-expression $p$ is of the form $p = \lambda x.\text{exp}$, then $x, (x.layer = \text{str}) \triangleright \text{exp}$ holds. However, in case we know $\lambda x$ is guaranteed to be eliminated, we do not need to enforce that $x, (x.layer = \text{str}) \triangleright \text{exp}$.

In the most general case, it is not possible to tell which $\lambda$ operators in the lexicon are expected to be eliminated in the final expression after parsing and which variables remain. However, under certain constraints it is possible to identify in advance some $\lambda$s that are guaranteed to be eliminated. One such constraint is the constraint that we call lexical safety rule Lex1. This rule restrict the type of any sub-expression of a $\lambda$SQL expression that is associated with a lexical entry. Lex1 requires that any such sub-expression that does not have a $\lambda$ operator at its outermost scope is of a basic types.

One consequence of Lex1 is a correspondence between $\lambda$ operators in a $\lambda$SQL expression and its syntactic category. In Section 2.1 we introduced a mapping $\text{Typ}$ between categories and their types. According to Lex1, the only way to introduce complex types into safe $\lambda$SQL expressions is by using $\lambda$ operators, and therefore the following relation between categories and $\lambda$SQL expressions holds:

Definition 7. Given a category $C$ and a $\lambda$SQL expression $\text{exp}$ of type $\gamma$, we say that $\text{exp}$ $\lambda$-correspond to $C$ if and only if either:

- $C$ is an atomic category and $\text{Typ}(C) = \gamma$
• C is complex category of the form A/B (or A\B) and exp = λx^α.exp1 where α = Typ(B) and exp1 λ-corresponds to A

Lemma 1. In a lexicon that is subject to Lex1, every λSQL expression λ-corresponds to its lexical category.

As a result of Lemma 1, every expression in the lexicon is of the form λx_1...λx_n.exp where the leading λ operators correspond to slashes in the syntactic category. We refer to the operators λx_1...λx_n as slash-corresponding λ operators. Based on the definition of λ operators that correspond to the syntactic category, we can define two types of λ operators: compositional λ that are expected to be eliminated in the final expression, and non-compositional λ that remains by the end of the parsing process.

Definition 8. given a category C and its corresponding λSQL expression exp, we define Comp(C : exp), the set of variables bound by a compositional λ operator\(^1\), as the minimal set of variables such that:

• If x is bound by a slash-corresponding λ operator in C : exp, x ∈ Comp(C : exp)

• If f(a_1,..a_n) is a sub-expression in exp, f ∈ Comp(C : exp) and a_i = λv_1...λv_m.exp for some 1 ≤ i ≤ n, 1 ≤ j ≤ m, then \{v_1,..v_m\} ⊂ Comp(C : exp).

The following is a suggested set of rules that are enforced for any category C and corresponding ASQL expression exp in the lexicon. We believe this set of rules is sufficient in order to guarantee the safety of our lexicon, due to unfinished work in progress.

Lex1 The type of any sub-expression that does not have a λ operator at its top scope must be a basic type.

Lex2 For any sub-expression of exp of the form λx^α.exp1, where x ∉ Comp(C : exp), α is a basic type.

Lex3 For any sub-expression of exp of the form λx^γ.exp1, where x ∉ Comp(C : exp) and γ ≠ e, exp1 is of the form ∃y^e.( (x = exp2) AND exp3)

Lex4 If C is not reducible to category N_{t=n} then all of the following conditions are true for any x ∈ V:

\(^1\)As a matter of convenience, we define a set of variables rather than a set of λ operators. There is a one-to-one correspondence between λ operators and variables due to renaming.
1. $x, (x.layer = str) \triangleright exp1$ is not true for any $str \in STR$.

2. For any $v \in V$ that is bound by $\lambda$ that slash corresponds to category that returns $N_{t=n}$, $x.v(e_1, \ldots, e_m, x) \triangleright exp1$ for any $e_1, \ldots, e_m$.

**Lex5** If $C$ can be reduced to the primitive category $N_{t=n}$ then $exp$ is of the form $\lambda v_1 \ldots \lambda v_n. \lambda x^e.exp1$ where $\lambda v_1 \ldots \lambda v_n$ are slash-corresponding $\lambda$ operators and the following conditions are true:

1. If $x, (x.layer = str) \triangleright exp1$ is true for some $str \in STR$ then $x, (x.layer = str1) \triangleright exp1$ implies $(str1 = str)$ and $x.v_i(e_1, \ldots, e_m, x) \triangleright exp1$ does not hold for any $e_1, \ldots, e_m$ and for any $v_i$ bound by a $\lambda$ operator that slash-corresponds to a category that returns $N_{t=n}$

2. If $x, (x.layer = str) \triangleright exp1$ is false for any $str \in STR$ then $x.v_i(e_1, \ldots, e_m, x) \triangleright exp1$ is true for some $e_1, \ldots, e_m$ and exactly one $v_i$ bound by a $\lambda$ operator that slash-corresponds to a category that returns $N_{t=n}$

**Lex6** For any sub-expression of $exp$ of the form $\exists x^e.exp1$ the following conditions are true:

1. If $x, (x.layer = str) \triangleright exp1$ is true for some $str \in STR$ then $x, (x.layer = str1) \triangleright exp1$ implies $(str1 = str)$ and $x.v_i(e_1, \ldots, e_m, x) \triangleright exp1$ does not hold for any $e_1, \ldots, e_m$ and for any $v_i$ bound by a $\lambda$ operator that slash-corresponds to a category that returns $N_{t=n}$

2. If $x, (x.layer = str) \triangleright exp1$ is false for any $str \in STR$ then $x.v_i(e_1, \ldots, e_m, x) \triangleright exp1$ is true for some $e_1, \ldots, e_m$ and exactly one $v_i$ bound by a $\lambda$ operator that slash-corresponds to a category that returns $N_{t=n}$
Appendix D

Tested queries

The following is a list of queries that were tested with the NLI.

Notes:

- *rova* is used to specify city quarters in the test data model.
- *Arlozerov, Hertzel, Haneviim, Hahalutz and Moria* are street names

Buildings with more than one business between buildings with more than one business that are up to 200 m from the intersection of Hertzel and Haneviim and buildings with more than one business that are up to 200 m from the intersection of Hertzel and Arlozerov

Buildings with more than one business that are up to 200 m from the intersection of Hertzel and Haneviim

Buildings that are up to 400 m from the intersection of Hertzel and Haneviim

Buildings between Hertzel and Hahalutz

Buildings that are up to 100 m from the border of rova 4

Buildings with businesses in a rova that borders a rova that contains Moria street

Streets that are up to 50m from buildings with more than 20 floors

Buildings with more than 20 floors

Rova that contains a building with more than 50 floors and borders a rova that
contains Moria street

Streets in a rova that contains a building with more than 50 floors and borders a rova that contains Moria street

A rova that contains a water body

Closest street to a water body

Streets that are up to 300m from the closest water body to Moria street

Streets between Moria street and closest water body with an area of at least 500m² to Moria

Buildings that are up to 200m from building with most floors

Water body that is up to 2km from Moria street with largest area

Biggest building up to 500m from the intersection of hertzel and Haneviim

Buildings with more than one business that are north of Moria street and west of Hertzel

Streets that are north of between 30 and 60 water bodies

Largest water body north of the center of Moria street
## Appendix E

### Lexicon

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>λSQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>(G_{p=n}/N_{p=0,t=n})</td>
<td>(\lambda n_1^t.(\text{GeomUnion}(\lambda g_1^g.\exists x^g.(g_1 = x.\text{the_geom} \text{ AND } n_1(x)))))</td>
</tr>
<tr>
<td>T2</td>
<td>(G_{p=n}/Gs)</td>
<td>(\lambda g_1^g.(\text{GeomUnion}(gs_1)))</td>
</tr>
<tr>
<td>T3</td>
<td>(N_{t=u}/(N_{t=u}\backslash Rs_{p=n,t=num}))</td>
<td>(\lambda n_1^t.\lambda n_2^t.\exists x^g.(n_1(x) \text{ AND } n_2(x)))</td>
</tr>
<tr>
<td>T4</td>
<td>((S/(S\backslash NP_{t=nondet}))/N_{p=0,t=n})</td>
<td>(\lambda n_1^t.\lambda n_2^t.\exists x^e.(q_1(\lambda y^f.\text{rel}(y,x))))</td>
</tr>
<tr>
<td>T5</td>
<td>(((S\backslash NP)/(S/(S\backslash NP)))/(S\backslash NP)/NP)</td>
<td>(\lambda p^t.\lambda n_1^t.\lambda n_2^t.(p(\text{count}(\lambda x^e.(n_1(x) \text{ AND } n_2(x))))))</td>
</tr>
<tr>
<td>T6</td>
<td>(((S/(S\backslash NP_{t=q}))/N_{p=0,t=n})/Rs_{p=n,t=num})</td>
<td>(\lambda p^t.\lambda n_1^t.\lambda n_2^t.(\lambda x^e.\text{dist}(x)))</td>
</tr>
<tr>
<td>T7</td>
<td>(((N_{p=0,n,t=n})/N_{p=0,t=n})/(S/(S\backslash NP_{t=q}))/((N_{p=0,n,t=n})/G_{p=n}))</td>
<td>(\lambda p^t.\lambda n_1^t.\lambda x^e.\text{dist}(n_1(x) \text{ AND } q_1(\lambda y^f.(\lambda x^e.\text{area}(\lambda z^e.\text{TRUE},x)))))</td>
</tr>
<tr>
<td>#</td>
<td>R_{p=n,t=num}</td>
<td>#</td>
</tr>
<tr>
<td>#area</td>
<td>(N_{t=#layer,t=u}\backslash Rs_{p=n,t=area})</td>
<td>(\lambda p^t.\lambda x^e.(p(x.#\text{area})))</td>
</tr>
<tr>
<td>#area</td>
<td>(N_{t=#layer,t=u}/Rs_{p=of,t=area})</td>
<td>(\lambda p^t.\lambda x^e.(p(x.#\text{area})))</td>
</tr>
<tr>
<td>#layer</td>
<td>N_{t=#layer,p=0,t=n}</td>
<td>(\lambda x^e.(x.\text{layer} = #\text{layer}))</td>
</tr>
<tr>
<td>#length</td>
<td>N_{t=#layer,t=u}\backslash Rs_{p=of,t=dist}</td>
<td>(\lambda p^t.\lambda x^e.(p(x.#\text{length})))</td>
</tr>
<tr>
<td>#length</td>
<td>(N_{t=#layer,t=u}/Rs_{p=of,t=dist})</td>
<td>(\lambda p^t.\lambda x^e.(p(x.#\text{length})))</td>
</tr>
</tbody>
</table>
\begin{tabular}{|c|c|c|}
\hline
Word & Category & $\lambda$SQL expression \\
\hline
#max & $N = \#layer, p = no, t = n / N = \#layer, p = no, t = n$ & $\lambda n^{1+}.\lambda x^e.(n1(x) AND x.#max = max(\lambda n^{1+}.\lambda y^e.(n1(y) AND n = y, \#max)))$ \\
#min & $N = \#layer, p = no, t = n / N = \#layer, p = no, t = n$ & $\lambda n^{1+}.\lambda x^e.(n1(x) AND x.#min = min(\lambda n^{1+}.\lambda y^e.(n1(y) AND n = y, \#min)))$ \\
#real & $N = \#layer, t = u / R_{p = no, t = num}$ & $\lambda p^{1+}.\lambda x^e.(p(x, \#real))$ \\
#realId & $N = \#layer, p = no, t = n / R_{p = n, t = num}$ & $\lambda n^{1+}.\lambda x^e.(x.layer = \#layer AND x.#realId = n)$ \\
#strval & $N_{p = no, t = n} / N_{\#layer, p = no, t = n}$ & $\lambda n^{1+}.\lambda x^e.(n(x) AND x.#attrLIKE#strval)$ \\
#uniqstr & $N_{p = no, t = n}$ & $\lambda x^e.(x.layer = \#layer AND x.#attrLIKE#uniqstr)$ \\
a & $N = \#x.p = no, t = y / N = \#x.p = no, t = y$ & $\lambda x^e, (x > n)$ \\
above & $R_{s = \#u / R_{p = \#u}}$ & $\lambda x^e, (x * 4047)$ \\
acres & $R_{p = n, t = area / R_{p = n, t = num}}$ & $\lambda g^{1+}.\lambda n^{2+}.\lambda x^e.(n2(x) AND x.the_geom & & g1 AND within(x.the_geom, convexhull(g1)))$ \\
among & $((N_{p = no, t = n} \land N_{p = no, t = n}) / G_{p = n})$ & $\lambda n^{1+}.(n)$ \\
and & $N = \#x.p = no, t = y / N = \#x.p = no, t = y$ & $\lambda n^{1+}.\lambda n^{2+}.\lambda x^e.(n1(x) AND n2(x))$ \\
and & $((S \setminus NP) \setminus (S \setminus NP)) / (S \setminus NP)$ & $\lambda n^{1+}.\lambda n^{2+}.\lambda x^e.(n1(x) AND n2(x))$ \\
and & $((N_{p = no, t = n} \land N_{p = no, t = n}) \land (N_{p = no, t = n} \land N_{p = no, t = n}))$ & $\lambda d^{(e^0)}, m_{(e^0)}^{(e^0)} \lambda n^{1+}.\lambda x^e.(a(n, x) AND b(n, x))$ \\
and & $R_{p = and / R_{p = n, t = num}}$ & $\lambda x^e.(x)$ \\
and & $G_{p = and / G_{p = n}}$ & $\lambda g^{1+} .(g1)$ \\
and & $N_{p = and, t = n} / N_{p = no, t = n}$ & $\lambda n^{1+}.(n1)$ \\
are & $((S \setminus N\_P = \#x) / (N_{p = no, t = n} / N_{p = no, t = n}))$ & $\lambda mod^{(e^0)}(e^0). (mod(\lambda x^e.(TRUE)))$ \\
are & $((S \setminus N\_P = \#x) / (N_{p = no, t = n} / N_{p = no, t = n}))$ & $\lambda mod^{(e^0)}(e^0). (mod(\lambda x^e.(TRUE)))$ \\
at least & $R_{s = \#u / R_{s = \#u}}$ & $\lambda n^{1+}.\lambda x^e.(x >= n)$ \\
at most & $R_{s = \#u / R_{s = \#u}}$ & $\lambda n^{1+}.\lambda x^e.(x <= n)$ \\
below & $R_{s = \#u / R_{s = \#u}}$ & $\lambda n^{1+}.\lambda x^e.(x < n)$ \\
between & $((R_{s = \#u / R_{p = no, t = \#u}} / R_{t = \#u})$ & $\lambda n^{1+}.\lambda n^{2+}.\lambda x^e.(x >= n AND x <= n2)$ \\
between & $((R_{s = \#u / R_{p = no, t = \#u}} / R_{t = \#u})$ & $\lambda n^{1+}.\lambda n^{2+}.\lambda x^e.(x >= n AND x <= n2)$ \\
between & $((N_{p = no, t = n} \land N_{p = no, t = n}) / G_{p = and} / G_{p = n})$ & $\lambda g^{1+}.\lambda g^{2+}.\lambda n^{2+}.\lambda x^e.(n2(x) AND x.the_geom & & GeomUnion(g1, g2) AND within(x.the_geom, di fference(convexhull (GeomUnion(g1, g2)), GeomUnion(convexhull (g1), convexhull(g2))))$ \\
\hline
\end{tabular}
<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>(\lambda SQL) expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>border</td>
<td>((Gs/N_{p=of,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda g^{1^{st}}.\exists x^{e}.(n1(x)\ AND\ g1 = boundary(x.the.geom)))</td>
</tr>
<tr>
<td>border with</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{touch}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>borders</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{touch}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>boundary</td>
<td>((Gs/N_{p=of,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda g^{1^{st}}.\exists x^{e}.(n1(x)\ AND\ g1 = boundary(x.the.geom)))</td>
</tr>
<tr>
<td>center</td>
<td>(((S\setminus N_{p=of,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda g^{1^{st}}.\lambda x^{e}.(n1(x)\ AND\ g1 = centroid(x.the.geom)))</td>
</tr>
<tr>
<td>closest</td>
<td>(((N_{p=0,t=n}/G_{p=to})/N_{p=to,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda g^{1^{st}}.\lambda x^{e}.(n1(x)\ AND\ distance(x.the.geom,g1) = min(\lambda n^{e}.\exists y^{e}.(n1(y)\ AND\ n = distance(x.the.geom,g1)))))</td>
</tr>
<tr>
<td>cm</td>
<td>((R_{p=n,t=dist}/R_{p=n,t=num}))</td>
<td>(\lambda x^{e}.(x/1000))</td>
</tr>
<tr>
<td>contain</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{contains}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>contains</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{contains}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>cross</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{crosses}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>crosses</td>
<td>(((S\setminus NP)/NP))</td>
<td>(\lambda y^{e}.\lambda x^{e}.(\text{crosses}(x.the.geom,y.the.geom)))</td>
</tr>
<tr>
<td>do not</td>
<td>(((S\setminus NP)/(S\setminus NP)))</td>
<td>(\lambda n^{1^{st}}.\lambda x^{e}.(\text{NOT}(n1(x))))</td>
</tr>
<tr>
<td>does not</td>
<td>(((S\setminus NP)/(S\setminus NP)))</td>
<td>(\lambda n^{1^{st}}.\lambda x^{e}.(\text{NOT}(n1(x))))</td>
</tr>
<tr>
<td>dunam</td>
<td>((R_{p=n,t=area}/R_{p=n,t=num}))</td>
<td>(\lambda x^{e}.(x * 1000))</td>
</tr>
<tr>
<td>east</td>
<td>(((N_{p=0,t=n}/N_{p=0,t=n})/G_{p=of}))</td>
<td>(\lambda n^{1^{st}}.\lambda n^{2^{nd}}.\lambda x^{e}.(n2(x)\ AND\ x.the.geom &gt;&gt; g1))</td>
</tr>
<tr>
<td>every</td>
<td>(((S\setminus NP)/N_{p=0,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda n^{2^{nd}}.\lambda x^{e}.(\text{count}(\lambda x^{e}.(n1(x)\ AND\ \text{NOT}(n2(x)))) = 0))</td>
</tr>
<tr>
<td>five</td>
<td>(R_{p=n,t=num})</td>
<td>5</td>
</tr>
<tr>
<td>four</td>
<td>(R_{p=n,t=num})</td>
<td>4</td>
</tr>
<tr>
<td>from</td>
<td>(((N_{p=0,t=n}\setminus N_{p=0,t=n})/G_{p=to})\setminus G_{p=to})</td>
<td>(\lambda p^{x^{e}}.\lambda g^{1^{st}}.\lambda n^{2^{nd}}.\lambda x^{e}.(n2(x)\ AND\ x.the.geom&amp;&amp;\text{expand}(g1,(\text{UBOUNDp}))\ AND\ \text{p}(\text{distance}(\text{x.the.geom},g1))))</td>
</tr>
<tr>
<td>from</td>
<td>((G_{p=from}/G_{p=n}))</td>
<td>(\lambda g^{1^{st}}.\lambda g^{1^{st}}.\lambda n^{2^{nd}}.\lambda x^{e}.(n2(x)\ AND\ x.the.geom&amp;&amp;\text{expand}(g1,(\text{UBOUNDp}))\ AND\ \text{p}(\text{distance}(\text{x.the.geom},g1))))</td>
</tr>
<tr>
<td>furthest</td>
<td>(((N_{p=0,t=n}/G_{p=from})/N_{p=0,t=n}))</td>
<td>(\lambda n^{1^{st}}.\lambda n^{1^{st}}.\lambda g^{e}.(n1(x)\ AND\ distance(x.the.geom,g1) = max(\lambda n^{e}.\exists y^{e}.(n1(y)\ AND\ n = distance(x.the.geom,g1)))))</td>
</tr>
<tr>
<td>got</td>
<td>(((S\setminus NP)\setminus x)/N_{p=0,t=u}))</td>
<td>(\lambda n^{1^{st}}(n1))</td>
</tr>
<tr>
<td>has</td>
<td>(((S\setminus NP)\setminus x)/N_{p=0,t=u}))</td>
<td>(\lambda n^{1^{st}}(n1))</td>
</tr>
<tr>
<td>Word</td>
<td>Category</td>
<td>ASQL expression</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>have</td>
<td>((S(NP)_{#x})/N=#x,t=u)</td>
<td>$\lambda n^1.(n) = \lambda n^1.(n) = \lambda n^2.(n^2).n$. (n = \text{count}(\lambda x^e \cdot (n(x) \text{ AND } n2(x))))</td>
</tr>
<tr>
<td>how many</td>
<td>((S(NP=#x))/NP_{#x,t=u})</td>
<td>$\lambda g^1.(\lambda_{#x}) \cdot x^e \cdot (n2(x) \text{ AND } x \text{.the_geom} &amp; &amp; g1 \text{ AND } \text{within}(x \text{.the_geom}, g1))$</td>
</tr>
<tr>
<td>in</td>
<td>((N_{#x,t=u} \text{.} \text{&amp;} P=#x, t=u))</td>
<td>$\lambda y^e \cdot \lambda x^e \cdot (\text{intersects}(x \text{.the_geom}, y \text{.the_geom}))$</td>
</tr>
<tr>
<td>intersect</td>
<td>(((N_P=#x,t=u)/N=#x,t=u))</td>
<td>$\lambda n^1.(\lambda n^2.(\lambda g^1.(\lambda x^e \cdot (n1(x) \text{ AND } n2(y) \text{ AND } \text{intersects}(x \text{.the_geom}, y \text{.the_geom})) \text{ AND } g1 = \text{intersection}(x \text{.the_geom}, y \text{.the_geom}))))$</td>
</tr>
<tr>
<td>intersection</td>
<td>((G_s/N=#x,t=u)/N=#x,t=u)</td>
<td>$\lambda n^1.(\lambda n^2.(\lambda g^1.(\lambda x^e \cdot (n1(x) \text{ AND } n2(y) \text{ AND } \text{intersects}(x \text{.the_geom}, y \text{.the_geom}) \text{ AND } g1 = \text{intersection}(x \text{.the_geom}, y \text{.the_geom}))))$</td>
</tr>
<tr>
<td>km</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.(x = 1000)$</td>
</tr>
<tr>
<td>km</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.(x = 1000)$</td>
</tr>
<tr>
<td>largest</td>
<td>((N_{#x,t=u} \text{.} \text{&amp;} N1=#x,p=#x,t=u))</td>
<td>$\lambda r2_{e}(r^0)(e^0) \cdot \lambda a^e(e^0) \cdot x^e \cdot (a(\lambda y^e.(\text{TRUE})), x) \text{ AND } r2_{e}(\lambda n^e.(n) = \text{max}(\lambda n^2. \cdot \exists y^e.(a(\lambda x^e.(\text{TRUE})), y) \text{ AND } r2_{e}(\lambda n^3.(n2 = n3), y)))) \cdot x)$</td>
</tr>
<tr>
<td>least</td>
<td>((N_{#x,t=u} \text{.} \text{&amp;} N1=#x,t,u))</td>
<td>$\lambda r2_{e}(r^0)(e^0) \cdot \lambda a^e(e^0) \cdot x^e \cdot (a(\lambda y^e.(\text{TRUE})), x) \text{ AND } r2_{e}(\lambda n^e.(n) = \text{min}(\lambda n^2. \cdot \exists y^e.(a(\lambda x^e.(\text{TRUE})), y) \text{ AND } r2_{e}(\lambda n^3.(n2 = n3), y)))) \cdot x)$</td>
</tr>
<tr>
<td>less than</td>
<td>(R_t_{#u}/R_t=#u)</td>
<td>$\lambda x^e.(x &lt; n)$</td>
</tr>
<tr>
<td>m</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.(x)$</td>
</tr>
<tr>
<td>m2</td>
<td>(R_p,n,t=area \text{&amp;} R_p,n,t=num)</td>
<td>$\lambda x^e.(x)$</td>
</tr>
<tr>
<td>meter</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.(x)$</td>
</tr>
<tr>
<td>meters</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.(x)$</td>
</tr>
<tr>
<td>mile</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.\cdot (x = 1609)$</td>
</tr>
<tr>
<td>miles</td>
<td>(R_p,n,t=dist &amp; R_p,n,t=num)</td>
<td>$\lambda x^e.\cdot (x = 1609)$</td>
</tr>
<tr>
<td>more than</td>
<td>(R_t_{#u}/R_t=#u)</td>
<td>$\lambda n^e.\cdot x^e.(x &gt; n)$</td>
</tr>
<tr>
<td>most</td>
<td>(N_{#x,t=u} \text{.} \text{&amp;} N1=#x,t=u)</td>
<td>$\lambda r2_{e}(r^0)(e^0) \cdot \lambda a^e(e^0) \cdot x^e \cdot (a(\lambda y^e.(\text{TRUE})), x) \text{ AND } r2_{e}(\lambda n^e.(n) = \text{max}(\lambda n^2. \cdot \exists y^e.(a(\lambda x^e.(\text{TRUE})), y) \text{ AND } r2_{e}(\lambda n^3.(n2 = n3), y)))) \cdot x)$</td>
</tr>
<tr>
<td>Word</td>
<td>Category</td>
<td>(\text{\lambda SQL expression} )</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>no</td>
<td>((S(S(NP_{eq}))/N_p=n0,t=n)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\text{(count}(\lambda x^{eq}.(n1(x) \text{ AND } n2(x)))) = 0)</td>
</tr>
<tr>
<td>north</td>
<td>((N_p=n0,t=n/N_p=n0,t=n)/G_p=off)</td>
<td>(\lambda g1^{rd}.\lambda n2^{nd}.\lambda x^{eq}.(n2(x) \text{ AND x.the.geom} &gt;&gt; g1)</td>
</tr>
<tr>
<td>not</td>
<td>(N_{t=u}/N_t=u)</td>
<td>(\lambda n1^{st}.\lambda x^{eq}.(\text{NOT}(n1(x)))</td>
</tr>
<tr>
<td>not</td>
<td>((N_p=n0,t=n/N_p=no,t=n)/(N_p=no,t=n/N_p=no,t=n))</td>
<td>(\lambda \text{mod}(\text{true})\text{eq}.\lambda n1^{st}.\lambda x^{eq}.(n1(x) \text{ AND NOT(mod(\lambda y^{eq}.(\text{true}),x)))}</td>
</tr>
<tr>
<td>of</td>
<td>(R_p=off,t=#x/R_p=n,t=#x)</td>
<td>(\lambda p^{eq}.(p)</td>
</tr>
<tr>
<td>of</td>
<td>(G_p=off/G_p=n)</td>
<td>(\lambda g1^{rd}.(g1)</td>
</tr>
<tr>
<td>of</td>
<td>(N_p=off,t=n/N_p=no,t=n)</td>
<td>(\lambda n1^{st}.(n1)</td>
</tr>
<tr>
<td>one</td>
<td>R_p=n,t=num</td>
<td>1</td>
</tr>
<tr>
<td>or</td>
<td>((N_{t=u}/N_t=u)/N_t=u)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(n1(x) \text{ OR } n2(x))</td>
</tr>
<tr>
<td>or</td>
<td>((S{NP}){S{NP}}){S{NP}})</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(n1(x) \text{ OR } n2(x))</td>
</tr>
<tr>
<td>or</td>
<td>(N_p=n0,t=n,N_p=n0,t=n/(N_p=n0,t=n,N_p=n0,t=n))</td>
<td>(\lambda a^{eq}(a),\lambda b^{eq}(a),\lambda n1^{st}.\lambda x^{eq}.(a(n,x) \text{ OR b(n,x))}</td>
</tr>
<tr>
<td>or</td>
<td>(N_p=n0,t=n,N_p=n0,t=n)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(n1(x) \text{ OR n2(x))}</td>
</tr>
<tr>
<td>overlap</td>
<td>(S{NP}/NP)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(x &gt; n)</td>
</tr>
<tr>
<td>overlaps</td>
<td>(S{NP}/NP)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(x &gt; n)</td>
</tr>
<tr>
<td>result</td>
<td>N_p=n0,t=n</td>
<td>(\lambda x^{eq}.(x, \text{layer} = \text{lastquery})</td>
</tr>
<tr>
<td>smallest</td>
<td>(N_p=n0,t=n/{N_{t=x,p=n0,t=n}/N_{t=x,t=u}})/</td>
<td>(\lambda \text{rt2eq}(\text{rt}(a),\lambda a^{eq}(a),\lambda x^{eq}.(a(\lambda y^{eq}.(\text{true}),x) \text{ AND rt2eq}(\lambda n^{eq}.(n \text{ = } =)</td>
</tr>
<tr>
<td>smallest</td>
<td>(N_p=n0,t=n/{N_{t=x,p=n0,t=n}/N_{t=x,t=u}})/</td>
<td>\text{min}(\lambda n2^{nd}.\lambda y^{eq}.(a(\lambda n^{eq}.(\text{true}),y) \text{ AND rt2eq}(\lambda n3^{rd}.(n2 = n3),y)),x)),</td>
</tr>
<tr>
<td>smallest</td>
<td>(N_p=n0,t=n/{N_{t=x,p=n0,t=n}/N_{t=x,t=u}})/</td>
<td>\text{AND rt2eq}(\lambda n^{eq}.(n \text{ = } =)</td>
</tr>
<tr>
<td>some</td>
<td>(S{NP}_{eq})/N_p=n0,t=n</td>
<td>\text{min}(\lambda n2^{nd}.\lambda y^{eq}.(a(\lambda n^{eq}.(\text{true}),y) \text{ AND rt2eq}(\lambda n3^{rd}.(n2 = n3),y)),x)),</td>
</tr>
<tr>
<td>south</td>
<td>(N_p=n0,t=n/N_p=no,t=n)/G_p=off)</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(n1(x) \text{ AND n2(x))}</td>
</tr>
<tr>
<td>sq. meters</td>
<td>(R_p=n,t=area/R_p=n,t=num)</td>
<td>(\lambda g1^{rd}.\lambda n2^{nd}.\lambda x^{eq}.(n2(x) \text{ AND x.the.geom} &lt; &lt;</td>
</tr>
<tr>
<td>sq. miles</td>
<td>(R_p=n,t=area/R_p=n,t=num)</td>
<td>(\lambda x^{eq}.(x)</td>
</tr>
<tr>
<td>ten</td>
<td>R_p=n,t=num</td>
<td>(\lambda x^{eq}.(x * 2.69e + 06)</td>
</tr>
<tr>
<td>that</td>
<td>(N_{t=x,p=n0,t=n}/N_{t=x,p=n0,t=n})/S{NP_{eq} };</td>
<td>(\lambda n1^{st}.\lambda n2^{nd}.\lambda x^{eq}.(n1(x) \text{ AND n2(x))}</td>
</tr>
<tr>
<td>the</td>
<td>(N_{t=x,p=no,t=y}/N_{t=x,p=no,t=y})</td>
<td>(\lambda n1^{st}.(n1)</td>
</tr>
</tbody>
</table>

\(n^{eq} \) denotes equality, \(n^{rd} \) denotes rectangular, \(n^{nd} \) denotes non-convex, \(n^{st} \) denotes straight, \(n^{true} \) denotes true, \(n^{false} \) denotes false, \(n^{area} \) denotes area, and \(n^{dist} \) denotes distance.
<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>$\lambda$SQL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>$(G_{p=n}/G_{p=n})$</td>
<td>$\lambda g_1^g.(g_1)$</td>
</tr>
<tr>
<td>three</td>
<td>$R_{p=n,t=num}$</td>
<td>3</td>
</tr>
<tr>
<td>to</td>
<td>$(G_{p=0}/G_{p=n})$</td>
<td>$\lambda g_1^g.(g_1)$</td>
</tr>
<tr>
<td>twenty</td>
<td>$R_{p=n,t=num}$</td>
<td>20</td>
</tr>
<tr>
<td>two</td>
<td>$R_{p=n,t=num}$</td>
<td>2</td>
</tr>
<tr>
<td>up to</td>
<td>$(R_{t=#u}/R_{t=#u})$</td>
<td>$\lambda n^r.x^r.(x &lt;= n)$</td>
</tr>
<tr>
<td>west</td>
<td>$((N_{p=no,t=n}\backslash N_{p=0,t=n})/G_{p=of})$</td>
<td>$\lambda g_1^g.\lambda n2^e.\lambda x^e.(n2(x) \ AND \ x.the_geom &lt;&lt; g_1)$</td>
</tr>
<tr>
<td>which</td>
<td>$((S\backslash(S\nabla N_{i=eq})))/N_{p=no,t=n}$</td>
<td>$\lambda n1^e.\lambda n2^e.\lambda x^e.(n1(x) \ AND \ n2(x))$</td>
</tr>
<tr>
<td>with</td>
<td>$((N_{i=#x,p=no,t=n}\backslash N_{i=#x,p=0,t=n})/(S\nabla N_{i=#x}))$</td>
<td>$\lambda n1^e.\lambda n2^e.\lambda x^e.(n1(x) \ AND \ n2(x))$</td>
</tr>
<tr>
<td>within</td>
<td>$((N_{p=no,t=n}\backslash N_{p=0,t=n})/G_{p=})$</td>
<td>$\lambda g_1^g.\lambda n2^e.\lambda x^e.(n2(x) \ AND \ x.the_geom &amp;&amp; g1 \ AND \ within(x.the_geom, g1))$</td>
</tr>
<tr>
<td>within</td>
<td>$((N_{p=0,t=dist})/R_{p=n,t=dist})$</td>
<td>$\lambda n^r.\lambda g1^g.\lambda n2^e.\lambda x^e.(n2(x) \ AND \ x.the_geom &amp;&amp; expand(g1, n) \ AND \ within(x.the_geom, buffer(g1, n)))$</td>
</tr>
<tr>
<td>without</td>
<td>$((N_{i=#x,p=no,t=n}\backslash N_{i=#x,p=0,t=n})/N_{i=#x,t=0})$</td>
<td>$\lambda n1^e.\lambda n2^e.\lambda x^e.(N OT(n1(x)) \ AND \ n2(x))$</td>
</tr>
<tr>
<td>zero</td>
<td>$R_{p=n,t=num}$</td>
<td>0</td>
</tr>
</tbody>
</table>
Bibliography


ממשק בשפה טבעית למערכות מידע גאוגרפי

סלע מזר-דיהם
ממשק בשפה טבעית למערכות מידע גיאוגרפי

hibor Ul Mudder

לשמ מיידי halki Shel herishot Lekeleha horazar
mgesor Lemdeyim BeMidey haMishab

Salz Mezor-Yidim

hong LeShom haTecunik – Hekhor Tekenologi LeIsrael

2006 April Tel-Aviv Hidpe
המחק נערך בהנחייתו של פרופ' יوذ וינטר בפקולטה להנדסה המחשב.

אני מודה טכניון על התמיכה הכספית הנדרשת בהשלמה.

אני מודה לテクニオン על pneumonia הבסיסה הנדרשת בהשלמה.
הקדמה

1.1 מוטיבציה

1.2 תרומה

1.3 עבודה קדימה

1.3.1 מערכות מידע גאוגרפי

1.3.2 ממשקים בשפה טבעית למערכות מידע גאוגרפי

1.4 מבנה העבודה

2 גישה קומפקטית לבניית שאילתות SQL

2.1 דקדוק קטגורי

2.2 SQL

2.3 בטוח

2.4 תרגום מSQL- בטוח לSQL

2.5 לקסיקון בטוח

3 סוגיות סמנטיות

3.1 מרחבי עצמי לעומת סמנטיקה יישמית

3.2 הסמנטיקה של ובין

4 ארכיטקטורה של לקסיקון

4.1 פריטים לקסיקליים לא מרחביים

4.2 פריטים לקסיקליים מרחביים

4.3 פריטים לקסיקליים תלויי נתונים

4.4 מעברי טיפוס

5 מימוש

5.1קובץ הגדרת לקסיקון

5.2 ארכיטקטורת תוכנה לדקדוקカテゴיאלי

5.3 קישור לGIS

5.4 אופטימיזציה של שאילתות

5.5 דוגמה

6 מסקנות

נספחים

A אלגוריתם התרגום מSQL- בטוח לSQL

B נספחו תרגום מSQL- בטוח לSQL

C חוקי לקסיקון בטוח

D שאילתות שנבדקו

רשימה ביבליוגרפית
רשימת איורים

25 .......................................................................................................................... דוגמה לתוכן 3.1
38 .......................................................................................................................... תוצאת שאילתת QGIS 5.1
תקציר

מערכות מידע גאוגרפי (GIS) הן מערכות מידע שנועדו לעבד מידע הקשור לקואורדינטות של כדור הארץ. על אף הגדילה המהירה בקהיליית המשתמשים ב-GIS, המערכות הקיימות הן בדרך כלל מסובכות לשימוש ודורשות תהליך לימוד ארוך. בספרות בתחום ה-GIS קיימת הכרה בפוטנציאל של ממשקים בשפה טבעית ויכולתם של ממשקים כאלו לאפשר למשתמשים לנצל את היכולות המורכבות יותר של תכנות GIS. עתיד שבראש ובראשונה, מערכות מידע גאוגרפי מלהשלמת שילובים, التعاون של מספר עניבונים המרטיבים או לא הפחתה ממצואה של ה-GIS.

SQL


1. Introduction

In this paper, we propose a system that automatically generates regular expressions from SQL expressions. The system is designed to work with a specific database management system, namely PostgreSQL. The system takes as input a SQL query and outputs an equivalent regular expression that can be used for data filtering and extraction.

2. Related Work

Previous work in the area of regular expressions and SQL has focused on the development of tools for generating regular expressions from SQL queries. However, these tools are typically limited in their ability to handle complex SQL queries.

3. System Description

Our system consists of two main components: an expression parser that converts a SQL query into an equivalent regular expression, and a query optimizer that translates the regular expression into a SQL query.

4. Implementation

The expression parser is implemented using a combination of parsing and pattern matching techniques. The query optimizer uses a set of rules to transform the regular expression into a SQL query.

5. Evaluation

We evaluated our system using a set of benchmark queries taken from the TPC-H benchmark. The results show that our system is able to accurately generate regular expressions from SQL queries, with an average error rate of 2%.

6. Conclusion

In conclusion, we have presented a system for generating regular expressions from SQL queries. The system is able to handle complex SQL queries and has been shown to be effective in practice.

References


II


ליברמן ז"ל