A LEXICALIST APPROACH TO NATURAL-LANGUAGE DATABASE FRONT-ENDS

by

Nicolas P. Demers
B.Sc., University of Ottawa, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Nicolas P. Demers 1999
SIMON FRASER UNIVERSITY
April 1999

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Natural language interfaces (NLI's) to databases allow end-users to access information in databases by typing in commands and requests in a natural language. These commands and requests are then translated into some formal database language, most commonly SQL. NLI's suffer from the same linguistic and computational problems as other natural language processing systems, though the severity of some problems is reduced due to the restricted language used and the restricted domain of discourse. Another important issue for database interfaces concerns porting them to new domains or to other natural languages.

This thesis introduces a lexicalist approach to database NLI's based on unification grammars, along with a small-scale example. A sample bilingual lexicon is proposed, to demonstrate the feasibility of this approach. The solution proposed shows that a lexicalist approach is not only feasible but, for unambiguous words and expressions, can result in reasonable complexity and processing time. In the case of ambiguity the solution is less satisfactory: there are inevitable tradeoffs between correctness and low processing time on the one hand, low complexity and ease of lexicon creation on the other.

I will also propose an algorithm to semi-automatically generate a bilingual lexicon for unambiguous expressions. The solution is based on the concept of a bilingual text; in this case we will be looking at sets of English/SQL query pairs. These pairs are meant to establish syntactic/semantic relationships between English expressions and SQL expressions, providing the necessary building blocks for more complex queries.
“Me, my thoughts are flower-strewn
Ocean storm, bayberry moon.
I have got to leave to find my way.”

—R.E.M., “Find The River”
Acknowledgements

I would like to express my deepest gratitude to some of the many people who made my life here at SFU exciting and memorable.
To Jeff and Nathan, for being there, and supporting me through some difficult times.
To Tracey, for the hugs, the late-night talks, and the muffins.
To Laura, just because she’s awesome.
To Ann, for the prodding, the movie passes, and believing in my crazy dreams.
To Fred, the best supervisor I could hope to have.
And lastly, to all the wonderful people at SF P!RG and Out on Campus, who made this stranger in a strange land feel right at home.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Quotation</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Natural Language Interfaces to Databases</td>
<td>4</td>
</tr>
<tr>
<td>2.2 The Relational Database Model and SQL</td>
<td>6</td>
</tr>
<tr>
<td>2.3 The Grammars</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Shake-And-Bake</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Nominal Compounds and Complex Nominals</td>
<td>24</td>
</tr>
<tr>
<td><strong>3 Translation of Nominal Compounds</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Specific Problems</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Basic Solution</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Problem Areas</td>
<td>50</td>
</tr>
<tr>
<td>3.4 Evaluation and Discussion</td>
<td>67</td>
</tr>
<tr>
<td>3.5 Future Work</td>
<td>72</td>
</tr>
<tr>
<td>3.6 Conclusion</td>
<td>73</td>
</tr>
<tr>
<td><strong>4 Portability and Lexical Generation</strong></td>
<td>75</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>75</td>
</tr>
</tbody>
</table>
4.2 English Fragments ................................................................. 76
4.3 SQL Queries and Distribution of Information ....................... 79
4.4 Algorithm for Lexical Generation ........................................... 79
4.5 Generation Examples ........................................................... 82
4.6 After Generation ................................................................. 91
4.7 Evaluation and Discussion ..................................................... 91
4.8 Other Components ............................................................... 92
4.9 Alternative Approaches ......................................................... 92
4.10 Conclusion ............................................................... 93

5 Conclusion ............................................................... 95

A Selected English Grammar Rules and Macros ........................ 97

B The SQL Grammar and Signature ........................................... 101

C Transfer Macros ............................................................... 109

Bibliography ............................................................................ 115
Chapter 1

Introduction

Natural language interfaces to databases, allowing end-users to access information in databases by typing in commands and requests in a natural language such as English, have been developed since the late sixties. The main advantage of these systems is that they facilitate access by users who are familiar with neither formal query languages nor the internal structures of database management systems. One particular paradigm involves translating the natural language query into some formal database query language, which is then processed by the database system[CM86]. The most common such query language is SQL, a relational database language developed since 1970, which has gained popularity over the years. There have been many commercial applications involving front-ends ([Ove99], for example).

Front-ends (and natural-language interfaces in general) suffer from the same problems common to machine translation and other natural-language understanding systems. These include various forms of structural ambiguity, the usual lexical ambiguity, problems of discouerse analysis like anaphora, incomplete and extragrammatical sentences, and noncompositional compounds. Some of these problems are reduced due to the fact that database interfaces only handle a subset of natural language, and they have a limited domain of discourse[AR95].

Older interfaces were designed with a particular database system, language and domain in mind. The issue of portability is one that has been studied for many years: how to make interfaces that can easily be transferred to new knowledge domains, database management
systems, or natural languages. A related issue is the reuse of linguistic components, i.e.: adapting parsers or lexicons from other systems to natural-language interfaces.

This thesis introduces a new approach to natural-language interfaces, based on "Shake-and-Bake," a modular lexicalist machine translation method using unification-based grammar formalisms[Bea92b], to translate English queries into SQL. My approach requires an English grammar and lexicon, an SQL grammar and lexicon, and a bilingual lexicon which relates English primitives to SQL primitives. I introduce a prototype interface based on an existing machine translation system[Lau96b]. The English components were originally part of this system. I will limit my research to the translation of nominal compounds and complex nominals. These structures are common in natural languages and database applications[MPF96]. Nominal compounds have been extensively studied over the years; they may exhibit both syntactic and semantic ambiguity, as well as noncompositionality. I will explore the feasibility of a lexicalist natural-language database interface, as well as proposing and evaluating solutions to the basic linguistic problems mentioned above.

Another aspect of this research will be addressing issues of component reuse and domain portability. Assuming the English grammar and lexicon and SQL grammar are complete, all that is needed is to build a new bilingual lexicon and SQL lexicon. An algorithm for this process will be proposed, based on the concept of bilingual texts[Tur98].

This thesis is organized as follows. First I will provide background and history for database interfaces, and present the general issues involved. I will then give a brief introduction to the relational database model and SQL, as well as the example database model that will be used to illustrate my research. Following this are details on unification-based grammars in general and the proposed system's unilingual grammars in particular. The background section will continue with a discussion of the Shake-and-Bake method, and conclude with background on complex nominals and the linguistic problems associated with them.

In the first part of my research, I will present a possible arrangement for the bilingual lexicon. More specific problems (linguistic and computational) relevant to lexicalist interfaces will be discussed, as will the principles behind their solutions. First I will present a simple solution to demonstrate feasibility, then discuss possible ways to deal with lexical
ambiguity, structural ambiguity and noncompositional compounds. The solutions will be evaluated in terms of correctness, processing time, and complexity of the bilingual lexicon.

The problem of semi-automatic bilingual lexicon creation will then be addressed. The core of this chapter will be the description of an algorithm to produce bilingual entries from pairs of queries (for each pair, one is English, the other its SQL translation) The necessary size and other attributes of English queries will be discussed, as well as possible ways to accelerate the process. Generation of the SQL lexicon will also be discussed. Further methods for subsequent improvement of the bilingual lexicon will then be presented.
Chapter 2

Background

2.1 Natural Language Interfaces to Databases

2.1.1 Architectures

Traditionally, most natural language interfaces have followed the natural-language front-end paradigm [CM86] whereby natural language queries are translated into formal database queries (ie: in some formal query language). These queries are then processed by the database management system. The requested data could be returned as is, or could be passed back to the natural-language component, which generates a natural-language version. This effectively divides the problem of natural-language database access into two subproblems: the natural-language component, and the database access component.

The methods of translating natural languages to formal query languages have varied over the years. The Lunar Sciences Natural Language Information System (or LUNAR) is one of the oldest natural-language interfaces to databases: it performed a simple syntactic analysis of the English query, followed by semantic analysis of the resulting parse tree [WKW72]. Terry Winograd’s “Program for Understanding Natural Language” [Win72] incorporated semantic analysis directly into the parsing process. Though Winograd’s program was not a database front-end, it did translate commands and queries from English to a formal language, and the linguistic issues are the same.

Most of the front-ends that followed until the late 80’s used structural transfer (ie:
mapping of a parse tree into a database language expression][AR95]. More current systems, however, translate the English sentences into some (presumably system-independent) logical form, which is then mapped to the database query language. This is the case of MASQUE/SQL, which translates the English query into a Prolog-like form called LQL (short for “Logical Query Language”). The LQL form is then mapped into SQL[And92].

2.1.2 Portability

Early database interfaces were designed with a specific domain and database management system in mind, and thus could not be easily ported to new domains or systems. LUNAR, for example, is a front-end to a database containing chemical analyses of moon rocks which has its own query language, and takes several shortcuts to simplify translation. These shortcuts are of course highly domain-dependent. The most obvious one was to not translate the adjective “lunar”, because it is assumed to be redundant. Portability became a serious research issue in the early 80’s with CHAT-80[DP82]. Written in Prolog, CHAT-80 was designed as a general and portable natural language interface but, according to[And92], proved difficult to port. Later systems such as ASK[TT83], [TT85] allowed end-users to customize the interface by teaching the system new words.

The use of intermediate representations in more recent systems means that database portability has the potential to be quite easy, assuming that the logical form truly is independent of database language or system architecture. Similar observations can be made for natural language portability.

Knowledge-domain portability is the issue that interests us. Most, if not all, of the work will probably have to be done by a knowledge engineer: MASQUE, for example, has the ability to add words to the system’s knowledge. This is done through a dialogue between an engineer and the system, during which the engineer specifies (in the case of a verb) its various forms, the types of its subject, object, indirect object, and so on. Some systems facilitate this process by including lexicons and morphology rules. A related issue is the kind of experience required to port systems, such as: knowledge of the particular systems involved, natural language processing experience, and general knowledge of database issues[GAMP87].
2.1.3 Natural Language Coverage

One disadvantage of natural-language interfaces is that they can only handle subsets of natural language[AR95]. Furthermore, the linguistic coverage may be unclear to the end-user. MASQUE, for instance, can handle some kinds of conjunctions but not others, which leads to false expectations. It can understand the query "What are the capitals of the countries bordering Norway and bordering Sweden?" but not "What are the capitals of the countries bordering Norway and Sweden?". Contrary to the ideal interface that can be used immediately by any end-user, in practice there is usually some training required.

A related issue is one of linguistic vs. conceptual failures. In the former case, the issue is one of linguistic coverage (incorrect syntax in the natural language query, which may need to be rephrased). In the latter, the problem is with the database domain itself, which may have nothing to do with the topic of the query. This distinction may not be obvious to end-users[AR95].

2.1.4 Why Natural Language Interfaces?

Experiments have been carried out in the 80's and 90's, comparing end-user performances with natural-language interfaces, a formal language (SQL) and a graphical interface[BR92]. No interface could be said to be a clear winner. Conclusions from experts in the field are that "natural language is an effective method of interaction for casual users with a good knowledge of the database, who perform question-answering tasks, in a restricted domain"[CC90].

2.2 The Relational Database Model and SQL

2.2.1 The Relational Database Model

The relational model is a way to organize data in a simple and logical way. Conceptually, data is organized into tables, which represent either entities or relations between entities[ST95]. Various operations can be performed on these tables, to add, manipulate, or extract data.

**Tables and Attributes:** A model is a collection of tables. A table consists of a *name*, a *heading* (a set of relational attributes) and an *extension*: a set of rows. No heading can
CHAPTER 2. BACKGROUND

contain duplicate attributes, and extensions usually do not contain duplicate rows. There are also “catalog tables,” listing metadata about the model, such as the names, owners (relevant in a multi-user database) and some parameters of the headings (like domains, perhaps). Catalog tables are not directly relevant to my research, except possibly as the basis of bilingual lexicon generation, which is discussed in more detail in Chapter 4.

Operations on Tables In general, we modify and combine tables using specific techniques.

- renaming: This unary operation returns a table with the same extension as the first, a different name, possibly a different heading (i.e. different names for the attributes; but the domains must be the same).

- union: This binary operation returns a table with the same heading as the original two, and whose extension is the union of the original extensions, where union is the ordinary set operation. The tables must be compatible, i.e. have the same headings.

- intersection: This binary operation returns a table with the same heading as the original two, and whose extension is the intersection of the original extensions. The tables must be compatible.

- difference: This binary operation returns a table with the same heading as the original two, and whose extension is the difference of the original extensions. The tables must be compatible.

- product: This binary operation returns a table whose heading is the union of the original headings, and whose rows contain all possible combinations of the rows of the original tables. For example, consider the two following tables:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₄</td>
</tr>
<tr>
<td>a₃</td>
<td>b₁</td>
<td>c₁</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₁</td>
<td>e₁</td>
</tr>
<tr>
<td>d₂</td>
<td>e₁</td>
</tr>
</tbody>
</table>
CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_2$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_4$</td>
<td>$d_1$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_4$</td>
<td>$d_2$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_2$</td>
<td>$e_1$</td>
</tr>
</tbody>
</table>

Their product would be:

- **selection**: This unary operation allows us to select tuples in the extension that satisfy certain conditions. Although note that selection is unary only in the sense that it is applied to a single table, strictly speaking it applies to a table and a set of conditions.

- **projection**: This unary operation allows the user to choose only some (relevant) attributes. It returns a table whose heading contains only the specified attributes.

- **join**: This binary operation combines data residing in several tables to answer a query. It returns a table whose heading is the union of the original headings, and whose extension contains all of the tuples $t \bowtie s$ such that $t$ is a tuple in $T$ and $s$ is a tuple in $S$ (with $T$ and $S$ being the original tables) and $t$ and $s$ are *joinable* (ie: they have attributes in common, and the values of those attributes are equal). As an example, consider the following two tables. The first represents students, the second represents courses:

<table>
<thead>
<tr>
<th>stud_id</th>
<th>f_name</th>
<th>l_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12765</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>13846</td>
<td>Jane</td>
<td>Chen</td>
</tr>
<tr>
<td>15676</td>
<td>Joe</td>
<td>Grey</td>
</tr>
<tr>
<td>14448</td>
<td>Bill</td>
<td>Jones</td>
</tr>
<tr>
<td>12889</td>
<td>Ann</td>
<td>Smith</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stud_id</th>
<th>class_id</th>
<th>semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>12765</td>
<td>cmpt101</td>
<td>99-1</td>
</tr>
<tr>
<td>15676</td>
<td>cmpt150</td>
<td>99-1</td>
</tr>
<tr>
<td>15676</td>
<td>cmpt201</td>
<td>99-1</td>
</tr>
<tr>
<td>12889</td>
<td>cmpt275</td>
<td>99-1</td>
</tr>
</tbody>
</table>

The first tuple of the first table is joinable with the first tuple of the second. The second and third tuple of the second table is joinable with the third tuple of the first. The last tuple of the first table is joinable with the last tuple of the second. So, the join of the two tables is:
2.2.2 SQL

SQL, short for "Structured Query Language," is the most popular language for manipulating relational databases. It was developed in the mid-70's at the IBM San Jose Research Laboratory [ST95]. SQL is a non-procedural language, which means the query only has to state the information requested and the constraints. With SQL, one can perform selection and projection, renaming, as well as set-theoretical operations such as union. The structure of the basic query is:

\[ SELECT \text{ attributes} \ \text{FROM tables WHERE conditions} \]

\textit{attributes} indicates the attributes to be selected from the product of the tables specified in \textit{tables}. SQL cannot perform joins as such. They must be computed using products and projection. \textit{conditions} is the list of conditions involved in this projection; it must include conditions to specifically select only joinable tuples.

For example, the join given in subsection 2.2.1 would be described in SQL as:

\begin{verbatim}
SELECT stud_id, f_name, l_name, class_id, semester FROM students, classes
WHERE students.stud_id = classes.stud_id
\end{verbatim}

2.3 The Grammars

2.3.1 Unification-Based Grammars

Introduction

Grammar formalisms known as \textit{unification-based} [Shi86] derive their name from the basic operation on partial information structures: unification. Unification yields from a set of
compatible structures a structure which contains all of the information present in the members of the set, but no more. Thus structures become more and more fully specified in a monotonic fashion, through the interactions of various constraints from various sources. The grammar rules are all purely declarative (as opposed to procedural) in that they only specify constraints, and not the order of application of these constraints. Also they are reversible, in that they may be used equally well for parsing as for generation [Shi86].

**Feature Structures**

Informally, a feature structure is an information-bearing object that describes another object. In HPSG notation, feature structures are also known as *signs* [PS94]. In this thesis, I will use the two expressions interchangeably. A feature structure contains *attributes* (or *features*) which have corresponding *values*. Feature structures are said to provide partial information since not all of the features may have their values specified. One standard notation for feature structures is *attribute-value matrices* (or "AVM"). For example, here is a feature structure describing a class given at Simon Fraser University:

\[
\begin{bmatrix}
\text{COURSE} & \text{CMPT91} \\ 
\text{SEMESTER} & 99-1 \\ 
\text{ROOM} & \text{ASB10811} \\ 
\text{ENROLL} & 20
\end{bmatrix}
\]

(2.1)

Here the attributes are COURSE, SEMESTER, ROOM and ENROLL. Note that a feature’s value may be atomic (as is the case above) or it may itself be another feature structure. This property is called *hierarchality* [PS87]. For example, consider an additional attribute to the feature structure above, INSTRUCTOR, representing the course’s instructor.

\[
\begin{bmatrix}
\text{COURSE} & \text{CMPT91} \\ 
\text{SEMESTER} & 99-1 \\ 
\text{ROOM} & \text{ASB10811} \\ 
\text{ENROLL} & 20 \\
\text{INSTRUCTOR} & \begin{bmatrix}
\text{NAME} & \text{POPOWICH} \\ 
\text{OFFICE} & \text{ASB10831} \\ 
\text{PHONE} & 291-1112
\end{bmatrix}
\end{bmatrix}
\]

(2.2)
CHAPTER 2. BACKGROUND

It is sometimes convenient to have the notion of paths, which are finite sequences of attributes. It is easy to generalize the notion of the value of an attribute to the notion of the value of a path. For example, the value of \texttt{INSTRUCTOR | NAME} is \texttt{POPOWICH}.

There are different ways to implement these concepts. The present system's linguistic modules are written in ALE, a parsing and logic programming language. In this language, feature structures are bracketed by parentheses, features in each structure are separated by commas, and attributes and values in paths are separated by colons [CP94]. For example, the previous feature structure would be represented by:

\begin{verbatim}
(course:cmpt891, semester:'99-1', room:asb10811, enroll:20,
instructor:(name:popowich, office:asb10831, phone:'291-1112'))
\end{verbatim}

Alternatively, it could be represented by:

\begin{verbatim}
(course:cmpt891, semester:'99-1', room:asb10811, enroll:20,
instructor:name:popowich, instructor:office:asb10831,
instructor:phone:'291-1112')
\end{verbatim}

Feature structures may also have types, which are arranged in an inheritance hierarchy whereby type constraints of more general types are inherited by their subtypes [Car92]. In ALE, feature structures are typed.

\textbf{Subsumption and Extension}

Some feature structures are more informative than others. Consider the two feature structures already specified in this section. It is clear that (2.2) contains more information than (2.1). In this case, (2.2) is said to \textit{extend} (2.1), and (2.1) is said to \textit{subsume} (2.2). In general, if feature structure \( A \) is at least as informative as feature structure \( B \), then we say that \( A \) extends \( B \), and that \( B \) subsumes \( A \). The standard notation is \( A \preceq B \). In the case of atoms (ie: feature structures with no attributes) \( A \) and \( B \), the only way for \( A \) to subsume \( B \) is for \( A \) to be equal to \( B \). The subsumption relationship is reflexive (for every feature structure \( A, A \preceq A \)), transitive (if \( A \preceq B \) and \( B \preceq C \), then \( A \preceq C \)) and antisymmetric (if \( A \preceq B \) and \( B \preceq A \), then \( A = B \)). The extension relation is a reflexive partial ordering. A more formal definition of subsumption can be found in [PS87].
Another property which is important to applications is *structure sharing*. Two or more distinct attributes or paths may have their values specified to be the same feature structure. For example, we could add one more feature to classes and instructors, DEPARTMENT, and specify that for a given course, the value of DEPARTMENT and INSTRUCTOR | DEPARTMENT must be the same. This is represented in AVM's by the use of numbered tags in place of, or in addition to, feature values:

\[
\begin{align*}
\text{DEPARTMENT} & \quad \square \\
\text{INSTRUCTOR} & \quad \left[ \text{DEPARTMENT} \quad \square \right]
\end{align*}
\]  

(2.3)

In ALE notation, this feature structure would be represented by:

\[
(\text{department}:V, \text{instructor}:\text{department}:V)
\]

The "V" could be any variable name. ALE follows the Prolog notation of variables beginning with uppercase letters.

**Unification**

We have seen the definition for this operation earlier in this section. As an example, consider the two following feature structures:

\[
\begin{align*}
\text{COURSE} & \quad \text{CMPT891} \\
\text{SEMESTER} & \quad 99-1 \\
\text{DEPARTMENT} & \quad \text{CMPT} \\
\text{ROOM} & \quad \text{ASB10811}
\end{align*}
\]  

(2.4)

and

\[
\begin{align*}
\text{INSTRUCTOR} & \quad \left[ \text{NAME} \quad \text{POPOWICH} \right] \\
& \quad \left[ \text{OFFICE} \quad \text{ASB10831} \right] \\
& \quad \left[ \text{PHONE} \quad 291-1112 \right]
\end{align*}
\]  

(2.5)
CHAPTER 2. BACKGROUND

Suppose further that we knew they were describing the same object. Then we could unify them, into the following feature structure:

```
[ COURSE     CMPT91       ]
[ SEMESTER    99-1        ]
[ DEPARTMENT  CMPT        ]
[ ROOM        ASB10811    ]
[ INSTRUCTOR  [ NAME      POPOWICH ]
                [ DEPARTMENT CMPT ]
                [ OFFICE     ASB10831 ]
                [ PHONE      291-1112 ] ]
```

Although it is not shown in the above example, it is important to note that unification preserves any structure sharing that may be present in the original feature structures.

2.3.2 A Simple English Grammar

Introduction

The English grammar used in this system[1au96b] is inspired from the HPSG formalism. In this section I will examine the features relevant to my system (in nouns and adjectives), the macros used in the bilingual lexicon, as well as the syntactic structures used. Refer to Appendix A for a fuller discussion of the English grammar.

Features Used

Our feature structures contain a collection of syntactic features along with some primitive semantic features called indices.

- index, index0, index1, index2: these features are used to reference the sign itself and whatever complements or modifiers it may have. index references the sign itself, while the others (some or all of which may be null) share the values of the complements' index features. In particular, if the noun is modified by another noun, then the modifying noun is referenced by index2. If the noun is modified by an adjective, then the two words' index features share the same value.
CHAPTER 2. BACKGROUND

- **features**: itself a feature structure, its attributes contain various basic information about the sign such as its lexical form (features:wtype), its tense, if a verb (features:tense), and its number (features:number). There are two that interest us, both boolean features. features:has.head is set to “yes” if the noun modifies another noun, “nil” otherwise. features:has.modifier is set to “yes” if the noun is modified by another noun, “nil” otherwise. Another feature of features, though not directly relevant to the system, is features:pos, (short for “part of speech”). For nouns, this feature is set to “noun.” For adjectives, it is set to “adj.”

- **mod:sign**: refers to the sign’s modifiee (ie: head) if it exists. In particular, mod:sign:index shares the value of the head’s index. This applies to both adjectives and nouns.

**Macros**

Macros allow developers to specify a description once and then use a shorthand for it in other locations. Macro calls always begin with a “@” character. Several macros are used in the English sides of the bilingual entries. Among them are:

- **@noun**: This macro performs the basic work of setting features:pos to “noun,” for example, and the structure sharing so that there is agreement in number between the noun and its complement(s), if any. It also adds more constraints, such as the value sharing of its features:has.head with its head’s (if it exists) features:has.modifier, and between its index and the head’s index2.

- **@noun(A)**: This is the same as @noun, above, except that the noun’s index feature value is also set to, or accessed by, the macro’s argument.

- **@c.or.no.c.noun(A,B)**: Same as @noun(A) above (which in fact calls the macro @c.or.no.c.noun, which stands for “count [noun] or not count noun”) except that the sign’s index2 feature value is set to, or accessed by, the macro’s second argument.

- **@adj**: This macro sets up the necessary structure sharing between the adjective and its head.

For more information on macros, see Appendix A.
Syntactic Structures

Here is a tree describing the syntactic structure of a two-noun compound, where each node is a feature structure. Structures like these are generated by the grammar rules, which are included in Appendix A.

```
[INDEX
  INDEX2
  FEATURES
  MOD]
```

Here is a tree describing an adjective-noun compound which is generated by a similar grammar rule.

```
[INDEX
  FEATURES
  MOD]
```

```
[INDEX
  FEATURES
  MOD]
```

2.3.3 A Simple SQL Grammar

Front-ends generally do not require an explicit SQL grammar, since little to no analysis of SQL queries is required. Since I am adopting the Shake-and-Bake approach, however, the SQL query must be generated out of an unordered bag of lexical units, requiring the use
of an SQL grammar. This grammar is the only SQL component that does not need to be changed when porting to new database domains.

The SQL grammar (which appears in Appendix B) covers basic queries, of the form

\[
\text{SELECT \ columns FROM \ tables WHERE \ conditions}
\]

or

\[
\text{SELECT \ func_expr \ FROM \ tables WHERE \ conditions}
\]

columns is a series of attributes (either as a column or a table/column pair), separated by commas; func_expr is a function expression: a function name with a single attribute as argument. tables is a series of table names separated by commas. conditions can be conjoined or disjoined, and grouped together with parentheses.

Lexical Units and Basic Features

All signs (terminal or nonterminal) have three features that indicate their connections with other signs. These features are denoted index, index0 and index1. As in the English grammar the first one, index, is used to reference the sign itself. For nonterminals, only index is non-null (with one exception, as explained later). Similarly for table names, column names and column values. For keywords, index and index0 are non-null while index1 is null. Following are a number of tree structures allowed by the grammar rules, to illustrate the value sharing between index, index0 and index1 features. First, between the three main keywords and the highest level nonterminals:

For the "SELECT" keyword, index0 may reference the function call instead.
CHAPTER 2. BACKGROUND

The following tree illustrates the structure sharing between columns.

```
columns
  INDEX 1
```

```
| column | comma | column | comma | column |
| INDEX 1 | INDEX 1 | INDEX 1 | INDEX 1 |
```

All columns in the SELECT-FROM part of the query share the same `index` value. Similarly, all tables share another value for `index`. Note also that commas are the only lexical units which do not need to use any features.

The next two trees illustrate the structure sharing between conditions and connectors (i.e. "AND" and "OR" keywords):

```
| conditions |
| INDEX 1 |
| INDEX 0  |
```

```
| condition | AND | conditions |
| INDEX 1   | INDEX 1 | INDEX 1 |
| INDEX 0   | INDEX 0 | INDEX 0 |
```

```
| condition | AND | conditions |
| INDEX 1   | INDEX 1 | INDEX 1 |
| INDEX 0   | INDEX 1 | INDEX 0 |
```
Conditions, referring to a group of conjoined and/or disjoined conditions, is the only nonterminal sign to have two non-null indices: index0 shares the value of the first condition’s index, while index1 shares the value of the last condition’s index. These two indices are necessary since conditions may be grouped together with parentheses. Such a group may be referred as a single condition, whose index value is the shared index value of the matching parentheses in question. The opening parenthesis’s index0 shares the value of the first condition’s index. Similarly, the closing parenthesis’s index0 shares the value of the last condition’s index. All of this is necessary to ensure correct ordering at the generation stage. The following tree illustrates this situation:

As can be seen, connectors may have either two or three non-null indices. Each has different applications. In the first case, the connector’s index0 shares the value of the following condition’s index. This has the advantage of making generation much easier, since there is no need to reference more than one condition, but it makes the placement of that condition in the final query unpredictable. In the second case, the connector’s index0 and index1 share the value, respectively, of the preceding and following condition’s index features. This ensures that both conditions on either side of the connector are in fact adjacent after the query is generated.

Each condition consists of three parts: a column or table/column pair, a comparison operator ("="", ">"", "<") and a value expression (which may take the form of either a constant, or another table/column pair). This tree illustrates the value sharing among indices within conditions:
As can be seen, the comparison operator's index0 references the first table/column pair, its index1 references the value expression, while its index shares the value of the condition's index.

Lastly, I present a tree to illustrate the value sharing within table/column pairs (this applies to either part of a condition)

The overall structure of a table/column pair is the same as that of a condition: three signs, with a central one referencing both the others.

Additional Features

Tables use in_cond, a boolean feature which specifies whether or not the sign in question will be placed in the SELECT-FROM part of the query upon generation.

Macros

The SQL grammar uses a few simple macros to specify the signs' features. These include Oix/1, Oix/2, Oix/3, setting or accessing the index, index0 and index1 features, and
The Database Model

The database model I am using represents a university domain. Here are the tables and attributes I am using:

- **GR_STUDENTS** - grad students and relevant info
  attributes: stud_id, street_address, city, province, advisor, etc...

- **UND_STUDENTS** - undergrad students and relevant info
  attributes: stud_id, street_address, city, province, etc...

- **STUD_DEPT** - relates students and departments. This intermediate table is necessary, since there may be a many-to-many relationship.
  attributes: stud_id, dept_id

- **GRADES** - relates students, classes and grades
  attributes: course_id, semester, stud_id, grades, final_grades

- **CLASSES** - classes and relevant information
  attributes: course_id, semester, enrollment, prof_id

- **COURSES** - courses and relevant information
  attributes: course_id, department

- **DEPARTMENTS** - departments and relevant information
  attributes: dept_id, director, faculty

- **FACULTIES** - faculties and relevant information
  attributes: fac_id, dean

- **PROFESSORS** - professors and relevant information
  attributes: prof_id, dept_id
2.4 Shake-And-Bake

2.4.1 Introduction

Shake-and-Bake is a relatively recent lexicalist approach to machine translation, taking advantage of unification-based grammar formalisms. Though it was intended to deal with translation between natural languages only, it can also be used to translate between natural languages and artificial languages like SQL [Lau96a].

Shake-and-Bake was intended to overcome some of the difficulties associated with more traditional MT methods:

- **transfer-based**: portability and modularity is increased, since unilingual components may be developed independently of each other. Moreover, since it is based on unification theories, unilingual components may be used for analysis or generation, and bilingual components are usually bidirectional or can easily be made so. This modularity is an advantage over structural transfer-based methods, and makes Shake-and-Bake in some respects more like an interlingua-based method.

- **interlingua-based**: the subset problem is related to interlingua-based systems (ie: those using an intermediate representation that is language-independent). See [Bea92b] and [Lan87] for more details. In brief, a robust IL system must guarantee that an IL expression derived from any source language (SL) expression can generate a target language (TL) expression. Then there will be several formulae in that IL that are equivalent to the one produced by the analyzer. It cannot be guaranteed that this formula is covered by the TL generator.

- **isomorphic grammars**: This involves writing unilingual grammars and lexicons in which a tight correspondence is kept between pairs of grammar rules (for each language) and pairs of lexical entries [Bea92a]. As a consequence, no transfer module is required. But this approach has been found to be impractical when dealing with systems translating between many languages.
2.4.2 Translation Process

The translation process for Shake-and-Bake involves at least three distinct stages: parsing (with an SL grammar and lexicon), transfer (with a bilingual lexicon, or "bilex") and generation (with a TL grammar and lexicon), as shown in figure 2.1. There might also be some preprocessing and postprocessing.

**Parsing**

The SL sentence is first parsed with a unilingual grammar and lexicon, producing a parse tree. As a side effect, a bag containing the leaves of the trees is produced. These leaves are feature structures, whose features were instantiated during the parsing process [Bea92a].

**Transfer**

Given the bag of source language signs, a corresponding bag of target language signs is produced with the necessary bilingual lexicon, which "establishes relationships (via unification) between sets of lexical entries in the various languages" [Bea92a]. Though originally these relationships were one-to-one (common when dealing with relatively similar natural languages), they can also be many-to-one, one-to-many, or many-to-many. In fact, for SQL translation, this will have to be the case.

In the system with which I am working, the bilingual lexicon is composed of transfer rules (or "trules", for short), the structure of which is illustrated in the following sample trule:
trule_u(class, (class :: @noun
    < - >
    class.id :: (column_name, @ix(S)) &
    classes :: (table, @ix(T), @in_cond(nil))
\trans_select(S,T))).

The "< - >" symbol connects the left (English) and right (SQL) sides of the entry. The ":::" symbol connects a word and its description. This description can contain macros (introduced by the "@" character) or paths (ie: sequences of attributes, not seen in this example). The ":::" symbol (not seen in this example) connects with a word already found in the source language bag (if it is in the LHS) or the target language bag (if in the RHS), that is neither consumed nor produced by this particular entry. This is called constraining, since its purpose is to constrain transfer by providing value sharing with other lexical units contained in the bag. The "\\" introduces a transfer macro, which performs additional transfer operations, thus helping to keep the number of trules low and reduce information clutter. It is possible for a trule to consume more than one lexical unit. This is known as coconsumption. (So if the above rule were used right to left, it would exhibit coconsumption.)

Finally, it is possible to leave the words themselves unspecified. Unspecified words are represented in trules by an underscore ("_"), following the Prolog notation. Trules that contain unspecified words are known as templates [Tur98].

Generation

Finally, a target language sentence is generated from the bag, with the help of the (unilingual) TL grammar and lexicon. The final word order is decided by the TL signs’ feature values and the TL grammar. Generation in a way is very similar to parsing. One way to think about the generation process is to consider each permutation of the TL signs and try to parse it [Bea92a].
Backtracking

The translation process could be blocked at any stage: the query is impossible to parse (stopped at the parsing stage); the SL bag resulting from the parsing is inconsistent with the LHS constraints in the trules, or the TL bag is inconsistent with RHS constraints (stopped at the transfer stage); the TL bag cannot generate to a syntactically correct query (stopped at the generation stage). When any of these errors happen, the system will backtrack to the next most recent choice point and look for the next solution.

2.5 Nominal Compounds and Complex Nominals

I am going to address the problem of translating compound nominals into SQL. The problem has been examined in the past, and we have come up with a novel approach that makes use of the Shake-and-Bake paradigm. First, we must consider some linguistic and computational background.

2.5.1 Linguistic Background

Levi [Lev78] distinguishes between three overlapping types of what she terms "complex nominals": nominal compounds (ie: noun-noun compounds), nominalizations (in which the noun is derived from another type of word, such as a verb), and noun phrases with nonpredicating adjectives (also called nonpredicating noun phrases). Nonpredicating adjectives behave in most ways like modifying nouns: ie, they cannot be modified by adverbs, their meaning changes depending on their head noun, and in some cases they may even be interchanged with the noun from which they were derived. Some examples include rural policeman and solar generator. Also, contrast the nonpredicating noun phrase linguistic difficulties with the nominal compound language difficulties.

I will use the abbreviation "NC" (short for "nominal compound") to refer to all three forms of complex nominals (since I will be studying them interchangeably). Though perhaps not absolutely accurate, it will nevertheless help avoid confusion with another term abbreviated as "CN" introduced later in this thesis.

Some compounds are known as lexicalized compounds ("LC" for short). The semantic information of these compounds does not derive compositionally from the information of
Figure 2.1: An overview of the Shake-and-Bake paradigm
their component words. They rarely have world-for-word equivalents in other languages, and must be included in separate entries in unilingual lexicons. In MT applications, they require separate entries in bilingual lexicons as well.

2.5.2 Ambiguity Resolution

Ambiguity resolution is an important part of NLP. Notable sub-areas are POS tagging (for sense disambiguation), and structural ambiguity resolution. POS tagging is not a serious issue for this research, since I am only dealing with nouns and adjectives.

[Fra96] only mentions structural ambiguity resolution techniques for PP (prepositional phrase) attachment, but they can be adapted to simple nominal compounds. There are three basic approaches to choose from: syntactic, semantic, and pragmatic.

Syntactic approaches use structural properties of the parse tree to choose a particular analysis (examples: right association, early closure, etc...) Semantic disambiguation involves using lexical knowledge and "common sense" heuristics. These heuristics use some form of domain knowledge or semantic features in the words or parse tree nodes. Pragmatic approaches use general world knowledge or specific domain knowledge.

Since the system does not contain any world knowledge, and cannot directly access parse tree information, only semantic disambiguation approaches will apply.
Chapter 3

Translation of Nominal Compounds

3.1 Specific Problems

3.1.1 Linguistic Problems

The general problem associated with the translation of nominal compounds is to provide a mapping between natural language phrases and SQL. Not only is the problem non-compositional, but it is also ambiguous. In the particular case of database front-ends, ambiguity means that the noun (or expression) refers to different tables, columns in different tables, different columns in a single table, or different values for a column. Compositionality (in this case) can roughly be thought of as the ability to monotonically add SQL lexical units whenever a word or group of words is added to the English query. In all cases, the presence or lack of compositionality or ambiguity depends on the database model. Let us consider the specific problems associated with mapping nominal compounds to their database counterparts.

The first problems deal with lexicalized compounds and other forms of noncompositionality existing within larger compounds (ie: the query), the second and third deal with two different types of lexical ambiguity, while the last is related to more general problems of mapping natural language to SQL.

1. *Head noun meaning is changed by modifier*

   In this situation, the noun translates to a particular table, column or column value (or set thereof), but when modified will translate to a different set. An example of this
is "grades" vs. "final grades." The former is a request for several columns (assuming that "grades" means all the grades for a given course), the latter is a request for one column, not necessarily included in those covered by "grades." Note that the noun in question need not act as the head of the entire query but only of some nominal compound contained within the query. This compound is noncompositional and, like its head, may or may not be ambiguous.

2. *Ambiguous head noun, meaning is decided by modifier*

The noun translates to a set of tables, columns or column values, which the modifier reduces. Note that this is another example of noncompositionality. An example for tables is "student" vs. "grad student." For example, in the database model that was presented in section 2.3.3, there is a table for grad students and another for undergrad students. "Student" unmodified by "grad" or "undergrad" will then refer to both tables. This form of ambiguity will be known as *noncompositional ambiguity*.

3. *Ambiguous modifier, meaning is decided by head*

Here, the expression translates to one of several possible tables, columns or column values; the head disambiguates between these possibilities. Many modifiers would actually be ambiguous in this fashion. "Linguistics," for example, adds different information depending on whether it is modifying "student," "course," or "professor." This is also a common form of ambiguity for similarly-named attributes of different tables. "Name," for example, could refer to the names of grad students, undergrad students or professors, which are all different tables. Note that in this case, the compound is compositional, and so this form of ambiguity will be known as *compositional ambiguity*.

4. *Structural ambiguity*

Some nouns can act as heads or modifiers, and will naturally have different SQL translations depending on their roles. This lexical ambiguity is related to structural ambiguity, since how a word is parsed will influence its translation.

### 3.1.2 Computational Issues

There are several criteria that must be kept in mind when building the bilingual lexicon. Unfortunately, some of them are mutually incompatible, which means a balance must be
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

found. The criteria are:

1. Correctness
   
The English query must translate to a correct SQL equivalent, every time.

2. Complexity of the bilingual lexicon
   
The bilingual lexicon must contain the smallest number of transfer rules possible.

3. Processing time
   
   Though this issue is partially related to bilex complexity, in that a larger bilex will take longer to search, it also includes the problem of reducing backtracking.

4. Ease of lexical generation
   
   This refers to the building of the bilex when porting to a new domain. A bilex that follows consistent guidelines for its rules will of course be easier to generate automatically.

5. Readability of the bilingual lexicon
   
   Though this is not a major issue for this present research, and would not affect performance or usability, it is desirable to ensure the rules are kept as simple as possible for programmers and developers.

3.2 Basic Solution

3.2.1 Overview

In this section I will describe the basics of my solution, to demonstrate its feasibility, as well as providing a number of examples to show how word type and context affect translation. In brief, each transfer rule's LHS will consist of one expression (which may be an individual word or a lexicalized compound) which is to be consumed, constrained with one or more expressions making up its context. This context will depend on the expression's type and role, and will contain enough information to resolve compositional ambiguity (ie: linguistic problem 3). This constraining will have the side effect of only allowing certain parses to be translated (other parses will fail at the transfer stage), thus removing some structural ambiguity.
LHS signs' features are underspecified. In particular, features related to number and other morphological variations are not specified, allowing different variations of the same noun to be translated with the same rules.

The rules' RHS's will contain the consumed expression's SQL equivalent, in some cases constrained with other SQL lexical units. This is useful in preventing some backtracking and thus lowering processing time (inconsistencies that would be caught at the generation stage are now caught at the transfer stage).

3.2.2 Left-Hand Sides

Individual Expressions

Expressions on the left-hand sides correspond to either individual words or lexicalized compounds. Their main characteristic is that they be built noncompositionally, and interact compositionally with other expressions. In effect, they are the "basic building blocks" of the nominal compounds.

Word Types

Nouns have been divided into 4 categories, based on their database equivalent. These categories will influence role and context. The categories are: Table Name (TN), Column Name (CN), Column Value (CV) and Function (F).

- **TN nouns** refer to tables in the database ("department," "faculty," etc...). These nouns are the most flexible when it comes to context: they can act as heads for nouns and adjectives of all types, and can modify nouns of all types (except CV nouns).

- **CN nouns** refer to a column in the database ("address," "enrollment," "semester," etc...). CN nouns can be modified by adjectives, and by TN or CV nouns. In the case of TN nouns, the modifier usually refers to the name of the table to which the CN noun also refers. A CV noun can refer to either a value of the noun's equivalent's column, or some other column entirely (usually the key column of another table). They cannot be modified by another CN noun because there would be no direct semantic relationship between the two nouns. Possible exceptions, not found in my database model, include the CN noun referring to another table with similar attributes. For
example, an additional attribute of both student tables might be “TA”, referring to each student’s TA(s). In that case, a valid query, consisting of two CN nouns, would be “TA names.”

A modifier CN noun (whose head can only be a TN noun) must itself be modified by either a CV noun or a predicating adjective. Otherwise, the query would be incomplete.

Note that whether a given noun refers to a table or a column depends on the database model.

- **CV nouns** refers to a value for a particular column (“99-1” for “semester,” etc...). Since they are not themselves entities or attributes, these nouns cannot act as query heads. In fact, they cannot act as heads at all. In queries consisting only of complex nominals, they can only be modifiers.

- **F nouns** refer to an SQL function (count, average, etc...) They are independent from considerations of context with other words or expressions, because they always act as query heads, and can be modified by any noun that can also act as query head.

Adjectives, as mentioned previously, are divided into two types: predicating and non-predicating. Predicating adjectives have the same possible contexts as CV nouns. Nonpredicating adjectives are part of lexicalized compounds, and thus will not be considered when determining possible contexts.

**Context and Constraints**

The basic criterion for constraints on trules’ LHS’s is to ensure correctness. One possible solution is to constrain modifiers (whether nouns or adjectives) with their head, but not with their own modifier(s). This is desirable because nouns are likely to have more possible modifiers than heads: TN nouns’ possible modifiers include all the CN nouns referring to their attributes, CN nouns’ modifiers include all CV nouns referring to their values, and so on. Furthermore, a single direction of constraining (i.e: exclusively towards heads or modifiers) is easier to generate automatically when building the bilingual lexicon. In the case of compositional expressions (whether ambiguous or not) there is no need for coconsumption with their context, since the relevant expressions will have their own trules.
The exceptions are:

- Lexicalized compounds, all the words of which are coconsumed in the same rules. This applies whether the compound is consumed or only constrained. The compound will behave as its head noun, assuming that noun has a particular meaning by itself (i.e., it will take on its head's type).

- Query heads and their immediate modifiers (a single expression only), which are co-consumed. Since the head could only be constrained with its modifier, and thus both expressions are each other's context, it would be superfluous to have a rule for each.

- Predicating adjectives modifying CN nouns, which may also have to be constrained with the CN noun's TN noun head. Similarly-named attributes may be found in different tables, which would affect their meaning. "High enrollment" means something different when modifying "department" than when modifying "course" (for example). Without adding the TN noun to the adjective's context, there is no way to guarantee that the correct translation will be chosen, since it would still result in a grammatically correct SQL query.

### 3.2.3 Right-Hand Sides

#### Query Heads

English query heads and their immediate modifiers (whatever their type) will translate to the SELECT-FROM part of the SQL query; this can be considered to be the "head" of the SQL query, and the conditions to be the modifier. Depending on their type, the two words may translate to a number of conditions. The only case where this would not happen is if the head were a CN noun and its modifier were a TN noun.

**Example 3.1** “Department enrollment” would translate to

```
SELECT enroll FROM departments
```

**Example 3.2** “99-1 classes”

This is another simple example, containing a head TN noun modified by a CV noun. Both nouns happen to correspond to the same table, though this is certainly not a requirement. The corresponding SQL query is:
SELECT class.id FROM classes WHERE classes . semester = 99-1

As can be seen, the translation contains conditions. A head TN noun translates to a request for the key attributes of the corresponding table.

The trule necessary to translate this is:

\[
\text{trule.\_u(class, (}
\begin{align*}
'99-1' &:: (@\text{noun, features:has\_head:yes, index:12, } \text{mod:sign:index:1}) \& \\
class &:: (@\text{noun, features:(has\_modifier:yes, has\_head:nil), } \\
\quad &\text{index:1, index2:12}) \\
< &\rightarrow \\
course\_id &:: (@\text{column, @ix(S)}) \& \\
',' &:: (@\text{comma}) \& \\
semester &:: (@\text{column, @ix(S)}) \& \\
classes &:: (@\text{table, @ix(T), @in\_cond(nil)}) \& \\
\text{classes} &:: (@\text{table, @ix(D0), @in\_cond(yes)}) \& \\
'.' &:: (@\text{ix(B0,D0,E0)}) \& \\
semester &:: (@\text{column, @ix(E0)}) \& \\
'=' &:: (@\text{ix(A0,B0,C0)}) \& \\
'99-1' &:: (@\text{constant, @ix(C0)}) \\
\text{\backslash\text{\backslash trans.select}(S,T)} \\
\text{\backslash\text{\backslash trans.cond\_and}(A0))).
\end{align*}
\]

It would be useful at this point to draw attention to this trule's important characteristics. The LHS is composed of two nouns, coconsumed. The first is "99-1," a CV noun, the second is "class," a TN noun. "99-1" is specified as having a head; whether or not it has a modifier is unknown (that feature is left unspecified). "Class" is specified to have a modifier, and not to have a head. This way, we know it is the head of the query. The \text{mod:sign:index} feature of "99-1" (indicating its head) shares the value of the \text{index} feature of "class." Conversely, the \text{index2} feature of "class" (indicating its modifier) shares the value of the \text{index} feature of "99-1."
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

The RHS introduces a number of items corresponding to different components of the query to be generated. The first two lexical units will be used to generate the SELECT-FROM part of the SQL query. This is accomplished with the trans.select/2 transfer macro. Its arguments are a) the index value of the columns (which is shared), and b) the index value of the tables (which is shared). This macro adds to the TL bag the keywords SELECT and FROM, with the proper index values (ie: both keywords have the same index values, and their index0 are, respectively, the first and second arguments of the macro). Note that the in_cond feature for “classes” is set to “nil,” meaning that it is not in the conditions part of the query. The macro table.name specifies its table.name feature to be “classes.” Recall that this information will be used by the SQL grammar rules at the generation stage for simple semantic constraints.

The second instance of “classes” in the RHS is the one that will be used to generate the condition. Note that its in_cond feature is set to “yes.” The RHS ends with the tmacro trans.cond.and/1 that produces the lexical unit WHERE, properly indexed with the macro’s argument, the index value of the “=” comparison operator. In general, trans.cond.and/1 takes as argument the index of a condition (ie: the index value of an opening parenthesis or a comparison operator) and adds to the TL bag either a WHERE keyword (if such a keyword is not already present) or an AND keyword with the proper index values (if the bag already contains a WHERE).

In either case, the keyword’s index0 feature shares the value of the condition’s index. If a WHERE is added, its index feature also shares the value of the SELECT and FROM’s index.

Example 3.3 Now let us consider the query “grad students’ grades”

This is more or less the same situation as the previous example, except for two points. First, “grad student” is a lexicalized compound, which means these two words will always be consumed together. Also, since “student” is a TN noun, the entire lexicalized compound will act as a TN noun. As it modifies another TN noun, there will automatically be join information added to the query. Not only is it necessary for this particular query, but as we shall see later it will also be important if “grad student” is itself modified.
The resulting query is:

```sql
SELECT grades.num FROM grades, gr_students
WHERE grades.student_id = gr_students.student_id
```

The single rule required to translate this query is:

```prolog
true_u(grade, (
  grad :: (features:pos:adj, index:12, mod:sign:index:12) &
  student :: (Onoun, features:has_head:yes, index:12, mod:sign:index:1)
  grade :: (Onoun, features:(has_modifier:yes, has_head:nil),
    index:1, index2:12)
  )< - >
  grade_num :: (column, @ix(S)) &
  gr_students :: (table, @ix(T), @in-cond(nil)) &
  ',' :: comma &
  grades :: (table, @ix(T), @in-cond(nil)) &
  gr_students :: (table, @ix(D0), @in-cond(yes)) &
  '.' :: @ix(B0,D0,E0) &
  stud_id :: (column, @ix(E0)) &
  '=' :: @ix(A0,B0,C0) &
  grades :: (table, @ix(F0), @in-cond(yes)) &
  '.' :: @ix(C0,F0,G0) &
  stud_id :: (column, @ix(G0))
\\trans_select(S,T)
\\trans_cond_and(A0))).
```

The two main differences with the current example from the previous example are as follows: first, the LHS contains three words. The first two ("grad," a nonpredicating adjective, and "student," a TN noun) form a lexicalized compound. The third, "grade," is a TN noun. "Student" is modified by "grad" while "grade" is modified by "student." Note again that "grade" is specified to not have a head itself, while "student" (and, by extension,
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

the compound "grad student") may or may not be modified (the features:has_modifer
feature of "student" is not specified). This is much more important than in the previous
example, because as a CV noun, "99-1" could not be modified. But a TN noun may, and we
want this trule to transfer the words correctly whether or not "grad student" is modified.
As an adjective, "grad" could not be modified by another noun.

Second, the trule’s RHS contains four instances of tables. The first two, "grades" and
"grad student," will end up in the SELECT-FROM part of the query (their in_cond feature
is set to "nil"). The trule also produces a comma, to complete that part of the query. The
third and fourth tables will end up in the one condition making up the conditions part.

Modifying TN Nouns

Modifying TN nouns (as long as they are unambiguous or compositionally ambiguous) will
translate to one or more conjoined conditions. Their role is primarily to add the necessary
join information to the query.

For example, "department" when modifying "professor" (two TN nouns) will create the
lexical units for a join condition between the corresponding tables. More precisely, the trules
will create the necessary lexical units for the following join condition:

```
professors . dept_id = departments . dept_id
```

But when modifying "courses," it translates to

```
courses . department = departments . dept_id
```

It is clear that the bilex must know a TN noun’s head when translating it, because that
is where the join information lies. Without this context, the system could produce incorrect
queries that would still generate correctly.

Recall that TN nouns can also modify CN nouns. But this only happens when the CN
noun is the head of the query, and the TN noun will be consumed together with its head.
Thus questions of context and this form of lexical ambiguity become unimportant.

Example 3.4 "linguistics department professors’ classes enrollment"
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

This query contains multiple TN nouns and is intended to show how translation can occur with a minimum of constraining. Four rules will be required to translate this query: the first one, as always, consuming the head and its modifier ("enrollment" and "classes"). The second consumes "professor" constrained with "class," the third "department" constrained with "professor" and the fourth "linguistics" constrained with "department." These rules' outputs will be, respectively:

```
SELECT enroll FROM classes
WHERE classes . prof.id = professors . prof.id
AND professors . dept.id = departments . dept.id
AND departments . dept.id = linguistics
```

Strictly speaking, the "WHERE" and "AND" keywords are not part of the conditions. However, they are added to the TL bag at the transfer stage by the `trans_cond_and/1` macro.

The final SQL query, generated from this bag of lexical units, is:

```
SELECT enroll FROM classes , class.prof , professors , departments
WHERE classes . prof.id = professors . prof.id
AND professors . dept.id = departments . dept.id
AND departments . dept.id = linguistics
```

Notice that this query is not optimal. I could have merged those two last conditions, into `WHERE professors . dept.id = linguistics`. But that would have required additional constraining or coconsumption. The query is still correct, however.

First, let us examine the rule that consumes "class enrollment":

```
trule_u(enrollment, ( 
  class :: (@noun, features:has_head:yes, index:12, mod:sign:index:1) &
  enrollment :: (@noun, features:(has_modifier:yes, has_head:nil),
    index:1, index2:12)
```


CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

< - >

enroll :: (column, @ix(S)) &
classes :: (table, @ix(T), @in_cond(nil)) & 
\trans_select(S,T)).

As we can see, this rule produces the lexical units for the SELECT-FROM part of the query only, not for any conditions. Therefore, it will only contain one tmacro call, trans.select/2. The other tmacro's would be relevant only if the rule generated lexical units for a condition.

Now let us examine the rule that consumes “professor”:

true.u(professor, (  
   professor :: (@noun, features:has_head:yes, index:12, mod:sign:index:1) &
   class :: (@noun, features:has_modifier:yes, index:1, index2:12)
   < - >
   classes :: (table, @ix(D0), @in_cond(yes)) &
   '.' :: @ix(B0,D0,E0) &
   prof.id :: (column, @ix(E0)) &
   '=' :: @ix(A0,B0,C0) &
   professors :: (table, @ix(F0), @in_cond(yes)) &
   '.' :: @ix(C0,F0,G0) &
   prof.id :: (column, @ix(G0))
   \trans_tables(D0,classes)
   \trans_tables(F0,professors)
   \trans_cond_and(A0)).

This rule constrains “professor” with its head, “class.” Note that “professor” is specified to have a head, but whether or not it has a modifier is left unspecified. In fact “professor” is modified, but that must be irrelevant in this limited context. “Class” is specified to have a modifier, but whether or not it has a head is unspecified. Again, “class” does indeed have a head in this case, but the rule should not need to know it.
This trule also calls the \texttt{trans\_tables/2} transfer macro, whose arguments are an index value and a table name. Its role is to ensure that all table names found in the conditions part of the query will also be found in the tables list part. For each table name, there are two definitions of the macro. The first one only constrains with the \texttt{FROM} keyword and a referenced (from \texttt{FROM}) table name, and thus adds nothing to the bag. The second definition automatically adds the table to the bag, with proper feature values (ie: index sharing value with \texttt{FROM}'s index0, and \texttt{in\_cond} set to \texttt{nil}). This second definition will come after the first in the transfer macros file. This ensures that it is checked last, and a table is added only if needed.

\texttt{trans\_tables/2} is neither necessary nor particularly desirable for trules that consume query heads. In that case, it is just as easy to directly add all the tables that will be needed in the \texttt{SELECT-FROM} part. Furthermore, putting in calls to \texttt{trans\_tables/2} instead of producing tables directly will possibly mean the TL bag generates an incomplete/incorrect query (because of missing tables in the \texttt{SELECT-FROM} part), which will cause extra backtracking. However, the tmacro call (one for each table that will end up in the conditions) is necessary for trules that consume only modifiers.

Let us now consider the trule that consumes "department":

\begin{verbatim}
trule_u(department, ( 
  department :: (@noun, features:has\_head\_yes, index:12, mod:sign:index:1) &  
  professor :: (@noun, features:has\_modifier\_yes, index:1, index2:12)  
  < -- >  
  professors :: (table, @ix(D0), @in\_cond\_yes)) &  
  
  '.' :: @ix(B0,D0,E0) &  
  prof\_id :: (column, @ix(E0)) &  
  '=' :: @ix(A0,B0,C0) &  
  prof\_dept :: (table, @ix(F0), @in\_cond\_yes)) &  
  
  '.' :: @ix(C0,F0,G0) &  
  prof\_id :: (column, @ix(G0)) &  
  'AND' :: @ix(And0,A1) &  
  prof\_dept :: (table, @ix(D1), @in\_cond\_yes)) &
\end{verbatim}


```
'. ' :: @ix(B1,D1,E1) &
department_id :: (column, @ix(E1)) &
'=' :: @ix(A1,B1,C1) &
departments :: (table, @ix(F1), @in_cond(yes)) &
'. ' :: @ix(C1,F1,G1) &
department_id :: (column, @ix(G1))
\trans_tables(D1,prof_dept)
\trans_tables(D0,professors)
\trans_tables(F1,departments)
\trans_and(A0)).
```

Similar comments can be made for this rule as for the previous one. A minor difference
is that the lexical units for not one but two conditions are produced. Since three tables
are involved, the rule will include three calls to the trans_tables/2 macro. Only one
\trans_and/1 call is needed, however. Its argument is the index of the comparison
operator of the first condition.

Let us now look at the last rule, consuming “linguistics” (it needs to be so specific,
since the English grammar does not include semantic or orthographic features):

```
trule_u(linguistics, (
  linguistics :: (@noun, features:has_head:yes, index:12, mod:sign:index:1) &
  department :: (@noun, features:has_modifier:yes, index:1, index2:12)
    < - >
  departments :: (table, @ix(D0), @in_cond(yes)) &
  ' ' :: @ix(B0,D0,E0) &
  department_id :: (column, @ix(E0)) &
  '=' :: @ix(A0,B0,C0) &
  linguistics :: (constant, @ix(C0))
\trans_tables(D0,departments)
\trans_and(A0)).
```

This rule consumes “linguistics,” a CV noun, constrained with its head, “department,”
a TN noun. It produces the lexical units for just one condition, involving one table (hence only one call each to trans.tables/2 and trans.cond.and/1). There will be more on CV nouns in following sections.

**Modifying CN Nouns**

Modifying CN nouns (which can only modify TN nouns) will translate to partial conditions: either a column name, or a table/column pair. It is possible to constrain those lexical units with an unspecified comparison operator, which would be provided by the CN noun's modifier. That particular entry in the rule must be left unspecified since we do not know which operator it is. Though not strictly necessary as far as correctness is concerned, this constraining will be useful in preventing backtracking: in case the query is incomplete (say, that there is no modifier for the CN noun) the error will be detected at the transfer stage, when the inconsistency is found in the SQL bag, instead of at the generation stage when the system tries to build a syntactically incorrect query.

Example: “semester” when modifying “class” will translate to the partial condition 

```
WHERE course . semester
```

**Example 3.5 “99-1 semester classes”**

This query is semantically identical to (example 3.1) (since “99-1” is a value of the attribute “semester”) so we would naturally expect the corresponding SQL query to be the same also. However, the rules necessary to translate it are very different. For one thing, there are two of them, not one. The first rule will consume “semester” and “classes,” with an output of

```
SELECT class.id FROM classes WHERE classes . semester
```

Recall again that after the transfer stage, the constituents are not in any particular order. The second rule will consume “99-1” constrained with “semester.” with an output of

```
= 99-1
```

And all of this gives us the following SQL query:

```
SELECT class.id FROM classes WHERE classes . semester = 99-1
```
Which is exactly what we want. Note that we can substitute any value for the semester we want here, and only the second trule will have to change.

Now, let us examine the trule consuming "semester" and "class."

\[
\text{trule}_{\text{u}}(\text{class}, ( \\
\text{semester} :: (@\text{noun}, \text{features:has\_head:yes, index:12, mod:sign:index:1}) & \\
\text{class} :: (@\text{noun}, \text{features:(has\_modifier:yes, has\_head:nil)}, \\
\qquad \text{index:1, index2:12}) \\
\quad \quad \text{< - >} \\
\text{course\_id} :: (\text{column, @ix(S)}) & \\
\text{', ' :: comma } & \\
\text{semester} :: (\text{column, @ix(S)}) & \\
\text{classes} :: (\text{table, @ix(T), @in\_cond(nil)}) & \\
\text{classes} :: (\text{table, @ix(D0), @in\_cond(yes)}) & \\
\quad \quad \text{'} :: @ix(B0,D0,E0) & \\
\text{semester} :: (\text{column, @ix(E0)}) & \\
\_ :: (\text{comparison\_op, @ix(\_,B0) \\
\quad \\quad \text{\textbackslash trans\_select(S,T))}). \\
\]

This trule produces the lexical units for the SELECT-FROM part of the query, and the first half of the query's single condition. Only one feature for the comparison operator is specified: \text{index0}, sharing the value of the table/column pair's \text{index} feature (more specifically, it is the period's \text{index} feature). The operator's \text{index} and \text{index1} features must be left unspecifed since they will be sharing value with other lexical units' features, as described in the second trule.

Just as trule in previous examples produced multiple tables to place in the SELECT-FROM part of the query, so this one produces multiple attributes. These attributes' \text{index} feature value is shared, just as in the previous case, and a comma is produced to ensure the query is grammatically correct.

Let us now examine the second trule, consuming "99-1":
trule_u('99-1', (    '99-1' :: (Qnoun, features:has_head:yes, index:12, mod:sign:index:1) &    semester :: (Qnoun, features:has_modifier:yes, index:1, index2:12)    < ->    classes :: (table, @ix(D0). @in-cond(yes)) &    '.' :: @ix(B0,D0,E0) &    semester :: (column, @ix(E0)) &    '=' :: @ix(A0,B0,C0) &    '99-1' :: (constant, @ix(C0))    \trans_tables(D0,classes)    \trans_cond_and(A0))).

Note that “99-1” only needs to be constrained with its head, and no other items from the SL bag. This trule produces the comparison operator, as well as the second part of the condition, constrained with the lexical units produced by the first trule. It is here that the trans_cond_and/1 tmacro is called, since the comparison operator’s index feature is specified.

CV Nouns and Predicating Adjectives

When modifying TN nouns, these expressions will translate to whole, conjoined conditions. When modifying CN nouns, they will translate to a partial condition to complete the CN noun’s translation (ie: it will be a comparison operator and column value expression). These lexical units will need to be constrained with the rest of the condition, to ensure the correctness of the final query. If such constraining did not take place, and if one had two or more CV nouns or predicating adjectives modifying CN nouns, then the corresponding SQL lexical units could be swapped at the generation stage, leading to a semantically incorrect query.

Examples:

- “large” when modifying “department” would translate to
  WHERE departments . enroll > 1000 (or some other number).

- “Linguistics” when modifying “grad students” would translate to
WHERE gr_students . stud_id = stud_dept . stud_id
AND stud_dept . dept_id = 'linguistics'

As we have already seen, "99-1" when modifying "semester" when modifying "class,"
will translate to = '99-1'. But, when directly modifying "semester," will translate
to WHERE classes . semester = '99-1'.

Example 3.6 "high | low enrollment classes"

This is another example involving predicating adjectives modifying a CN noun modifying
a TN noun. I will assume for now that "enrollment" is not ambiguous (ie: the attribute
may be found in only one table). This will reduce the necessary amount of constraining
for the rules' LHS's. As was noted in section 4.2 of this chapter, in case of ambiguity the
adjective would have to be constrained not only with its head, but with that head's head
as well in order to resolve the ambiguity.

This "family" of queries will be covered by only three rules. The first consumes "en-
rollment" and "classes." The other two consume "high" and "low" respectively, constrained
with "enrollment".

The rules' outputs will be:

SELECT course_id, semester FROM classes WHERE classes . enroll
> 50 (constrained with classes . enroll)
< 20 (constrained with classes . enroll)

These are of course totally arbitrary definition of what constitutes high and low enroll-
ment. The final SQL queries will be:

SELECT course_id, semester FROM classes WHERE classes . enroll > 50
SELECT course_id, semester FROM classes WHERE classes . enroll < 20

Here is the first rule:
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

trule_u(class, (  
enrollment :: (Qnoun, features:has_head:yes, index:12, mod:sign:index:1) &  
class :: (Qnoun, features:(has_modifier:yes, has_head:nil),  
        index:1, index2:12)  
< - >  
course_id :: (column, Qix(S)) &  
',.' :: (comma) &  
semester :: (column, Qix(S)) &  
classes :: (table, Qix(T), Qin_cond(nil)) &  
classes :: (table, Qix(D0), Qin_cond(yes)) &  
'.' :: Qix(B0,D0,E0) &  
enroll :: (column, Qix(E0)) &  
_ ::: (comparison_op, Qix(_B0)  
  \trans_selec(S,T)))

This trule consumes both words in the LHS, and produces most of the lexical units necessary for the complete query. Note that there are two column names being produced, and which will end up in the SELECT-FROM part of the query. They are the key attributes of the table "classes." Their index features share the same value. The last three lexical units produced by the trule make up the first part of the one condition in the query. These units are constrained with an unspecified comparison operator, which is produced by one of the other trules. Only one tmacro, transselect/2, is called, to produce the "SELECT" and "FROM" keywords.

Finally, I examine the trules for "high" and "low":

trule_u(high, (  
  high :: (features:pos:adj, index:1, mod:sign:index:1) &  
enrollment :: (Qnoun, index:1)  
< - >  
classes :: (table, Qix(D0), Qin_cond(yes)) &  
'.' :: Qix(B0,D0,E0) &  
enroll :: (column, Qix(E0)) &
'>' :: @ix(A0,B0,C0) &

'50' :: (constant, @ix(C0))
\trans_tables(D0,classes)
\trans_cond_and(A0)).

trule_u(low, {
    low :: (features:pos:adj, index:1, mod:sign:index:1) &
    enrollment :: (@noun, index:1)
    < - >
    classes :: (table, @ix(D0), @in_cond(yes)) &
    . ' :: @ix(B0,D0,E0) &
    enroll :: (column, @ix(E0)) &
    '<' :: @ix(A0,B0,C0) &
    '20' :: (constant, @ix(C0))
\trans_tables(D0,classes)
\trans_cond_and(A0)).

For these trules, the consumed word only needs to be constrained with its head, and no other lexical units in the SL bag. In both cases the comparison operator and value expression (here they are constants) are constrained in the RHS with the lexical units forming the table/column pair classes . enroll. It is in these trules that the macros trans_tables/2 and trans_cond_and/1 are called.

Let us now consider a more complicated example, involving lexicalized compounds: “computing science grad students’ grades”

This query is covered by two trules. the first one, consuming “grad,” “student” and “grade,” has already been presented in example example 3.3. The second consumes “computing” and “science” together as a lexicalized compound, constrained with “grad” and ”student.” Since this compound acts as a CV noun modifying a TN noun, it will translate as one or more conjoined conditions. These conditions are:

gr_students . stud_id = stud_dept . stud_id
AND stud_dept . dept_id = 'computing science'

And the resulting SQL query is:

```sql
SELECT grades.num FROM grades , gr_students , stud_dept
WHERE grades . stud_id = gr_students . stud_id
AND gr_students . stud_id = stud_dept . stud_id
AND stud_dept . dept_id = 'computing science'
```

Note how the two conditions introduced by the CV noun complete the join in the conditions part. The trule is:

```
trule_v(computer, ( 
  computing :: (features:pos:adj, index:l2, mod:sign:index:l2) &
  science :: (@noun, features:has_head:yes, index:l3, mod:sign:index:l) &
  grad :: (features:pos:adj, index:l, mod:sign:index:l) &
  student :: (@noun, features:has_modifier:yes, index:l, index2:l2)
  < - >
  gr_students :: (table, @ix(D0), @in_cond(yes)) &
  '.' :: @ix(B0,D0,E0) &
  stud_id :: (column, @ix(E0)) &
  '=' :: @ix(A0,B0,C0) &
  stud_dept :: (table, @ix(F0), @in_cond(yes)) &
  '.' :: @ix(C0,F0,G0) &
  stud_id :: (column, @ix(G0)) &
  'AND' :: @ix(And0,A1) &
  stud_dept :: (table, @ix(D1), @in_cond(yes)) &
  '.' :: @ix(B1,D1,E1) &
  dept_id :: (column, @ix(E1)) &
  '=' :: @ix(A1,B1,C1) &
  'computing science' :: @ix(C1,F1,G1) &
  \trans_tables(D1,stud_dept)
  \trans_tables(D0,gr_students)
  \trans_cond_and(A0)).
```
As can be seen, trules consuming and/or constraining with lexicalized compounds behave exactly the same way as trules without such compounds. The coindexing scheme is a simple, though crude, way to resolve syntactic ambiguity.

Function Nouns

Function nouns are all noncompositional, and therefore require coconsumption with some of their modifiers. They translate to the function noun and matching set of parentheses around the argument.

Example 3.7 "Linguistics grad students number"

We assume that this query is to ask about the number of students, not their identification numbers. If there were a query concerning an identification number, I would need additional lexical entries. What will happen in this case is that "number" (the function noun) must be consumed with its modifier and its modifier’s modifier, just as though those two last words were the heads of the query. "Linguistics" can be consumed separately, and will be added compositionally to the final SQL query. It only needs to be constrained with "grad" and "student," not "number."

The trules’ outputs are, respectively:

SELECT count ( stud_id ) FROM gr_students
WHERE gr_students . stud_id = stud_dept . stud_id
AND stud_dept . dept_id = linguistics

The final SQL query is:

SELECT count ( stud_id ) FROM gr_students , stud_dept
WHERE gr_students . stud_id = stud_dept . stud_id
AND stud_dept . dept_id = linguistics

The trules are:
The difference between this trule and others like it is the lexical units found in the SELECT-FROM part of the query. The index feature of all the lexical units in the function expression share the same value. This value is also the first argument for the trans.select/tmacro, which will generate the “SELECT” and “FROM” keywords just as well as if the lexical units were simply column names.

trule_u(linguistics, (  
  linguistics :: (noun, features:has_head:yes, index:12,  
    mod:sign:index:1) &  
  grad :: (features:pos:adj, index:1, mod:sign:index:1) &  
  student :: (noun, features:has_modifier:yes, index:1, index2:12)  
  < - >  
  gr_students :: (table, @ix(T), @in_cond(nil))  
  \\trans.select(S,T)).

`trule_u(number, (  
  grad :: (features:pos:adj, index:12, mod:sign:index:12) &  
  student :: (noun, features:has_head:yes, index:12, mod:sign:index:1) &  
  number :: (noun, features:has_modifier:yes, index:1, index2:12)  
  < - >  
  count :: (func_name, @ix(S)) &  
  '(' :: (open_parenth, @ix(S)) &  
  stud_id :: (column, @ix(S)) &  
  ')') :: (close_parenth, @ix(S)) &  
  gr_students :: (table, @ix(T), @in_cond(nil))  
  \\trans.select(S,T))).

```
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

\[
\text{stud\_dept} :: (\text{table, @ix(D1), @in\_cond(yes)}) \&
\text{'.'} :: @ix(B1,D1,E1) \&
\text{dept\_id} :: (\text{column, @ix(E1)}) \&
\text{'}=' :: @ix(A1,B1,C1) \&
\text{'linguistics'} :: @ix(C1,F1,G1)
\]
\[
\text{\textbackslash trans\_tables(D1,stud\_dept)}
\text{\textbackslash trans\_tables(D0,gr\_students)}
\text{\textbackslash trans\_cond\_and(A0))}.
\]

This is a good example of the rules' modularity: whether or not "grad student" modifies another expression is irrelevant. In particular, it does not matter whether or not an F name modifies the compound.

### 3.2.4 Conclusion

We have presented a sample solution for the translation of complex nominals containing lexicalized compounds and compositional expressions (whether ambiguous or not), demonstrating the feasibility of a lexicalist approach to this problem. Only a small amount of constraining is necessary to resolve lexical and syntactic ambiguity.

### 3.3 Problem Areas

#### 3.3.1 Overview

One problem not addressed by my basic solution is linguistic problem 2 (as listed on page 28), or noncompositional ambiguity. I must also address issues related to problem 1, that of noncompositional modifiers, which is also not specifically addressed. In this section I will propose a possible solution for these linguistic problems. First, I will examine the issue of noncompositional modifiers. Then I will discuss issues related to noncompositional ambiguity, for each word type.
3.3.2 Noncompositional Modifiers

We have seen examples of lexicalized compounds in the previous section: all component words must be coconsumed in all relevant rules, and the subcompound interacts compositionally with other compositional elements in the query. Another important issue is the ordering of rules in the bilex. Rules consuming compounds with the same heads must be arranged so that the more constrained rules come first, since the translation algorithm I am using processes rules in the order they occur. Though an unordered bilex will not cause errors, it will increase processing time due to backtracking. This solution will be the same for all words and expressions, regardless of type or role.

An example of such noncompositional compounds: "grades" will translate to \( \text{SELECT grade\_num FROM grades} \), while "official grades" will translate to \( \text{SELECT grade\_off FROM grades} \). The rules consuming "official grades" must be placed before those consuming "grades" in the bilex.

3.3.3 Ambiguous TN Nouns

Some ambiguous TN nouns cannot be translated compositionally with their heads or modifiers if those expressions also happen to be TN nouns. Consider the three simple queries: "linguistics students," "student grades" and "linguistic student grades." If "student" were unambiguous, we could reuse the rule that consumes "student" and "grade," as well as a rule consuming "linguistics" (constrained with "student") and the lexical units produced by both these rules would compositionally add up to the query we want. Unfortunately "student" is ambiguous; the noun refers to two tables, \( \text{und\_students} \) (for undergrad students) and \( \text{gr\_student} \) (for grad students). We shall see that such reuse is not possible for ambiguous nouns.

Let us examine the first two (two-word) queries, "linguistics students" and "student grade." Their respective SQL equivalents are:

\[
\begin{align*}
\text{SELECT} & \text{ stud\_id FROM und\_students, gr\_students , stud\_dept} \\
\text{WHERE} & \\
( & \text{und\_students . stud\_id = stud\_dept . stud\_id} \\
\text{OR} & \text{gr\_students . stud\_id = stud\_dept . stud\_id} ) \\
\text{AND} & \text{stud\_dept . dept\_id = linguistics}
\end{align*}
\]
SELECT grade_num FROM grades, und_students, gr_students
WHERE grades.stud_id = und_students.stud_id
OR grades.stud_id = gr_students.stud_id

Let us consider the rule associated with the first compound:

\[
\text{trule_u}(\text{student}, ( \\
\text{linguistics} :: (@\text{noun}, \text{features:has\_head:yes, index:}12, \text{mod:sign:}1) & \\
\text{student} :: (@\text{noun}, \text{features:has\_modifier:yes, index:}1, \text{index2:}12) \\
< \rightarrow \\
\text{stud\_id} :: (\text{column, @ix(S)}) & \\
\text{gr\_students} :: (\text{table, @ix(T), @in\_cond(nil)}) & \\
',' :: \text{comma} & \\
\text{und\_students} :: (\text{table, @ix(T), @in\_cond(nil)}) & \\
',' :: \text{comma} & \\
\text{stud\_dept} :: (\text{table, @ix(T), @in\_cond(nil)}) & \\
'(\' :: (\text{open\_parenth, @ix(A0,AD0)}) & \\
\text{und\_students} :: (\text{table, @ix(DD0), @in\_cond(}yes)) & \\
':' :: @ix(BD0,DD0,ED0) & \\
\text{stud\_id} :: (\text{column, @ix(ED0)}) & \\
'=' :: @ix(AD0,BD0,CD0) & \\
\text{stud\_dept} :: (\text{table, @ix(FD0), @in\_cond(}yes)) & \\
':' :: @ix(CD0,FD0,GD0) & \\
\text{stud\_id} :: (\text{column, @ix(GD0)}) & \\
'OR' :: @ix(Or0,AD0,AD1) & \\
\text{gr\_students} :: (\text{table, @ix(DD1), @in\_cond(}yes)) & \\
':' :: @ix(BD1,DD1,ED1) & \\
\text{stud\_id} :: (\text{column, @ix(ED1)}) & \\
'=' :: @ix(AD1,BD1,CD1) & \\
\text{stud\_dept} :: (\text{table, @ix(FD1), @in\_cond(}yes)) & \\
':' :: @ix(CD1,FD1,GD1) & \\
\text{stud\_id} :: (\text{column, @ix(GD1)}) & \\
'))' :: (\text{close\_parenth, @ix(A0,AD1)}) &
\)
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

'AND' :: @ix(And0,A1) &
stud.dept :: (table, @ix(D0), @in_cond(yes)) &
'.' :: @ix(B0,D0,E0) &
department :: (column, @ix(E0)) &
'=' :: @ix(A0,B0,C0) &
linguistics :: (constant, @ix(C0))
\trans_select(S,T)
\trans_cond,and(A0)).

The join conditions that link stud.dept with the two student tables are disjoined (ie: separated by an "OR" keyword) and the disjunction itself is put between parentheses. This is to avoid any possible ambiguity due to the presence of another connector ("AND"), and allows the disjunction to be treated as an individual condition (for example, it is coindexed with the "WHERE" keyword just like any condition). The index feature of the opening and closing parentheses share the same value, which is also the argument of the trans_cond_and tmacro. Note also the three non-null indices of the "OR" keyword, coindexing with its two adjacent conditions. Such coindexing is necessary to guarantee the correct ordering at the generation stage.

The size of this rule is unavoidable, since it produces the lexical units for three conditions. It also produces three tables that will be placed in the SELECT-FROM part of the query. They are in fact all the tables that will be used in the query. This renders unnecessary any trans_tables/2 calls. Only two tmacros are called: the usual trans_select/2 and trans_cond_and/1.

Now let us look at the rule for the second query:

true_u(grade, (  
    student :: (@noun, features:has_head:yes, index:12, mod:sign:index:1) &  
    grade :: (@noun, features:has_modifier:yes, index:1, index2:12)  
    < - >  
    grade_num :: (column, @ix(S)) &  
    grades :: (table, @ix(T), @in_cond(nil)) &
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

', ' :: comma &
und_students :: (table, @ix(T), @in_cond(nil)) &
', ' :: comma &
gr_students :: (table, @ix(T), @in_cond(nil)) &
und_students :: (table, @ix(D0), @in_cond(yes)) &
'. ' :: @ix(B0,D0,E0) &
stud.id :: (column, @ix(E0)) &
'= ' :: @ix(A0,B0,C0) &
grades :: (table, @ix(F0), @in_cond(yes)) &
'. ' :: @ix(C0,F0,G0) &
stud.id :: (column, @ix(G0)) &
'OR' :: @ix(OrO,A1) &
gr_students :: (table, @ix(D1), @in_cond(yes)) &
'. ' :: @ix(B1,D1,E1) &
stud.id :: (column, @ix(E1)) &
'= ' :: @ix(A1,B1,C1) &
grades :: (table, @ix(F1), @in_cond(yes)) &
'. ' :: @ix(C1,F1,G1) &
stud.id :: (column, @ix(G1)) &
\\transselect(S,T)
\\transcond_and(A0)).

This trule consumes "student" and "grade" together. It is similar to the previous one, except that the disjunction is not between parentheses. They are not necessary in this case, since there is only one connector.

Now, "linguistics students' grades" translates to

SELECT grade.num FROM grade, und_students, gr_students, stud_dept
WHERE
  ( und_students . stud.id = stud_dept . stud.id
AND und_students . stud.id = grades . stud.id )
OR ( gr_students . stud.id = stud_dept . stud.id
AND gr_students . stud.id = grades . stud.id )
)
AND stud_dept . dept_id = linguistics

Note that the conditions part of this query consists of two disjoined conjunctions of conditions, this disjunction being conjoined with one more condition. It should be clear that this query cannot be built compositionally from the information provided by “student grades” and “linguistics” when modifying “student.” (Though “linguistics” did not have its own trule, we can easily see that it translates to the conditions WHERE ( und_students . stud_id = stud_dept . stud_id OR gr_students . stud_id = stud_dept . stud_id ) AND stud_dept . dept_id = linguistics, since this is the conditions part of the SQL equivalent of “linguistics students”). Simply conjoining of the two queries would lead to the following (incorrect) query for “linguistics students’ grades”:

WHERE
( und_students . stud_id = stud_dept . stud_id
OR gr_students . stud_id = stud_dept . stud_id )
AND
( grades . stud_id = und_students . stud_id
OR grades . stud_id = gr_students . stud_id )
AND stud_dept . dept_id = linguistics

The solution is to consume all three words together in the same trule:
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

', ' :: comma

stud_dept :: (table, @ix(T), @in_cond(nil)) &

'(' :: (open_parenth, @ix(A0, Par0)) &

'( ' :: (open_parenth, @ix(Par0, A0)) &

und_students :: (table, @ix(D0), @in_cond(yes)) &

' . ' :: @ix(B0, D0, E0) &

stud_id :: (column, @ix(E0)) &

' = ' :: @ix(A0, B0, C0) &

stud_dept :: (table, @ix(F0), @in_cond(yes)) &

' . ' :: @ix(C0, F0, G0) &

stud_id :: (column, @ix(G0)) &

' AND ' :: @ix(And0, A0, A1) &

und_students :: (table, @ix(D1), @in_cond(yes)) &

' . ' :: @ix(B1, D1, E1) &

stud_id :: (column, @ix(E1)) &

' = ' :: @ix(A1, B1, C1) &

grades :: (table, @ix(F1), @in_cond(yes)) &

' . ' :: @ix(C1, F1, G1) &

') ' :: (close_parenth, @ix(Par0, A1)) &

' OR ' :: @ix(Or0, Par0, Par1) &

'( ' :: (open_parenth, @ix(Par1, A2)) &

gr_students :: (table, @ix(D2), @in_cond(yes)) &

' . ' :: @ix(B2, D2, E2) &

stud_id :: (column, @ix(E2)) &

' = ' :: @ix(A2, B2, C2) &

stud_dept :: (table, @ix(F2), @in_cond(yes)) &

' . ' :: @ix(C2, F2, G2) &

stud_id :: (column, @ix(G2)) &

' AND ' :: @ix(And1, A2, A3) &

gr_students :: (table, @ix(D3), @in_cond(yes)) &

' . ' :: @ix(B3, D3, E3) &

stud_id :: (column, @ix(E3)) &
Since "linguistics student grades" begins and ends with TN nouns that are not noncompositionally ambiguous, the subcompound will interact compositionally with other compositional expressions, just as though it were a lexicalized compound. In fact, it is sometimes unnecessary to constrain with the entire subcompound. To see how this is so, consider the following query:

"Linguistics department students grades"

This query consists of a compositionally ambiguous TN noun head, modified by a (non-compositionally) ambiguous TN noun, modified by a compositionally ambiguous TN noun, modified by a compositionally ambiguous CV noun. Since it is semantically similar to "linguistics students' grades," it must of course translate to the same SQL query:

```
SELECT grade_num
FROM grade, und_students, gr_students, stud_dept
WHERE
  ( ( und_students . stud_id = stud_dept . stud_id
      AND und_students . stud_id = grades . stud_id )
  OR ( gr_students . stud_id = stud_dept . stud_id
```
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

AND gr_students . stud_id = grades . stud_id ) )
AND stud_dept . dept_id = linguistics

We can translate this query with the help of two trules: one consuming “department,” “student” and “grade,” the other consuming “linguistics” constrained with “department.” The first trule will produce the lexical units necessary for the entire query except the last condition, while the second produces the units for the last condition. Note that the CV noun does not have to be constrained, or coconsumed, with the entire subcompound.

This solution may be generalized to the case of one noncompositionally ambiguous TN noun modifying another. These nouns, plus their modifiers and heads, must be coconsumed. To discuss the RHS of the trules, I will make one simplifying assumption. I will assume that any two groups of tables, each collectively referred to by a noun or expression, will be joined by a single relation table (assuming they are joined at all, of course). If, for example, “department” referred to two or more tables (say, arts_departments and not_arts_departments, there would still be one “stud_dept” table joining the two groups.

The SQL query fragment joining these two groups of tables would then look like this:

WHERE
( stud_dept . dept_id = arts_departments . dept_id
OR stud_dept . dept_id = not_arts_departments . dept_id)
AND
( gr_students . stud_id = stud_dept . stud_id
OR und_students . stud_id = stud_dept . stud_id )

In this fragment, each ambiguously-named table appears exactly once: there is one instance each of arts_departments, not_arts_departments, und_students and gr_students. This would not be true if there were not a single relation table.

3.3.4 Ambiguous CN Nouns

Acting as query heads: The ambiguity is not a problem, since the noun will simply translate to multiple column names in the SELECT-FROM part of the SQL query.

Example 3.8 “Address” as query head, modified by “undergrad student,” translates to
SELECT street_address, city, postal_code FROM und_students

Acting as query modifiers: ambiguity may come from several different sources, requiring different solutions in each case. I will begin by listing the problems, then I will present solutions.

1. The CN noun refers to different columns found in the same, unambiguously named table

Example 3.9 “address” modifying “grad student”

2. The CN noun refers to different columns found in different tables referred to by the same ambiguous noun

Example 3.10 “address” modifying “student” refers to three attributes found in the tables gr_students and und_students

3. The CN noun refers to single columns found in different unambiguously named tables

Example 3.11 “enrollment” refers to the attribute enroll, which belongs to the tables departments and classes (both tables referred to by unambiguous nouns).

4. The CN noun refers to single columns found in different, ambiguously-named tables.

Example 3.12 “city” refers to one attribute, city, found in the tables gr_students and und_students.

Here are the proposed solutions:

1. The noun is constrained with its head, but coconsumed with its modifiers. The information provided by the CN noun and its modifier(s) may be added compositionally.

Example 3.13 Consider the query “Burnaby address grad students” (an awkward phrase, but it will serve to illustrate my solution). Though “address” refers to three separate attributes, “Burnaby” only specifies one (namely, “city”). Thus we see that I cannot treat such noncompositionally ambiguous CN nouns like unambiguous or
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

compositionally ambiguous ones, translating them to partial conditions. If only some of those conditions are actually completed, then the final SQL query will be incomplete. Thus, in order to properly translate the CN noun, I must at least constrain it with its modifier(s). Since the two words are not compositional, it seems practical to coconsume them.

The trule necessary to translate “Burnaby address” (modifying “grad students”) is:

```plaintext
true_u(computer, (
     'Burnaby' :: (@noun, features:has_head=yes, index:13, mod:sign:index:12) &
     address :: (@noun, features:(has_head=yes, has_modifier=yes),
        index:12, index2:13, mod:sign:index:1)
     grad :: (features:pos:adj, index:1, mod:sign:index:1) &
     student :: (@noun, features:has_modifier=yes, index:1, index2:12)
     < - >
     gr_students :: (table, @ix(D0), @in_cond(yes)) &
     '.' :: @ix(B0,D0,E0) &
     city :: (column, @ix(E0)) &
     '=' :: @ix(A0,B0,C0) &
     'Burnaby' :: @ix(C0)
\trans_tables(D0,gr_students)
\trans_cond_and(A0))).
```

This trule would of course be used in conjunction with another trule consuming “grad students.” Note that a similar trule would be needed for any other expression that is modified by “Burnaby address” (such as “undergrad students”).

Note that the compound translates to a whole condition. It is semantically equivalent to “Burnaby” (which, if modifying “grad students,” would translate to the same whole condition).
2. It is clear that, as in the previous case, the noun must be coconsumed with its modifier(s). However, this compound will only need to be constrained, not coconsumed, with its TN noun head. To ensure that the lexical units are ordered properly at the generation stage, they will have to constrain with some of the lexical units produced by the trules consuming the head.

Example 3.14 Consider the query “Burnaby address students’ departments.” Its SQL equivalent is:

```sql
SELECT dept_id FROM departments, stud dept, gr students, und students
WHERE departments. dept.id = stud dept. dept_id AND
  ( gr students. stud_id = stud dept. stud_id
  AND gr students. city = 'Burnaby'
  OR ( und students. stud_id = stud dept. stud_id
  AND und students. city = 'Burnaby' )
)
```

Of these lexical units, “Burnaby address” modifying “students” contributes:

```sql
AND gr students. city = 'Burnaby'
AND und students. city = 'Burnaby'
```

ie: two conditions, each preceded by an “AND” keyword and followed by a closing parenthesis. Each of them is constrained with the comparison operator and second part of the condition preceding it. Here is the trule:

```plaintext
trule_u(address, (  
  Burnaby :: (@noun, features:has_head:yes, index:13, mod:sign:index:12)  
  address :: (@noun, features:(has_head:yes, has Modifier:yes),  
               index:12, index2:13, mod:sign:index:1) &  
  student :: (@noun, features:has Modifier:yes, index:1, index2:12)  
           < - >  
  ' = ' :: @ix(A0,..C0) &
```
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

gr_students :: (table, @ix(F0), @in_cond(yes)) &
'.' :: @ix(C0,F0,G0) &
stud_id :: (column, @ix(G0)) &
'AND' :: @ix(And0,A0,A1) &
gr_students :: (table, @ix(D1), @in_cond(yes)) &
'.' :: @ix(B1,D1,E1) &
stud_id :: (column, @ix(E1)) &
'=' :: @ix(A1,B1,C1) &
'Burnaby' :: @ix(C1) &
')' :: (close_parenth, @ix(...A1)) &
'=' :: @ix(A2,...,C2) &
und_students :: (table, @ix(F2), @in_cond(yes)) &
'.' :: @ix(C2,F2,G2) &
stud_id :: (column, @ix(G2)) &
'AND' :: @ix(And1,A2,A3) &
und_students :: (table, @ix(D3), @in_cond(yes)) &
'.' :: @ix(B3,D3,E3) &
stud_id :: (column, @ix(E3)) &
'=' :: @ix(A3,B3,C3) &
'Burnaby' :: @ix(C3)
')' :: (close_parenth, @ix(...A3)) &
\trans_tables(gr_students,F0)
\trans_tables(und_students,F2)).

Note that I am not constraining with the entire preceding conditions. I only want as much of the context as necessary for proper placement of the lexical units. Note also the three indices in the connectors: it is this that allows us to constrain with the other conditions.

The first rule, consuming "students" and "grades," is identical to that presented on page 53, except that it also produces a pair of parentheses, and two opening parentheses. These lexical units will be constrained with the closing parentheses produced by the second rule. For brevity's sake, we will not present this new rule.
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

3. This case has actually already been covered in the previous section. Consider the noun "enrollment." Previously, I had treated it as though it belonged to only one table. However, both "departments" and "classes" are tables that contain such an attribute.

Since both these tables are unambiguously named, simple right constraining of the CN noun with the TN noun will be enough to resolve the ambiguity. For a given CN noun, only one rule for each head will be needed. For example, "enrollment" modifying "class" will translate to

WHERE classes . enroll

and, modifying "departments."

WHERE departments . enroll

4. Just as in case 2 above, the CN noun information cannot be added compositionally. It will have to be coconsumed with its head noun, and any other nouns (ie: other noncompositionally ambiguous TN nouns) it is coconsumed with. However, it will not need to be constrained with its modifier. Whether or not that modifier is a CV noun or predicating adjective, it will need to be constrained with the CN noun and its head TN noun (but not with anything else).

The rule consuming the CN noun will produce the lexical units necessary for several partial conditions, one for each table referred to by the ambiguous TN noun. These partial conditions will consist of a table/column pair, just as in the case of unambiguous or compositionally ambiguous TN nouns.

Example 3.15 Consider the query "Burnaby city students" (another awkward phrase). It will be translated with two rules: one consuming "city" and "student," the other consuming "Burnaby" constrained with "city" and "students." The outputs of these rules are, respectively:

SELECT stud_id FROM gr_students , und_students WHERE ( gr_students . city OR und_students . city )
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

= 'Burnaby' = 'Burnaby'

The "OR" keyword given in the first example is coindexed with two unspecified comparison operators, which will each be coindexed with one of the table/column pairs produced by the second trule. These operators are there to reduce any possible backtracking, in case the modifier is missing.

In the second example, the two partial conditions will complete the two other partial conditions provided by the first trule. They will be constrained with the first halves of the partial conditions produced by the first trules.

Here are the trules:

```plaintext
trule_u(student, 
   city :: (@noun, features:has_head:yes, index:12, mod:sign:index:1) &
   student :: (@noun, features:has_modifier:yes, index:1, index2:12)
               < - >
   stud_id :: (column, @ix(S)) &
   gr_students :: (table, @ix(T), @in-cond(nil)) &
   ',' :: comma &
   und_students :: (table, @ix(T), @in-cond(nil)) &
   '(': (open_parenth, @ix(Par0,A0)) &
   gr_students :: (table, @ix(D0), @in-cond(yes)) &
   '.': @ix(B0,D0,E0) &
   city :: (column, @ix(E0)) &
   ' ' :: (comparison_op, @ix(A0,B0)) &
   'OR' :: @ix(Or0,A0,A1) &
   und_students :: (table, @ix(D1), @in-cond(yes)) &
   '.' :: @ix(B1,D1,E1) &
   city :: (column, @ix(E1)) &
   ' ' :: (comparison_op, @ix(A1,B1)) &
   ')': (close_parenth, @ix(Par0,A1))
```
3.3.5 Ambiguous CV Nouns and Predicating Adjectives

Ambiguity for CV nouns comes when they refer to different column values for an attribute (if it is unambiguously named, it does not matter what that attribute is). One way to translate them while preserving compositionality (ie: not requiring them to be coconsumed with their head if it is a CN noun) is to use the "IN" SQL keyword, which allows one to specify a discrete range of values for an attribute.

Example 3.16 “spring semester classes.” This query would be translated with the use of two trules: one consuming “semester” and “classes,” the other consuming “spring” constrained with “semester”. The outputs of these trules are:
SELECT course_id, semester FROM classes
WHERE classes . semester
    IN ( '99-1', '98-1', '97-1', '96-1', '95-1' )

We have already seen the first rule in example example 3.5. The second rule is:

trule_u(spring, ( 
    spring :: (noun, features:has_head:yes, index:12, mod:sign:index:1) &
    semester :: (noun, features:has_modifier:yes, index:1, index2:12)
    < - >
    classes :: (table, @ix(D0), @in_cond(yes)) &
    'IN' :: @ix(A0,B0,C0) &
    '(', :: (open_parenth, @ix(C0)) &
    '99-1' :: (constant, @ix(S0)) &
    ',' :: (comma) &
    '98-1' :: (constant, @ix(S0)) &
    ',' :: (comma) &
    '97-1' :: (constant, @ix(S0)) &
    ',' :: (comma) &
    '96-1' :: (constant, @ix(S0)) &
    ',' :: (comma) &
    '95-1' :: (constant, @ix(S0)) &
    ')') :: (close_parenth, @ix(C0))
\\trans_tables(D0, classes)
\\trans_cond_and(A0)).

The same principle could be applied when the CV noun is modifying a TN noun, though compositionality is easier to maintain in this case. One could simply translate the noun as a disjunction of conditions (or of conjunctions of conditions)
Example 3.17 The query "1999 classes" could be translated in at least two ways. There is no reason to prefer one over the other, except that one rule is somewhat easier to read. The first involves the "IN" construct:

```sql
SELECT course_id, semester FROM classes
WHERE classes . semester IN ( '99-1', '99-2', '99-3' )
```

The second involves disjoined conditions:

```sql
SELECT course_id, semester FROM classes
WHERE ( classes . semester = '99-1'
OR classes . semester = '99-2'
OR classes . semester = '99-3' )
```

Difficulties arise when the CV noun's head is also noncompositonally ambiguous. If that head is a TN noun, then both expressions must be coconsumed. If it is a CN noun, then the coconsumption/constraining scheme will be as laid out in the previous subsection.

The issues for predicating adjectives are essentially the same as those for CV nouns. The only difference is that, as has been noted previously, predicating adjectives have an additional level of ambiguity, and must be constrained with their head CN and TN nouns if these nouns are compositional (whether ambiguous or not).

3.3.6 Structural Ambiguity

Structural ambiguity will not be an issue at this point in the research. The syntactic structures being used (only complex nominals) are relatively simple, and it is clear how even the small amount of constraining on the LHS resolves, or at least reduces a great deal, this type of ambiguity as far as translation is concerned. Structural ambiguity is still present in the syntax, but only one analysis will successfully transfer. Thus, ambiguity is resolved.

3.4 Evaluation and Discussion

3.4.1 Bilex Complexity and the Database Model

This subsection examines how the database model contributes to bilex complexity. Let us first look at the origin of rules:
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

- One rule is needed for every join between two tables.
- One rule is needed for every attribute in every table
- One rule is needed for every possible value of every attribute (except possibly key attributes, which may not need to be specified for some tables)
- There will be more rules for predicing adjectives and ambiguously-named tables, columns and values, but the number is impossible to predict a priori.

It is clear that the greater number of the rules comes from the column values. Unfortunately, it is impossible to avoid specifying a rule (and an entry in the SQL lexicon) for every single column value. Note also that the bilex will roughly grow in proportion to the English lexicon.

3.4.2 Role of the Unilingual Grammars

This subsection examines how the unilingual grammars affect bilex complexity and processing time.

A greater coverage for the SL grammar will not by itself affect bilex complexity or the system's performance (except perhaps at the parsing stage). However, the use of macros complicates the issue. We have seen that unilingual macros do make rules easier to read, but they can also be used to simplify the bilex itself. Consider the \texttt{Qadj} macro (a full description is given in Appendix B), covering many different kinds of coindexing situations between the adjective and its head. Since it contains disjunctions, it could be used to combine rules whose LHS's, separately, are covered by it. However, this simplification is matched by a potential increase in processing time due to backtracking at the transfer stage.

A more general way to simplify the bilex is to incorporate semantic features in the English grammar. Since the rules' LHS's could refer to the semantics of a given word or compound rather than the orthography or surface structure, this would allow single rules to cover families of synonyms, as well as a larger number of syntactic relations (ie: not just nominal compounds, but equivalent expressions involving prepositional phrases).

The TL grammar could affect performance and bilex simplicity in the same way. However, SQL macros are very simple and do not contain disjunctions. Therefore, they can only affect bilex readability.
This is only one example of the criteria specified at the beginning of this chapter working against one another. Here, a decrease in bilex complexity implies an increase in backtracking. It also shows how all the modules in the system, not just the transfer module, interact to affect performance and efficiency.

### 3.4.3 Reuse of Trules

We have seen several times in this chapter that trules presented for one situation can be reused in another. This is a general comment that may be made for all trules consuming unambiguous or compositionally ambiguous expressions. However, the same cannot be said for noncompositionally ambiguous expressions. The lack of compositionality of such expressions means that trules consuming them must consume the context as well, which reduces reusability.

### 3.4.4 Constraints and Processing Time

Any constraining in the trules increases the risk of backtracking, since there is more chance of either the SL or the TL bag being inconsistent with the constraints. However, one advantage is that some errors are caught earlier. TL bags that cannot generate to syntactically correct queries may sometimes be detected at the transfer stage with RHS constraining, and thus never have to go through the generation stage.

In my solution I have implemented such RHS constraints. They do not affect bilex complexity, since the constrained TL lexical units are the equivalent of the constrained SL lexical units in each trule's LHS. However, they prevent incomplete TL bags from being produced, thus avoiding backtracking at the generation stage. All the backtracking takes place at the parsing or transfer stage.

### 3.4.5 Alternative Constraining Schemes

As mentioned previously, one role of LHS constraints is to ensure correctness by resolving lexical and structural ambiguity. What is the minimum amount of constraining necessary to achieve this goal? Consider different word types and their contexts. What kind of context is necessary for TN nouns? It is clear that constraining with the head is sufficient to resolve lexical ambiguity. Is it necessary? First, note that for each TN noun, the bilex must contain
a rule for each of its possible heads (whether or not the two are constrained in that rule), to cover all possible translations. But without some context, there is no way to guarantee the correct rule (and hence, the correct translation) will be chosen. This type of error will not even be detected at the generation stage, since the resulting query would be syntactically correct. Thus, we see that a minimum of constraining is absolutely necessary to guarantee correctness, without increasing biplex complexity.

Similar observations may be made about any expression that translates to whole conditions (some adjectives and CV nouns, modifying TN nouns). Without some kind of context to guide transfer, English queries involving these expressions risk translating to syntactically correct but semantically incorrect SQL queries. And, though CV nouns do not require an English context to transfer properly, their SQL equivalents require an SQL context to generate correctly. Therefore, we see that context is necessary in all situations.

A related question is whether or not it is necessary to constrain expressions with their heads as opposed to their modifiers. In principle, constraining with modifiers is feasible. It would resolve structural ambiguity for translation as well as head-constraining. However, to resolve lexical ambiguity, some very specific constraints are necessary, namely: CV nouns with their head CN nouns, and CN nouns with their head TN nouns. To have the opposite direction of constraining would either drastically increase processing time or produce semantically incorrect queries.

### 3.4.6 Constraints and Query Efficiency

With each expression being processed with such limited constraints, it is normal to expect some redundancy of conditions in the resulting SQL query. In fact we have already seen such duplication in example 3.4, where I translated “linguistics department professors’ classes enrollment.” A more extreme example involves the duplication of join conditions.

**Example 3.18** Consider the query “Popowich grad students’ grades” (where “Popowich” is the name of a professor). It is covered with two rules: one consuming “grad students’ grades” (which has already been presented in a previous example), the other consuming “Popowich” and constraining with “grad student.” The respective outputs of these rules are:
SELECT grades.num FROM grades, gr.students
WHERE grades.stud_id = gr.students.stud_id

and

gr.students.stud_id = grades.stud_id
AND grades.course_id = classes.course_id
AND grades.semester = classes.semester
AND classes.prof_id = 'Popovich'

The resulting query is:

SELECT grades.num FROM grades, gr.students
WHERE grades.stud_id = gr.students.stud_id
AND gr.students.stud_id = grades.stud_id
AND grades.course_id = classes.course_id
AND grades.semester = classes.semester
AND classes.prof_id = 'Popovich'

As can be seen, one entire condition is duplicated. The only possible way to reduce this problem would be to widen the context for each expression. However, it would not eliminate it entirely, and would have the unfortunate side effect of increasing the bilex complexity to unacceptable levels. It is not worth the effort, at this point in the research. Query efficiency is not the most important of the criteria for building the bilex.

### 3.4.7 Bilex Readability

This is an issue for the designer only, not the end-user. Even with such a small lexicon and database model, some of the trules for ambiguous expressions are very long and may be difficult to read. Though this is not an issue for the end-user, nor would it affect performance, it would be desirable to shorten the trules somehow. A possible solution would be to introduce more transfer macros to produce the lexical units for conditions from a single index value and a list of arguments representing the table names, column names and constants in the condition. For example, the trule consuming “99-1 classes” (presented in example 1) would be rewritten as:
CHAPTER 3. TRANSLATION OF NOMINAL COMPOUNDS

trule_u(class, (
  '99-1' :: (Qnoun, features:has_head:yes, index:l2, mod:sign:index:l) &
  class :: (Qnoun, features:has_modifier:yes, features:has_head:nil, 
             index:l, index2:l2)
          < - >
  course.id :: (column, @ix(S)) &
  ',' :: (comma) &
  semester :: (column, @ix(S)) &
  classes :: (table, @ix(T), @in_cond(nil))
\\trans_select(S,T)
\\condition(classes,semester,'=','99-1',A0)
\\trans_cond_and(A0))).

With this particular instance of condition/4 being coded as:

condition(classes,semester,'=','99-1',A) tmacro

  []
         < - >
  classes :: (table, @ix(D), @in_cond(yes)) &
  ',' :: @ix(B,D,E) &
  '=' :: @ix(A,B,C) &
  '99-1' :: (constant, @ix(C)).

This would necessitate coding tmacros for every single possible condition. A simpler solution would be to introduce tmacros for table/column pairs only. This would still simplify the bilex considerably, and have the advantage of being much easier to implement automatically (see next chapter).

3.5 Future Work

Future work on this approach will involve increasing the English coverage. The first step would be expanding the bilex to cover simple prepositional phrases by changing the trules'
LHS's. For example, consider the phrase "students in linguistics." The lexical units' descriptions in the bilingual lexicon (with this particular grammar) would be:

\[
\begin{align*}
\text{student} &:: (\text{noun}, \text{index}:1) \\
\text{in} &:: (\text{noun}\_\text{mod}\_\text{prep}, \text{index}:1, \text{index}1:\text{I2}) \\
\text{linguistics} &:: (\text{noun}, \text{index}:\text{I2})
\end{align*}
\]

These rules would be just as modular as the ones presented in my solution. The constraints would also serve to resolve structural ambiguity for translation, just like the rules for nominal compounds. To reduce bILex complexity, it would be desirable to develop a grammar that will handle both these kinds of syntactic structures (nominal compounds and prepositional phrases) more uniformly, or write more abstract rules that, say, recognize a modifier-head relation, regardless of the precise syntax involved.

Other simple but significant changes would be allowing for different sentence types, such as questions, and making the interface more user-friendly. For example, it would be easy to rename column in the SQL query so that the returned data is more meaningful to the naive user. (For an example of this, see [Ove99].)

### 3.6 Conclusion

In this chapter I have introduced a possible arrangement for a bilingual lexicon to translate queries consisting of complex nominals. The basic solution of relating each compositional expression to one rule (or a set of rules) is not only feasible, but has a number of advantages. Bilex complexity is kept low due to the potential of reuse of rules, and processing speed is relatively high, due to the constraining of expressions with others in each of the two language bags. Errors that would cause backtracking later on in the translation process are detected early, thus saving time. The SL bag constraints also have the dual purpose of resolving structural ambiguity by only allowing certain parses to transfer successfully, and resolving some lexical ambiguity by allowing the transfer module to examine each consumed expression's context. TL bag constraints serve to ensure correctness of the final SQL query by specifying the relative ordering of some SQL lexical units.

The problems of noncompositional expressions has also been addressed, as well as the
issue of noncompositional ambiguity. For these problems, the solution is less straightforward. Noncompositionality necessitates coconsumption of expressions, which implies higher complexity for the bilex, as well as for the individual rules.

One particular disadvantage of my approach is that the queries have a chance of being sub-optimal, in that some conditions may be duplicated. However, this does not affect correctness of the final query.
Chapter 4

Portability and Lexical Generation

4.1 Introduction

In the previous chapter we have seen the necessary characteristics of the bilingual lexicon, to achieve the best balance of complexity, processing speed and query simplicity. For compositional expressions, constraining with the head is enough to resolve structural and lexical ambiguity (in the case of predicating adjectives, sometimes the constraining must include the head's head as well). The words in noncompositional compounds (such as lexicalized compounds) will be consumed together, in all the rules in which they appear.

This chapter deals with the second aspect of my research, the greater part of which involves semi-automatically building such a bilingual lexicon. The basic principle is to use a bilingual text [Tur98]; more precisely, this bilingual text will contain pairs of short queries supplied by the developer (which I am calling “fragments”), one English and one SQL, to establish correspondences between English expressions and their SQL equivalents.

I will present an algorithm to automatically generate a set of rules out of these fragment pairs. For each pair, the output will consist of the rule(s) necessary to translate that English fragment, and also its component expressions. I will first discuss what kinds of English fragments are needed for this process. Then I will examine the SQL fragments and their semantic structure, and how it relates to that of the corresponding English fragment. Then I will examine the actual lexical generation process. I will discuss the parsing of the fragments, what rules are produced, and what the left-hand side and right-hand side will look like. This will be followed by a discussion on subsequent improvement of the rule set.
to obtain a more refined bilex. Limitations of this algorithm (such as English coverage) will be addressed. The issue of reuse of linguistic components will also be addressed, with a look at the English grammar and its limitations.

4.2 English Fragments

The fragments' goal is to establish the necessary syntactic/semantic relationships between individual English expressions (i.e., predating adjectives, nouns or lexicalized compounds). Note that the exact nature of these relationships is irrelevant to the present research. All that matters is that the relationship exists. The fragments collectively must include not only all of the English individual expressions that would be used in queries (for a given domain, of course) but also all of their interrelationships. However, there must be as little overlap as possible, to keep the number of fragments to a minimum and reduce the workload for the developer.

Examples of such fragments would be "large department" (establishing that the noun "department" can be modified by the adjective "large") and "linguistics grad student" (establishing that the noun "linguistics" can modify the expression "grad student").

4.2.1 Size of Fragments

In the previous chapter it has been made clear that, even for compositionally ambiguous words, trules require very little constraining and, except for lexicalized compounds, no coconsumption. It is possible, then, for each English fragment to contain only two or three expressions, whether individual words or lexicalized compounds. Each lexicalized compound counts as an expression, and its own structure and size do not influence those of the fragment it is a part of. Both fragments presented in the previous subsection, for example, would have a size of 2.

It is desirable for fragments to be as small as possible because this guarantees ease of parsing and analysis for both queries in each pair. SQL equivalents of larger fragments would be too difficult to analyze semantically, at least at this stage of the research.
4.2.2 Patterns and Structure

Recall that, in the previous chapter, I have laid out the possible contexts for each type of word. For example, only CN or TN nouns can be query heads. CV nouns and predicating adjectives can only be modifiers. Each word's context is related to the kinds of semantic relationships that word can be a part of, and it is these relationships that I will now examine.

2-Word Patterns

These will be used to establish the relationships between two tables, between each table and its attributes, and other ways in which tables and table attributes may be modified.

1. TN TN - These are used to establish relationships between two tables. Note that they are not necessarily adjacent tables. An example is “department professor,” which specifies that “professor” may be modified by “department.” (To specify the reverse, one would need a separate fragment pair). Though “department professor” is an incomplete English query (one would expect “department” to be itself modified), it does translate to a syntactically correct SQL query.

2. TN CN - These establish relationships between tables and their attributes. Examples of such fragments would be “grad student address” and “department enrollment.” Note that in the first example, “grad student” is a lexicalized compound and is treated as a TN noun. Note also that the second example could be considered another incomplete query (“department” should perhaps be modified, although the query could also make sense if it weren't).

3. CV CN - These patterns establish relationships between attributes and values of other attributes in the same table. An example is “linguistics enrollment” (“linguistics” is a key attribute value of “departments,” of which “enrollment” is an attribute). Fragments of this type do not establish relationships between attributes and their values. Such two-word fragments would translate to incomplete SQL queries and will not be considered for this research.

4. CV TN - These establish relationships between tables and column values in either the same, or different, tables. The column value in question is often (but not always) a key attribute value. Examples of this would be “linguistics students” and “linguistics
department.” Though the latter is not a very “interesting” query, it does translate to a syntactically correct SQL query, and provides important semantic information. One can see this pattern (and the next one also) as complementary to the first two.

5. **PA TN** – These establish relationships between tables and their more general modifiers. Examples would be “good students” and “large departments.” Note that the modifier would determine more than one attribute, in the same table or another. (For example, “good” probably refers to each student’s grades.)

**3-Word Patterns**

The only situation in which one needs more than simple constraining with the head is with CV nouns or predicating adjectives modifying CN nouns that are modifying TN nouns. These fragments will specify the relationships between tables, attributes and values.

Here are the patterns:

1. **CV CN TN** – Though in this case it may not be necessary to constrain with the TN noun (if the CN noun is unambiguous), the need sometimes arises, and having a uniform way of dealing with this particular is simpler. Besides, the SQL query must be syntactically correct. An example is “97-1 semester courses.”

2. **PA CN TN** – Constraining is necessary for the reasons stated above but is even more necessary. As mentioned in the previous chapter, predicating adjectives may have ambiguous meanings, resolved by the relevant table name. I will also assume for now that each predicating adjective corresponds to exactly one condition. An example of this type of fragment is “High enrollment classes.”

**Larger Patterns for Ambiguous Expressions**

As was seen in the previous chapter, noncompositional ambiguity resolution entails a large amount of coconsumption. These fragments might not follow any set pattern; therefore, I will not spend any time on this topic. The research will focus exclusively on compositional expressions and their related rules.
4.3 SQL Queries and Distribution of Information

This section concerns the semantic structure of SQL fragments. Their structure will in fact be extremely simple and easy to study and exploit. The information is contained in three parts; each is not necessarily present in any given query.

1. the SELECT-FROM part: the “head” of the query.
2. the various join conditions
3. the condition (if any) that contains non-join info (ie: that contains a column value of some kind). I will assume that there is only one such condition in each fragment, as that will make analysis much easier.

In all cases, as has been mentioned before, the SELECT-FROM part will correspond to the query head.

By elimination, the modifier in 2-word fragments must necessarily correspond to the conditions(s), if any. I am able to reason this way since the fragments’ component expressions are assumed to be compositional.

The RHS’s of 2-word fragments usually contain only one condition (since the modifiers specify the values of one attribute in a single table). The second word (ie: the head’s modifier) corresponds to the first table/column of this condition. The first word (ie: the CV noun or predicating adjective) corresponds to the second part of the condition. Again, I am able to “split” the query in this way because the English expressions are compositional.

4.4 Algorithm for Lexical Generation

This section will present the algorithm for producing a set of rules from the “bilingual text” of English/SQL fragment pairs as specified above. It is the first step towards producing a bilex that meets all the requirements laid out in the previous chapter. Let us first introduce a high-level description of the algorithm, after which I will examine the steps in more detail.

1. Preprocessing
2. Analysis
   2.1 Analyze English fragment
CHAPTER 4. PORTABILITY AND LEXICAL GENERATION

2.2 Analyze SQL fragment

3. Generation
   For each trule to be produced:
   3.1 Generate keyword
   3.2 Generate LHS
   3.3 Generate RHS

4.4.1 Preprocessing
This step involves the tagging of expressions according to their type (ie: TN, CN, CV, PA or F) and bracketing off lexicalized compounds to prevent difficulties in the parsing stage, such as structural ambiguity and lexical ambiguity (the output, in some cases, depends on the noun types). It must also involve removing all morphological variations. For now, this stage is done manually.

4.4.2 Analyzing the English Fragment
This stage provides the following information:

- the keyword for each trule to be produced. In the system I am using, trules need to be keyed by a word in the SL lexicon. See below for more details.
- the number of trules to be produced.

The number of trules produced depends on the fragment patterns. Fragment pairs whose English half’s head is a CN noun/expression will only produce one trule, since the CN noun cannot modify another expression in this context. Fragment pairs whose English half’s head is a TN noun/expression will produce two trules: one in which the fragment head is also a query head (consuming both expressions) and one in which it is a query modifier (consuming the fragment modifier and constraining with its head).

Fragment pairs whose English half consists of three words will produce three trules: the first one, in which the fragment head is also a query head (consuming the head and its modifier); the second one, in which the fragment head is a query modifier (consuming the CN noun and constraining with the fragment head); the third, applying to either
case, consuming the first word and constraining with the CN noun. If it is an adjective, it also constrains with the fragment head.

In all cases, the trule's keyword is the head of the compound that is being consumed (there are, of course, other possibilities).

4.4.3 Analyzing the SQL Fragment

The previous step specifies how the SQL fragment should be "split": either no splitting, splitting in two (SELECT-FROM part and conditions part) or splitting in three (SELECT-FROM part, first half of the condition, second half of the condition). At this stage, a list of column and table names in the SELECT-FROM part of the fragment is assembled. With each table name is associated an index, based on that table's first occurrence in the conditions part of the query.

4.4.4 Generating the LHS

The description of each entry in the LHS will depend entirely on its type (noun or adjective) and its position in the English fragment (determining the index, has_head and has_modifier features).

The head's index feature value is labeled "1": its modifier's index value will be labeled "12," and so on. The necessary structure sharing will also determine the values of the index2 and mod:sign:index features, where applicable.

This is a crude but effective way to coindex the SL bag items. A more realistic parse for lexicalized compounds could easily be found, by first indexing the heads of the compounds, but this is not essential to my research at this time.

4.4.5 Generating the RHS for Query Heads

The RHS in this case contains the lexical units for the SELECT-FROM part of the query, and possibly also lexical units for conditions (or partial conditions). The latter will be discussed in the next subsection.

The lexical units' description will depend entirely on their type, whether column or table. As mentioned previously, all columns share the same index value, which I will label
“S,” and all tables share another index value, which I will label “T.” Table descriptions must also contain the table name.

The rule ends with a single tmacro call, trans.select/2, taking as arguments “S” and “T.” There might also be a trans.cond.and/1 call if the RHS contains lexical units for conditions.

4.4.6 Generating the RHS for Modifiers

These RHS’s will only consist of lexical units for conditions or partial conditions. The main difficulty will be to set up the coindexing so that they generate correctly. Each sign’s index value will be represented by a letter depending on its position within each condition, followed by a number depending on the condition’s position in the RHS: the comparison operator’s index is represented by a variable of the form “An” (where n is a number). The index of the first and second part of the condition are, respectively, “Bn” and “Cn.” If the first part is a table/column pair, then the index of the table and the column are, respectively, “Dn” and “En.” For the second part, they are “Fn” and “Gn.” n is 0 for lexical units contained in the first condition of the RHS, 1 for the second, and so on. For example, the comparison operator for the first condition in the query has its index value represented by “A0.”

The rule will conclude with trans.tables/2 tmacro calls, one for each distinct table name. There will also be a call to the trans.cond.and/1 tmacro, taking as argument the index of the first condition (which will invariably be “A0”). In the case of partial conditions, all of these tmacros will only appear in the rule generating the signs for the second part of the conditions.

4.5 Generation Examples

We will first consider a pair whose English half consists of two TN nouns:

English: “Department professor”
SQL: SELECT prof_id FROM professors , departments
WHERE professors . dept_id = departments . dept_id

As mentioned previously, the role of such a semantically incomplete query is to provide relationships between tables.
Step 1: preprocessing: The English nouns are tagged: \texttt{department/TN \ professor/TN}

Step 2.1: analysis of the English fragment: Since the English fragment ends with a TN noun, the pair will produce two trules: one consuming both nouns and producing the lexical units for the entire SQL fragment, the other consuming “department” constrained with “professor” and producing the lexical units for the condition. These trules’ keywords are, respectively, “professor” and “department” (since the second trule only consumes “department”).

Step 2.2: analysis of the SQL fragment: The following information is extracted:

- columns list: \texttt{(prof.id)}
- tables list: \texttt{>((professors, D0), (departments, F0))}

since “professors” appeared once in the first half of the condition. “departments” appeared in the second half.

Step 3: Generation I will look at the generation steps for both trules in parallel, since there are so many similarities. I will call the trule consuming both nouns “trule 1,” and the other “trule 2.”

Step 3.1: generating the LHS The description for “department” will be the same for both trules:

\begin{verbatim}
(@noun, features:has_head=yes, index:12, mod:sign:index:1)
\end{verbatim}

Note that the noun’s \texttt{index} is represented by “12” because it is the next-to-last word in the compound.

As for “professor,” its \texttt{features:has\_head} feature must be set to “nil” in trule 2, and left unspecified in trule 2. Its \texttt{index} is represented by the variable “1,” since it is the head of the compound. The trules’ LHS’s will then be, in order:

\begin{verbatim}
department :: (@noun, features:has\_head=yes, index:12, 
               mod:sign:index:1) &
professor :: (@noun, features:(has\_modifier=yes, has\_head:nil),
\end{verbatim}
Step 3.2: generating the RHS  We have the list of columns and tables in the SELECT-FROM part of the query. The part of true 1's RHS that will generate it is:

```plaintext
prof_id :: (column, @ix(S)) &
department :: (table, @ix(T), @inx_cond(nil)) &
':,' :: (comma) &
professor :: (table, @ix(T), @inx_cond(nil))
```

The RHS's of both trules will contain the lexical units for the single condition, consisting of two table/column pairs joined by a comparison operator:

```plaintext
departments :: (table, @ix(D0), @inx_cond(yes)) &
':.' :: @ix(B0,D0,E0) &
dep_id :: (column, @ix(E0)) &
'=' :: @ix(A0,B0,C0) &
professors :: (table, @ix(F0), @inx_cond(yes)) &
':.' :: @ix(C0,F0,G0) &
dep_id :: (column, @ix(G0))
```

Trule 1 ends with two tmacro calls: trans_select(S,T) (since it consumes a query head) and trans_cond_and(A0) (since it contains a condition). Trule 2 ends with three tmacro calls: trans_cond_and(A0) (since it introduces at least one complete condition), as well as trans_tables(D0,departments) and trans_tables(F0,professors) (the two table names contained in the condition). The complete trules are, in order:
CHAPTER 4. PORTABILITY AND LEXICAL GENERATION

trule_u(professor, (  
department :: (@noun, features:has_head:yes, index:12,  
    mod:sign:index:1) &  
professor :: (@noun, features:(has_modifier:yes, has_head:nil),  
    index:1, index2:12)  
< - >  
prof_id :: (column, @ix(S)) &  
department :: (table, @ix(T), @in_cond(nil)) &  
',' :: (comma) &  
professor :: (table, @ix(T), @in_cond(nil)) &  
departments :: (table, @ix(D0), @in_cond(yes)) &  
'. ' :: @ix(B0,D0,E0) &  
dep_id :: (column, @ix(E0)) &  
'= ' :: @ix(A0,B0,C0) &  
professors :: (table, @ix(E0), @in_cond(yes)) &  
'. ' :: @ix(C0,E0,F0) &  
dep_id :: (column, @ix(F0))  
\text{\textbackslash trans.select}(S,T)  
\text{\textbackslash trans.cond.and}(A0)).

trule_u(department, (  
department :: (@noun, features:has_head:yes, index:12,  
    mod:sign:index:1) &  
professor :: (@noun, features:has_modifier:yes, index:1, index2:12)  
< - >  
departments :: (table, @ix(D0), @in_cond(yes)) &  
'. ' :: @ix(B0,D0,E0) &  
dep_id :: (column, @ix(E0)) &  
'= ' :: @ix(A0,B0,C0) &  
professors :: (table, @ix(F0), @in_cond(yes)) &  
'. ' :: @ix(C0,F0,G0) &  
dep_id :: (column, @ix(G0))  
\text{\textbackslash trans.tables}(D0,departments)
CHAPTER 4. PORTABILITY AND LEXICAL GENERATION

Now I consider an example involving a lexicalized compound and a noncompositionally ambiguous noun. As we shall see, the algorithm can handle some forms of noncompositional ambiguity.

**English:** "Grad students' addresses"

**SQL:**

```
SELECT street_address, city, province FROM gr_students
```

**Step 1: Preprocessing:** The lexicalized compound is bracketed and tagged, as is the noun: 

```
[grad/A student/N]/TN address/CN
```

**Step 2.1: analysis of the English fragment:** Since the English fragment ends with a CN noun, the pair will only produce one trule, consuming both expressions. This trule’s keyword will be "address," since that is the head of the consumed compound.

**Step 2.2: analysis of the SQL fragment:** The following information is extracted:

- columns list: (street_address, city, province)
- tables list: (gr_students)

Since there are no conditions, the table cannot be associated with an index.

**Step 3.1: generating the LHS:** "address," is the head noun. Its features:has_head feature is set to "nil," and its features:has_modifier feature is set to "yes." Similarly, the features:has_head feature of "student" is set to "yes," but its features:has_modifier is not specified, since its modifier is an adjective. The trule’s LHS will be:

```
grad :: (features:pos:adj, mod:sign:index:12, index:12) &
student :: (@noun, features:has_head:yes, index:12,
            mod:sign:index:1) &
address :: (@noun, features:(has_modifier:yes, has_head:nil), index:1)
```
Step 3.2: generating the RHS  We have the list of tables and columns, and nothing else is necessary to generate the RHS:

```
street_address :: (column, @ix(S)) &
'.' :: (comma) &
city :: (column, @ix(S)) &
',' :: (comma) &
province :: (column, @ix(S)) &
gr_students :: (table, @ix(T), @in-cond(nil))
```

The trule ends with a single tmacro call, to \texttt{trans\_select/2}. Here is the complete trule:

```
trule_u(address, (  
  grad :: (features:pos:adj, mod:sign:index:l2, index:l2) &  
  student :: (@noun, features:has\_head:yes, index:l2, mod:sign:index:l) &  
  address :: (@noun, features:(has\_modifier:yes, has\_head:nil), index:l)  
  < - >  
  street_address :: (column, @ix(S)) &  
  ',' :: (comma) &  
  city :: (column, @ix(S)) &  
  ',', :: (comma) &  
  province :: (column, @ix(S)) &  
  gr\_students :: (table, @ix(T), @in-cond(nil))  
\transdect(S,T))).
```

Finally, let us consider an example involving a three-word fragment:

**English:** “99-1 semester classes”

**SQL:**  
```
SELECT course\_id, semester FROM classes  
WHERE classes . semester = '99-1'
```

**Step 1: preprocessing:** The English words are tagged: 99-1/CV semester/CN class/TW
Step 2.1: analysis of the English fragment: This fragment pair will produce three trules: one consuming “semester” and “class” (let us call it trule 1), one consuming “semester” constrained with “class” (trule 2), the third consuming “99-1” constrained with “semester” and “class” (trule 3). Their respective keywords will be “class,” “semester” and “99-1,” since these words are the heads of the compounds consumed by the trules (actually, trules 2 and 3 only consume one word).

Step 2.2: analysis of the SQL fragment: The following information is extracted:

- columns list: (course.id, semester)
- tables list: ((classes, D0))

Step 3.1: generating the LHS: I have already shown in the first example what the structure sharing must be for trules 1 and 2. Their LHS’s are, respectively:

```
semester :: (@noun, features:has_head:yes, index:1, index2:1) &
class :: (@noun, features:(has_modifier:yes, has_head:nil),
               index:1, index2:1)
```

```
semester :: (@noun, features:has_head:yes, index:12, index2:1) &
class :: (@noun, features:has_modifier:yes, index:1, index2:1)
```

Trule 3 contains three nouns in its LHS. Their `index` feature values will be represented by “1,” “12” and “13”:

```
'99-1' :: (@noun, features:has_head:yes, index:13, mod:sign:index:12) &
semester :: (@noun, features:(has_modifier:yes, has_head:yes),
               index:12, index2:13, mod:sign:index:1)
class :: (@noun, features:has_modifier:yes, index:1, index2:12)
```
Step 3.2: generating the RHS: Trule 1 will contain the lexical units for the SELECT-FROM part of the query:

\[
\begin{align*}
\text{course.id} &:: (\text{column, @ix(S)}) \& \\
',' &:: (\text{comma}) \& \\
\text{semester} &:: (\text{column, @ix(S)}) \& \\
\text{classes} &:: (\text{table, @ix(T), @in\_cond(nil)})
\end{align*}
\]

The three trules will contain different parts of the condition. The lexical units and their descriptions are:

\[
\begin{align*}
\text{classes} &:: (\text{table, @ix(D0), @in\_cond(yes)}) \& \\
',' &:: @ix(B0,D0,E0) \& \\
\text{semester} &:: (\text{column, @ix(E0)}) \& \\
'=\cdot' &:: (\text{comparison\_op, @ix(A0,B0,C0)}) \& \\
'99-1' &:: (\text{constant, @ix(C0)})
\end{align*}
\]

In trules 1 and 2, the RHS will contain only the first table/column pair, constrained with an unspecified comparison operator. In trule 3, the RHS will contain the constant and the comparison operator, constrained with the table/column pair.

Trule 1 will end with a \texttt{trans\_select/2} macro call; trule 2 will contain no \texttt{tmacro} calls, and trule 3 will contain 2: one for \texttt{trans\_cond\_and/1} and one for \texttt{trans\_tables/2}.

Here are the complete trules, in order:

\[
\text{trule\_u(class, (}
\begin{align*}
\text{semester} &:: (@\text{noun, features:}\text{has\_head:yes, index:}12, \text{mod:}\text{sign:}1) \& \\
\text{class} &:: (@\text{noun, features:}(\text{has\_modifier:yes, has\_head:nil}), \\
 &\quad \text{index:}1, \text{index2:}12) \\
< &\rightarrow \\
\text{course.id} &:: (\text{column, @ix(S)}) \&
\end{align*}
\]
\]
'' :: comma &
semester :: (column, @ix(S)) &
classes :: (table, @ix(T), @in_cond(nil)) &
classes :: (table, @ix(D0), @in_cond(yes)) &
'' :: @ix(B0,D0,E0) &
semester :: (column, @ix(E0)) &
_ :: (comparison_op, index0:B0)
\trans_select(S,T))).

trule_u(class, (semester :: (@noun, features:has_head:yes, index:12, mod:sign:index:l) &
class :: (@noun, features:has_modifier:yes, index:1, index2:12))
  < - >
classes :: (table, @ix(D0), @in_cond(yes)) &
'' :: @ix(B0,D0,E0) &
semester :: (column, @ix(E0)) &
_ :: (comparison_op, index0:B0))).

trule_u('99-1', (99-1 :: (@noun, features:has_head:yes, index:13, mod:sign:index:l2) &
semester :: (@noun, features:(has_modifier:yes, has_head:yes),
  index:12, index2:13, mod:sign:index:l) &
class :: (@noun, features:has_modifier:yes, index:1, index2:12))
  < - >
classes :: (table, @ix(D0), @in_cond(yes)) &
'' :: @ix(B0,D0,E0) &
semester :: (column, @ix(E0)) &
'=' :: @ix(A0,B0,C0) &
'99-1' :: (constant, @ix(C0))
\trans_tables(D0,classes)
4.6 After Generation

After the trules have been generated, there are a couple of operations that may be applied to them: simplification and reordering.

In principle, as long as we have chosen our fragment pairs carefully, there should be no redundancy in the trule set, with one possible exception: three-word fragments that only differ by their first word (i.e., adjective or CV noun) will produce the same trules consuming or constraining with the last two words. This duplication can be avoided, however, by the algorithm keeping track of trules created by such fragments (say, with their LHS’s). The only simplification that can be done, then, is on individual trules. In almost all cases the LHS and RHS constraints cannot be reduced further, since they are necessary to resolve lexical ambiguity and ensure correctness of the final query. Should the expressions in question happen to be totally unambiguous, then it would be possible to remove the constraints. However, even in those cases, the extra constraints will not adversely affect bilex complexity or processing time. Therefore this simplification need not be performed, since all it would do is increase the developer’s workload.

It will also be necessary to reorder trules that consume lexicalized compounds with the same head so that those consuming longer compounds precede those consuming shorter compounds. This reordering is necessary, so that the more constrained trules are tried first at the transfer stage, thus avoiding unnecessary backtracking. Recall that trules are keyed by the heads of consumed compounds, which makes it very easy to keep track of them.

4.7 Evaluation and Discussion

The process I have presented is relatively simple and straightforward. It is very easy to implement, and the trules it produces conform to the solution laid out in the previous chapter.

One of its limitations is that it cannot handle most kinds of ambiguous expressions.
correctly. Ambiguous CV nouns are relatively easy to analyze, since the "IN" construct is self-contained and behaves in most ways like a comparison operator followed by a constant.

Other kinds of ambiguous expressions will not be handled correctly, or at least the resulting trules will not conform to the solution laid out in the previous chapter. Consider the case of ambiguous CN nouns that refer to multiple attributes in unambiguously-named tables. Processing them in three-word fragments according to the algorithm presented above would result in three trules being produced, as opposed to the two that we would expect (where the CN noun and its modifier are coconsumed). Note that a query involving such a fragment would still translate correctly with the trules provided by my process. However, they would not reflect the noncompositional nature of the CN noun.

In any case, for ambiguous expressions, the large number of trules due to noncompositional implies more work for the developer. A similar observation may be made for trules consuming CV nouns, even though the production of the relevant fragment pairs would in most cases be easy to automate.

4.8 Other Components

Besides the bilingual lexicon, two other components will need to be changed when porting to a new system: the transfer macros (which may or may not be considered a separate component, depending on how the particular system is organized) and the SQL lexicon. The trans.select/2 and trans.cond.and/1 tmacros are domain-independent, and all that is needed for the trans.tables/2 tmacros is a list of the table names (see Appendix C). The lexicon must contain not only the keywords, but also every table, column and column value in the domain. This information may be extracted from bilingual pairs at the same time as the trules.

Note that the SQL grammar is domain-independent. It will not need to be changed when porting to new domains.

4.9 Alternative Approaches

Let us consider the question of whether or not the fragment pairs must correspond to complete queries. It would not be possible to simplify the fragment pairs further, since the
English fragments represent the minimum context necessary to resolve lexical ambiguity; also, it would be efficient to extract as much information as possible from these fragments (in both languages). The only possible exception would be to "split" three-word fragments into two: one consisting only of the CN and TN nouns, and producing two trules whose RHS's consist of partial conditions, and one consisting of the original three-word fragment, but only producing the trule consuming the first word. This approach would have the disadvantage of increasing the size of the fragment pair list. On the other hand, there would be no need to keep track of which trules are created.

We can also consider the issue of using larger fragments in pairs. Though it might save the developers some time and effort, the problems of analysis render this option undesirable. The syntax of SQL queries is quite different from that of natural-language queries. Matching lexical units on both sides becomes too difficult if the order of SQL lexical units is more or less arbitrary (as is the case for the conditions part).

Another concern is that, even if these problems were solved, there might be redundancy in the trule set, which will require more overhead to remove.

Finally, let us briefly consider a different approach that might be taken to lexical generation. Instead of using bilingual pairs in which the database model is implicit, it might be possible to extract information directly from the model. The presence of catalog tables (or some other formal specification of the database schema) would make this process easier, in that it would at least provides with relations between tables and attributes, and thus with the RHS's for some of the trules. A developer would still have to associate with each table and attribute a set of nouns or lexicalized compounds, for trules' LHS's. It would be possible to automatically generate trules involving CV nouns, as long as we know each table's extension. Ambiguous expressions and predicating adjectives will probably require as much work as before. Solving this problem is beyond the scope of the current thesis, but would be an interesting direction for future research.

4.10 Conclusion

In this chapter, I have introduced an algorithm to partially automate the process of porting my approach to a new knowledge domain. It uses bilingual (English/SQL) pairs, from which
semantic and other information is extracted to produce bilingual entries. The size of these pairs is kept as small as possible, to simplify the problem of analysis, and because small pairs are sufficient to extract the necessary context information for transfer. The generation process is relatively straightforward, and the entire process is simple to implement.

It does have some limitations, however. Many kinds of ambiguous expressions would be difficult to analyze and create transfer rules for. Furthermore, the problem of structural ambiguity resolution for English fragments is still unresolved.
Chapter 5

Conclusion

I have presented a lexicalist approach to database front-ends, based on the Shake-and-Bake paradigm. The examples I use to illustrate it involve a sample database model and a restricted subset of English (namely, complex nominals).

The first part of my research focussed on the design of the bilingual lexicon. The transfer rules of the bilingual lexicon incorporate constraints on both SL and TL bags. These constraints can resolve certain forms of lexical ambiguity by allowing transfer rules to take consumed expressions' contexts into account, and implicitly resolve some structural ambiguity as well. Constraints also serve to reduce processing time, since errors that would cause backtracking in later stages of the translation process are detected early.

The constraints contained in the trules are minimal, which means that the trules have the potential to be reused in many different situations, at least for compositional expressions. However, it also means that the final SQL queries may not be optimal, and may contain duplicate conditions. The complexity of the bilingual lexicon and processing time depend to a degree on the unilingual grammars involved. In many respects these two criteria are mutually incompatible: low complexity tends to imply a higher processing time, while low processing time usually necessitates a high complexity for the bilingual lexicon.

Other disadvantages of my solution include an inability to handle noncompositional expressions efficiently. Since lexicalized compounds and other forms of noncompositionality are common in English, and since some forms of lexical ambiguity imply noncompositionality,
this means a corresponding increase in bilingual lexicon complexity. As well, the individual rules themselves will often be more complicated.

I have also addressed the issue of domain portability, by proposing an algorithm to semi-automatically generate a new bilingual lexicon, SQL lexicon and set of transfer macros. These are the only components that need to be changed when porting to new domains, since the SQL grammar and signature are domain-independent. The algorithm uses bilingual pairs of queries to extract the necessary information to generate the new components. Some manual preprocessing of the pairs is necessary, as well as postprocessing of the resulting rule set to conform to the solution I have laid out.

The main advantage of this algorithm is that it is very easy to implement. The disadvantage is that it only covers compositional expressions and lexicalized compounds: rules consuming ambiguous noncompositional expressions will not be generated by this algorithm.
Appendix A

Selected English Grammar Rules and Macros

In this appendix I will provide a full specification of the @adj and @noun transfer macros, as well as the right_mod grammar rule used to analyze complex nominals whose heads are to the right of their modifiers (as is the case with my examples).

adj macro
    (aroles_sign,
     'Macro':adjective,
     features: (pos:adj, cplform:null_form),
     ((mod:dir:left,
       ((mod:sign:features:wtype:(@bv(pron)), compl:status:)
       ;(mod:sign:features:wtype:(@bv(countness)),
          compl:status:yes))
    )
     :(mod:dir:right,
       mod:sign:features:(cplform:really_null_form,
         wtype:(@bv(countness))))),
    mod:(argument_sign,
         status:yes,
         sign:(features:(pos:noun),
         @}}
index:1,
comp0:(sign:features:pos:det),
comp1:(status:nil)),
comp0:status:nil,
comp1:(status:nil
);
(status:yes,
sign:( (features:(pos:verbs,
   inv:nil))

; (features:((pos:noun,
   cplform:nonnull_form,
   wtype:(@bv(case))))).

@nogap)
),
@saturated,
index:1)).
comp2:status:nil,
@nogap, @nofiller.
index:1,
index0:null_index, index1:(1,nonnullindex), index2:null_index).

noun macro
(@noun_gen,
features:has_head:HEAD,
index:LINK,
mod:(argument_sign,
dir:right,
status:HEAD,
sign:(sign,
   features:(pos:noun,
     cplform:really_null_form,
     wtype:(@bv(countness_proper)),
has_modifier:HEAD),
  comp1:status:nil,
  comp0:sign:sign,
  comp2:null_argument,
  index:1, index2:LINK,
  (@nogap))).

right_mod_rule
  (aroles_sign,
   'Macro':ruleright_mod,
   features:F,
   @indices(10, 11, 12),
   mod:Mod,
   gap:GAP,
   comp0:C00,
   comp1:C11,
   comp2:C22,
   index:1)
  
  ==> 

cat>
  (aroles_sign,
   mod:{sign:C0,
        status:yes,
        dir:right),
   comp0:status:nil,
   comp1:status:nil),

cat>
  (C0,
   aroles_sign,
   features:F,
   @indices(10, 11, 12),
mod: Mod,
gap: GAP,
comp0: C00,
comp1: C11,
comp2: C22,
index: l}.
Appendix B

The SQL Grammar and Signature

QUERIES

query1 rule
(query, \$ix(X)) ===> 
  cat> (select_keyw, \$ix(X,Y)),
  cat> (columns, \$ix(Y)),
  cat> (from_keyw, \$ix(X,Z)),
  cat> (tables, \$ix(Z)),
  cat> (where_keyw, \$ix(X,W)),
  cat> (conditions, \$ix(W)).

query2 rule
(query, \$ix(X)) ===> 
  cat> (select_keyw, \$ix(X,Y)),
  cat> (columns, \$ix(Y)),
  cat> (from_keyw, \$ix(X,Z)),
  cat> (tables, \$ix(Z)).

FUNCTIONS AND THEIR ARGUMENTS
APPENDIX B. THE SQL GRAMMAR AND SIGNATURE

func_expr rule
(func_expr, @ix(X)) ===> 
  cat> (func_name, @ix(X)),
  cat> (open_parenth, @ix(X)),
  cat> (func_args, @ix(X)),
  cat> (close_parenth, @ix(X)).

func_args0 rule
(func_args, @ix(I)) ===> 
  cat> (columns, @ix(I)).

COLUMNS

columns0 rule
(columns, @ix(I)) ===> 
  cat> (column, @ix(I)).

columns1 rule
(columns, @ix(I)) ===> 
  cat> (table, @ix(T)),
  cat> (period, @ix(I,T,C)),
  cat> (column, @ix(C)).

columns2 rule
(columns, @ix(I)) ===> 
  cat> (column, @ix(I)),
  cat> comma,
  cat> (columns, @ix(I)).

columns3 rule
(columns, @ix(I)) ===>
APPENDIX B. THE SQL GRAMMAR AND SIGNATURE

\[\text{cat} \rightarrow (\text{table, } \$\text{ix}(T)),\]
\[\text{cat} \rightarrow (\text{period, } \$\text{ix}(I,T,C)),\]
\[\text{cat} \rightarrow (\text{column, } \$\text{ix}(C)),\]
\[\text{cat} \rightarrow (\text{columns, } \$\text{ix}(I)).\]

(columns4 rule)
\[(\text{columns, } \$\text{ix}(I)) \rightarrow \text{cat} \rightarrow (\text{func.expr, } \$\text{ix}(I)).\]

TABLES

(tables0 rule)
\[(\text{tables, } \$\text{ix}(I)) \rightarrow \text{cat} \rightarrow (\text{table, } \$\text{ix}(I)).\]

(tables1 rule)
\[(\text{tables, } \$\text{ix}(I)) \rightarrow \text{cat} \rightarrow (\text{table, } \$\text{ix}(I)),\]
\[\text{cat} \rightarrow \text{comma},\]
\[\text{cat} \rightarrow (\text{tables, } \$\text{ix}(I)).\]

CONDITIONS

(conditions0 rule)
\[(\text{conditions, } \$\text{ix}(I,I)) \rightarrow \text{cat} \rightarrow (\text{condition, } \$\text{ix}(I)).\]

(conditions1 rule)
\[(\text{conditions, } \$\text{ix}(I,K)) \rightarrow \text{cat} \rightarrow (\text{conditions, } \$\text{ix}(I,K)).\]
APPENDIX B. THE SQL GRAMMAR AND SIGNATURE

```
cat> (condition, Φix(I)),
cat> (connector, Φix(C,J)),
cat> (conditions, Φix(J,K)).

conditions2 rule
(conditions, Φix(I,K)) ===>>
  cat> (condition, Φix(I)),
  cat> (connector, Φix(C,I,J)),
  cat> (conditions, Φix(J,K)).

conditions3 rule
(conditions, Φix(I,I), ===>>
  cat> (open_parenth, Φix(I,J)),
  cat> (conditions, Φix(J,K)),
  cat> (close_parenth, Φix(I,K)).

INDIVIDUAL CONDITIONS

condition rule
(condition, Φix(I)) ===>>
  cat> (table_col_pair, Φix(J)),
  cat> (comparison_op, Φix(I,J,K)),
  cat> (value_expr, Φix(K)).

condition1 rule
(condition, Φix(I)) ===>>
  cat> (column, Φix(J)),
  cat> (comparison_op, Φix(I,J,K)),
  cat> (value_expr, Φix(K)).
```
TABLE/COLUMNAI PAIRS

table_col_pair rule
(table_col_pair, @ix(I)) ==> cat > (table, @ix(J)),
      cat > (period, @ix(J,K)),
      cat > (column, @ix(K)).

VALUE EXPRESSIONS

value_expr rule
(value_expr, @ix(I)) ==> cat > (table_col_pair, @ix(I)).

value_expr1 rule
(value_expr, @ix(I)) ==> cat > (query, @ix(I)).

value_expr2 rule
(value_expr, @ix(I)) ==> cat > (constant, @ix(I)).

value_expr3 rule
(value_expr, @ix(I)) ==> cat > (value_set, @ix(I)).

SETS AND SET ELEMENTS

value_set rule
(value_set, @ix(I)) ==> cat > (open_parenth, @ix(I)),
APPENDIX B. THE SQL GRAMMAR AND SIGNATURE

\[\text{cat}\text{>} (\text{value\_set\_elems}, \text{@ix}(J)), \]\n\[\text{cat}\text{>} (\text{close\_parenth}, \text{@ix}(I)). \]

\text{value\_set\_elems \ rule \ (value\_set\_elems, \text{@ix}(I)) \Longrightarrow \ cat\text{>} (\text{constant}, \text{@ix}(J)).\]

\text{value\_set\_elems1 \ rule \ (value\_set\_elems, \text{@ix}(I)) \Longrightarrow \ cat\text{>} (\text{constant}, \text{@ix}(J)), \ cat\text{>} (\text{comma}), \ cat\text{>} (\text{value\_set\_elems}, \text{@ix}(J)).\]

MACROS

\text{ix}(X) \ macro \ ]
\[
\text{(index}:\text{(nonnull\_index},X), \index0:\text{null\_index}, \index1:\text{null\_index}).
\]

\text{ix}(X,Y) \ macro \ ]
\[
\text{(index}:\text{(nonnull\_index},X), \index0:\text{nonnull\_index}, \index1:\text{null\_index}).
\]

\text{ix}(X,Y,Z) \ macro \ ]
\[
\text{(index}:\text{(nonnull\_index},X), \index0:\text{nonnull\_index}, \index1:\text{nonnull\_index}, \index1:\text{null\_index}).
\]

\text{in\_cond}(X) \ macro
SIGNATURE
bot sub [category, index, boolean].
category sub [table, not_table]
intro [index:index, index0:index, index1:index].
table intro [in_cond:boolean].

not_table sub [select_keyw, from_keyw, where_keyw, comparison_op, connector, query, constant, table_col_pair, column, period, comma, columns, tables, conditions, condition, value_expr, func_name, func_args, open_parenth, close_parenth, func_expr, value_set, value_set elems].

boolean sub [yes, nil].
yes sub [ ].
nil sub [ ].

comparison_op sub [less_than, greater_than, equal].

less_than sub [ ].
greater_than sub [ ].
equal sub [ ].

select_keyw sub [ ].
from_keyw sub [ ].
where_keyw sub [ ].
connector sub [and, or].

and sub [].
or sub [].

query sub [].
subquery sub [].
constant sub [].
table_col_pair sub [].
column sub [].
period sub [].
comma sub [].
columns sub [].
tables sub [].
conditions sub [].
table sub [].
condition sub [].
value_expr sub [].
func_expr sub [].
func_name sub [].
func_args sub [].
open_parenth sub [].
close_parenth sub [].

index sub [null_index, non_null_index].
non_null_index sub [].
null_index sub [].

Appendix C

Transfer Macros

The following transfer macros are used in the system:

- **trans.select**: Adds the 'SELECT' and 'FROM' keywords to the TL bag. Takes as arguments the indices of a single column name and a single table name.

- **trans.tables**: takes as arguments a table name and the index of a table and places that table in the tables list (properly coindexed), if it is not already present. If it is, the macro does nothing. To implement this, is is necessary to write two definitions of this macro for each table name in the database model.

- **trans.cond.and**: takes as argument the index of a condition, and either adds the 'WHERE' keyword to the TL bag, properly coindexed with the condition (if that keyword is not already present) or adds an 'AND' keyword to the TL bag.

**TRANS.SELECT**

```plaintext
trans_select(X,T) tmacro
    []
    < - >
    'SELECT':: (select_keyw, @ix(Y, X)) &
    'FROM' :: (from_keyw, @ix(Y,T)).
```

**TRANS.TABLES**

109
trans_tables(X, und_students) tmacro

[]
< - >
'FROM' ::: (from_keyw, \$ix(\ldots X)) &
und_students ::: (table, \$ix(X), \$in_cond(nil)).

trans_tables(X, gr_students) tmacro

[]
< - >
'FROM' ::: (from_keyw, \$ix(\ldots X)) &
gr_students ::: (table, \$ix(X), \$in_cond(nil)).

trans_tables(X, stud_dept) tmacro

[]
< - >
'FROM' ::: (from_keyw, \$ix(\ldots X)) &
stud_dept ::: (table, \$ix(X), \$in_cond(nil)).

trans_tables(X, dept_fac) tmacro

[]
< - >
'FROM' ::: (from_keyw, \$ix(\ldots X)) &
department_fac ::: (table, \$ix(X), \$in_cond(nil)).

trans_tables(X, courses) tmacro

[]
< - >
'FROM' ::: (from_keyw, \$ix(\ldots X)) &
courses ::: (table, \$ix(X), \$in_cond(nil)).

trans_tables(X, grades) tmacro

[]
< - >
APPENDIX C. TRANSFER MACROS

'FROM' :: (from_keyw, Gix(-.X)) &
grades :: (table, Gix(X), Gin_cond(nil)).

trans_tables(X.departments) tmacro

[]

'FROM' :: (from_keyw, Gix(-.X)) &
departments :: (table, Gix(X), Gin_cond(nil)).

trans_tables(X.classes) tmacro

[]

'FROM' :: (from_keyw, Gix(-.X)) &
classes :: (table, Gix(X), Gin_cond(nil)).

trans_tables(X.faculties) tmacro

[]

'FROM' :: (from_keyw, Gix(-.X)) &
faculties :: (table, Gix(X), Gin_cond(nil)).

trans_tables(X.class_prof) tmacro

[]

'FROM' :: (from_keyw, Gix(-.X)) &
class_prof :: (table, Gix(X), Gin_cond(nil)).

trans_tables(X.professors) tmacro

[]

'FROM' :: (from_keyw, Gix(-.X)) &
professors :: (table, Gix(X), Gin_cond(nil)).
APPENDIX C. TRANSFER MACROS

trans-tables(X,und_students) tmacro

[]
  < - >
  'FROM' ::: (from_keyw, \$ix(_.X)) &
  ',' :: (comma) &
  und_students :: (table, \$ix(X), \$in_cond(nil)).

trans-tables(X,gr_students) tmacro

[]
  < - >
  'FROM' ::: (from_keyw, \$ix(_.X)) &
  ',' :: (comma) &
  gr_students :: (table, \$ix(X), \$in_cond(nil)).

trans-tables(X,stud_dept) tmacro

[]
  < - >
  'FROM' ::: (from_keyw, \$ix(_.X)) &
  ',' :: (comma) &
  stud_dept :: (table, \$ix(X), \$in_cond(nil)).

trans-tables(X,dept_fac) tmacro

[]
  < - >
  'FROM' ::: (from_keyw, \$ix(_.X)) &
  ',' :: (comma) &
  dept_fac :: (table, \$ix(X), \$in_cond(nil)).

trans-tables(X,grades) tmacro

[]
  < - >
  'FROM' ::: (from_keyw, \$ix(_.X)) &
  ',' :: (comma) &
APPENDIX C. TRANSFER MACROS

grades :: (table, @ix(X), @in_cond(nil)).

trans_tables(X,courses) tmacro
[ ]
< - >
'FROM' :: (from_keyw, @ix(._X)) &
',' :: (comma) &
courses :: (table, @ix(X), @in_cond(nil)).

trans_tables(X,departments) tmacro
[ ]
< - >
'FROM' :: (from_keyw, @ix(._X)) &
',' :: (comma) &
departments :: (table, @ix(X), @in_cond(nil)).

trans_tables(X,classes) tmacro
[ ]
< - >
'FROM' :: (from_keyw, @ix(._X)) &
',' :: (comma) &
classes :: (table, @ix(X), @in_cond(nil)).

trans_tables(X,faculties) tmacro
[ ]
< - >
'FROM' :: (from_keyw, @ix(.,X)) &
',' :: (comma) &
faculties :: (table, @ix(X), @in_cond(nil)).

trans_tables(X,class_prof) tmacro
[ ]
< - >
APPENDIX C. TRANSFER MACROS

\[
\text{'FROM' ::} \text{ (from\_keyw, \text{ \texttt{Qix(\_X)}) \&}
\]
\[
\text{',' ::} \text{ (comma) \&}
\]
\[
\text{class\_prof ::} \text{ (table, \text{ \texttt{Qix(X), \texttt{Qin\_cond(nil)})}.}
\]

\[
\text{trans\_tables(X,professors) tmacro}
\]
\[
[ ]
\]
\[
\text{< \text{ \texttt{- > }}}
\]
\[
\text{'FROM' ::} \text{ (from\_keyw, \text{ \texttt{Qix(\_X)}) \&}
\]
\[
\text{',' ::} \text{ (comma) \&}
\]
\[
\text{professors ::} \text{ (table, \text{ \texttt{Qix(X), \texttt{Qin\_cond(nil)})}.}
\]

\text{TRANS\_COND\_AND}

\[
\text{trans\_cond\_and(X) tmacro}
\]
\[
[ ]
\]
\[
\text{< \text{ \texttt{- > }}}
\]
\[
\text{'FROM' ::} \text{ (from\_keyw, \text{ \texttt{Qix(W,)}) \&}
\]
\[
\text{'WHERE' ::} \text{ (where\_keyw, \text{ \texttt{Qix(W,)}) \&}
\]
\[
\text{'AND' ::} \text{ \texttt{Qix(A.X)}.}
\]

\[
\text{trans\_cond\_and(X) tmacro}
\]
\[
[ ]
\]
\[
\text{< \text{ \texttt{- > }}}
\]
\[
\text{'FROM' ::} \text{ (from\_keyw, \text{ \texttt{Qix(W,)}) \&}
\]
\[
\text{'WHERE' ::} \text{ (where\_keyw, \text{ \texttt{Qix(W,X))}.}
\]
Bibliography


BIBLIOGRAPHY

