Ph.D. Proposal, Department of Computer Sciences,
University of Texas at Austin, May 2000

Integrating Statistical and Relational Learning for Semantic Parsing: Applications to Learning Natural Language Interfaces for Databases

Lappoon R. Tang
Department of Computer Sciences
University of Texas
2.124 Taylor Hall
Austin, TX 78712
rupert@cs.utexas.edu

Supervising Professor: Dr. Raymond J. Mooney

May 3, 2000

Abstract

The development of natural language interfaces (NLIs) for databases has been an interesting problem in natural language processing since the 70's. The need for NLIs has become more pronounced given the widespread access to complex databases now available through the Internet. However, such systems are difficult to build and must be tailored to each application. A current research topic involves using machine learning methods to automate the development of NLI's. This proposal presents a method for learning semantic parsers (systems for mapping natural language to logical form) that integrates logic-based and probabilistic methods in order to exploit the complementary strengths of these competing approaches. More precisely, an inductive logic programming (ILP) method, TABULATE, is developed for learning multiple models that are integrated via linear weighted combination to produce probabilistic models for statistical semantic parsing. Initial experimental results from three different domains suggest that an integration of statistical and logical approaches to semantic parsing can outperform a purely logical approach. Future research will further develop this integrated approach and demonstrate its ability to improve the automated development of NLI's.

1 Introduction

Semantic parsing refers to the process of mapping a natural language input (a sentence) to some structured meaning representation which is suitable for manipulation by a machine (Allen, 1995). For example, in building a natural language interface (NLI) for a commercial database, one may want to map a user data request expressed in a natural language to the underlying database access language like SQL. The target query which is expressed in SQL, in this case, would serve as the meaning representation for the user request. The choice of a semantic representation language is entirely domain dependent since as of now there has not been developed a "univeral" semantic representation language which is expressive enough to handle the world of possible meaning structures. Semantic parsing is a difficult problem in natural language processing (NLP) since anyone who attempts to approach it would necessarily have to tackle the very difficult task of *natural language understanding*.

Let's begin the discussion in this proposal with two important questions: 1) Why do we care about semantic parsing? and 2) Why do we care about empirical or statistical approaches for this problem?

First, semantic parsing has been an interesting problem in NLP as it would very likely be part of any interesting NLP applications, particularly those that would require translation of a natural language input to a specific command. Research in semantic parsing with a focus on developing natural language interfaces for database querying started in the 70's (Woods, 1970; Waltz, 1978) and carries on to the 90's (Miller, Stallard, Bobrow, & Schwartz, 1996; Zelle, 1995; Kuhn & De Mori, 1995). With the advent of the "information age", the availabilty of such applications would definitely widen the "information delivery bottleneck". Online database access in natural languages makes information available to users who do not necessarily possess the knowledge of the underlying database access language and therefore makes information a lot more accessible. One great potential impact would be on the utility of the World Wide Web where information could be delivered through NLIs implemented as Web pages. The success of semantic parsing techiques would definitely be the cornerstone of a prospering development of such interesting applications.[1]

Second, there has been a resurgence of statistical or empirical approaches to natural language processing since the late 1980's. The success of such approaches in areas like speech recognition (Rabiner, 1989; Bahl, Jelinek, & Mercer, 1983), part-of-speech tagging (Charniak, Hendrickson, Jacobson, & Perkowitz, 1993), syntactic parsing (Ratnaparkhi, 1999; Manning & Carpenter, 1997; Charniak, 1996; Collins, 1997; Pereira & Shabes, 1992), and text or discourse segmentation (Litman, 1996) is evidential. In fact, it has been coined a "revolution" within the NLP community (Hirschberg, 1998). There are reasons why such approaches have experienced a resurgence: 1) the success in information and networking technology has made large volumes of real world corpora of text available (which could serve the role of "raw data" in any empirical approach) and 2) empirical approaches have proven successful to develop systems that satisfy some of the desirable properties of an NLP application, namely a) *acquisition*, automatically acquiring knowledge (domain specific or not) that would be necessary for the task, b) *coverage*, handling the potentially wide range of possibilities that could arise in the application, c) *robustness*, accommodating real data which

[1]Even though building NLIs to databases is being emphasized here, we by no means imply that it would be the only important application of semantic parsing.

may not be "perfect" (like having noise) and still being able to perform reasonably well, d) *portability*, easily applicable to a different task in a new domain (Armstrong-Warwick, 1993).

While statistical approaches to parsing bear the mentioned advantages, most existing methods (especially in syntactic parsing) use a hand-crafted set of contextual features on which probabilistic parsing models are built. The CHILL system (Zelle, 1995) represents an approach to *learning* relevant contextual information (represented as relational knowledge) for the task of disambiguation given complete contexts (i.e. the entire parse state) instead of relying on handcrafting features for parsing. However, the original system builds a deterministic parser organized as a decision list; the first parsing action applicable to the current parse state is selected and applied. One problem of using such a parser is that one misapplied parsing action corrupts the entire parse and no meaningful parse can possibly be constructed. A parsing action could be wrongly chosen due to an over generalization of the control rule learned for the misapplied action (which happens to appear before the right parsing action) or an over specialization for the control rule of the right parsing action (or both). To overcome these robustness problems and yet retain advantages of a (relational) learning approach to parsing, we propose to build a probabilistic framework for the task and integrate *inductive logic programming* (ILP) (the learning method in CHILL) with statistical learning techniques for learning probabilistic semantic parsers. The remainder of the proposal is organized as follows. Section 2 provides a brief overview of research work on semantic and statistical parsing, an overview of CHILL and the working of the parser employed, some background on inductive logic programming, and a brief overview of integrating statistical and relational methods for learning. Section 3 explains the new ILP learning algorithm used for CHILL and how it can be used to learn multiple models. Section 4 describes the probabilistic parsing framework and how multiple learned models found by the ILP algorithm can be integrated with statistical techniques to produce probabilistic models for learning the semantic parser. Section 5 presents preliminary experimental results on the new approach followed by a discussion of them. Section 6 discusses some of the possible future work that could be explored on the current framework. Section 7 provides a brief discussion of related work. Finally, we will summarize and present our conclusions in Section 8.

2 Background

2.1 Semantic Parsing

The early work on semantic parsing can be dated back to the 70's (Reeker, 1976; Siklossy, 1972) with emphasis on discovering learning mechanism for language acquisition and cognitive modelling of human language learning. While some focused on cognitive modelling of language acquisition, others focused on building realistic NLP applications.

Traditional NLP approaches to tackling tasks like building NLIs for databases include *augmented transition networks* (Woods, 1970) which operationalize context-free grammars for producing semantic representations, *semantic grammars* (Hendrix, Sacerdoti, Sagalowicz, & Slocum, 1978; Brown & Burton, 1975) which are context-free grammars in which non-terminals are used to represent domain specific concepts (instead of syntactic categories), and *logic grammars* (Abramson & Dahl, 1989; Warren & Pereira, 1982) which encode linguistic dependencies and structure building operations using logical unification.

Traditional (*rationalist*) approaches to constructing semantic parsers very often involve hand-crafting of expert knowledge represented as rules (maybe with limited automation). However, hand-crafted parsers suffer from problems with robustness and incompleteness, even for domain specific applications. As the task scales up in size, hand-crafting becomes more and more difficult which is the so-called problem of *knowledge engineering bottleneck* that exists in many interesting AI domains. This results in applications that are time-consuming to build and yet perform poorly – incomplete, inefficient, and brittle.

More recent approaches, therefore, have shifted from this knowledge-engineering perspective to a more empirical based paradigm where parsers are constructed through learning algorithms which use a large corpus of training data. For instance, Miller (1995) presents a statistical approach to the task of mapping flight information requests (in English) to SQL which could be used to access the relevant flight information. The frame (or semantic) representation for a given parse tree of the user request which can be further transformed into an SQL is choosen based on statistics collected from training data. A method based on semantic classification trees for parser construction is described in (Kuhn & De Mori, 1995). The classification trees which are used for semantic interpretation are learned from a corpus of training data. Zelle (1995) employs *inductive logic programming* techniques to learn control rules to "specialize" the parser acquired by the CHILL system. We will provide a more thorough discussion of CHILL in Section 2.3 and the two other mentioned systems in Section 7.

2.2 Statistical Parsing

The emergence of statistical approaches to parsing natural languages was largely influenced by the success of using statistical techniques based on hidden Markov models (HMMs) in the area of speech processing (Rabiner, 1989; Bahl et al., 1983). The use of corpus-based learning where a statistical model is learned from a large corpus of annotated text has proven to be very successful (as far as performance is concerned) in part-of-speech (POS) tagging, a task that involves assigning appropriate lexical categories (like noun, verb or article) to the words of a sentence. The level of accuracy was close to even that of human beings (Merialdo, 1994; Charniak et al., 1993; Church, 1988). The influence of corpus-based statistical approach is not only apparent in POS tagging. In syntactic parsing, *probabilistic context-free grammars* (PCFGs) (Charniak, 1996), *probabilistic left-corner grammars* (PLCGs) (Manning & Carpenter, 1997), *dependency grammars* (Collins, 1996), and more recently *maximum entropy models* (Ratnaparkhi, 1999) have been developed for the task of building syntactic trees for sentences. We are going to briefly describe some of the key ideas behind statistical parsing here.

In probabilistic parsing, one is to build a probabilistic *language model* for the language which could be used to rank the different possible parse trees for a given sentence. Intuitively, the idea of building a language model is that given a *generative grammar* G which is assumed to be capable of generating all the sentences s in the language $\mathcal{L}$, one can build a probabilistic model $P(t, s|G)$ for all the parse trees where t is a parse tree resulting from parsing a sentence $s \in \mathcal{L}$ using the grammar G. Since the grammar is assumed to be

Database Category	*Database Objects*
City	cityid(austin,tx)
State	stateid(mississippi)
River	riverid(mississippi)
Place	placeid('death valley')

Table 1: Sample of objects and categories in the Geography database

generative, one can estimate the probability for each derivational rule in the grammar and calculate the probability of a parse tree using these estimates. Normally, one could have a large amount of different parse trees for the same sentence and a heuristic beam search is used to find the most probable parse tree $\hat{t}$ such that

$$\hat{t} = \mathrm{argmax}_t \; P(t, s \mid G). \tag{1}$$

The estimation of the probability of each grammar rule usually relies on some hand-crafted contextual features (except, of course, for a probabilistic context free grammar). For instance, in a probabilistic left-corner grammar, the left-corner (syntactic) category and the goal category of the parse tree are used to decide whether a particular grammar rule should be applied or not.

2.3 Overview of CHILL

Since we will discuss the statistical model we are going to develop for the parser used by CHILL, we will provide a detailed discussion of the system here to explain the working of the parser and how contextual information can be learned and utilized for the parsing operators. The natural language interface developed for a U.S. Geography database is used as an example application here.[2] Further details on the system can be found in (Zelle, 1995).

The (syntactic) structure of a sentence is not enough to express its meaning. For instance, the NP *the catch* can have different meanings depending on whether one is talking about a baseball game or a fishing expedition. To talk about different possible readings of the phrase *the catch*, one therefore has to define each specific sense of the phrase. The representation of the context-independent meaning of a sentence is called its *logical form* (Allen, 1995).

Database items can be ambiguous when the same item is listed under more than one attribute (i.e. a column in a relational database). For example, the term "Mississippi" is ambiguous between being a river name or a state name, in other words, two different *logical forms*, in our U.S. Geography database. The two different senses have to be represented distinctly for an interpretation of a user query. Databases are usually accessed by some well defined structured languages, for instance, SQL. These languages bear certain characteristics similar to that of logic in that they require the expression of quantification of variables (the

[2] It is available via `http://www.cs.utexas.edu/users/ml/geo.html`.

Predicates	*Description*
city(C)	C is a city
capital(S,C)	C is the capital of state S
density(S,D)	D is the population density of state S
loc(X,Y)	X is located in Y
len(R,L)	L is the length of river R
next_to(S1,S2)	state S1 borders S2
traverse(R,S)	river R traverses state S

Table 2: Sample of predicates of interest for a database access

attributes in a database) and the notion of logical operations[3] on them. The different pieces of information in a database may also be related to each other and this relational knowledge could be useful for constructing the parser. First order logic, therefore, becomes our choice of knowledge representation framework for these logical forms of all the database objects, relations and any other information related to representing the meaning of a user query. However, it is not the case that the parser used in CHILL can only work with a strictly logical representation. The choice of a representational scheme is flexible. For instance, CHILL is also applied to a database containing facts about Northern California restaurants and the semantic representation scheme resembles SQL. Some examples of the semantic representation of database items of the U.S. Geography database are shown in Table 1.

We will briefly describe the language used for representing the meaning of a natural language query, the parsing framework employed, and the approach that is taken in CHILL for parser acquisition.

2.3.1 Semantic Representation and the Query Language

The most basic constructs of the representation language are the terms used to describe objects in the database and the basic relations between them. Some examples of objects of interest in the domain are states, cities, rivers and places. We have given semantic categories to these objects. For instance, stateid(texas) represents the database item texas as an object of the database category state. Of course, a database item can be a member of multiple categories.

Database objects do bear relationships to each other or can be related to other objects of interest to a user who is requesting information from it. In fact, a very large part of accessing database information is to sort through tuples that satisfy the constraints imposed by these relationships of database objects in a user query. For instance, in a user query like "What is the capital of Texas?", the data of interest is a city that bears a certain relationship to a state called Texas, or more precisely its capital. The capital/2 relation (or predicate) is ,therefore, defined to handle questions that require them. More of these relations of possible interest to the domain are shown in Table 2.

We also need to handle object modifiers in a user query such as "What is the largest city in California?". The object of interest X which belongs to the database category city

[3]For instance, in SQL, we have AND and OR to express the logical relationships between constraints on the attributes over the query.

Meta-predicates	*Description*
answer(A,Goal)	A is the answer to retrieve in Goal
largest(X, Goal)	X is the largest object satisfying Goal
smallest(X, Goal)	similar to largest
highest(X,Goal)	X is the highest place satisfying Goal
lowest(X,Goal)	similar to highest
longest(X,Goal)	X is the longest river satisfying Goal
shortest(X,Goal)	similar to longest
count(X,Goal,N)	N is the total number of X satisfying Goal
most(X,C,Goal)	X satisfies Goal and maximizes C
fewest(X,C,Goal)	similar to most

Table 3: Sample of meta-predicates used in database queries

has to be the largest one in California and it would be represented as largest(X, (city(X), loc(X,stateid(california)))). The meaning of an object modifier depends on the type of its argument. In this case, it means the city X in California that has the largest population (in the number of citizens). To allow predicates to describe other predicates would be a natural extension to the first order framework in handling these kind of cases. These "meta-predicates" have the property that at least one of their arguments take a conjunction of predicates. Finally, an object which is an argument of a certain predicate can appear at a later point in a sentence and this requires the use of a predicate like const(X,Y) (which means the object X equals the object Y) for the parser to work. The use of const/2 will be further explained in the following section where the working of the parser is discussed. A list of meta-predicates is shown in Table 3. Some sample database queries for the U.S. Geography domain are shown in Table 4.

U.S. Geography
What is the capital of the state with the largest population? answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).
What state has the most rivers running through it? answer(S, most(S, R, (state(S), river(R), traverse(R,S)))).
How many people live in Iowa? answer(P, (population(S,P), const(S,stateid(iowa))).

Table 4: Sample of Geography questions in different domains

2.3.2 Actions of the Parser

The parser presented here that builds a logical query given a sentence is based on a standard *shift-reduce* parsing framework. (A more thorough discussion on shift-reduce parsing can be found in (Allen, 1995; Tomita, 1986).) There is no explicit semantic grammar but the parsing actions are derived from the examples (which is a pair of sentence and its logical query) and they are guaranteed *complete* with respect to them (i.e. there exists a sequence of parsing actions (a derivation) that leads to the right logical query for each sentence). The parser actions are generated from templates given a logical query; an action template will be instantiated to form a specific parsing action. The templates are INTRODUCE, COREF_VARS, DROP_CONJ, LIFT_CONJ, and SHIFT. INTRODUCE pushes a logical form onto the parse stack based on information in the lexicon. COREF_VARS binds two arguments of two different logical forms to the same variable. DROP_CONJ (or LIFT_CONJ) takes a logical form on the parse stack and puts it into one of the arguments of a meta-predicate. DROP_CONJ assumes the logical form precedes the meta-predicate on the parse stack while LIFT_CONJ assumes it is the other way around. SHIFT pushes a word from the input buffer onto the parse stack. Their actions are summarized in Table 5. The parsing actions are tried in exactly that order; the set of parsing actions resemble a decision list in which the first applicable choice is taken.

The parser also requires a lexicon to interpret meaning of phrases into specific logical forms. The lexicon can be learned from a given set of sample sentence and query pairs (Thompson & Mooney, 1999). We will briefly illustrate what action each template does here by showing a trace of parsing a simple example:

Sentence:	What is the capital of Texas?
Logical Query:	answer(C, (capital(C,S), const(S, stateid(texas)))).

The first thing we need is a lexicon. A very simple lexicon that maps 'capital' to 'capital(_,_)' and 'Texas' to 'const(_,stateid(texas))' would suffice here. The parser begins with an initial stack and a buffer holding the input sentence which is the initial parse state. Each predicate on the parse stack has an attached buffer to hold the context in which it was introduced; words from the input sentence are shifted onto the (stack) buffer during parsing. The contextual information may be useful for the learning of contextual knowledge for disambiguation. The initial parse state is shown below:

Parse Stack:	[answer(_,_):[]]
Input Buffer:	[what,is,the,capital,of,texas,?]

Since the first three words in the input buffer do not map to any logical forms, the next sequence of steps are three SHIFT actions which result in the following parse state:

Parse Stack:	[answer(_,_):[the,is,what]]
Input Buffer:	[capital,of,texas,?]

Now, 'capital' is at the head of the input buffer and is mapped to 'capital(_,_)' in our lexicon. The next action to apply is, therefore, INTRODUCE which is actually instantiated to introduce(capital(_,_), [capital], S0, S1). Notice that a particular phrase in general can be mapped to different logical forms due to lexical ambiguities. The contextual knowledge required for the proper interpretation of a phrase is learned by the induction algorithm. The resulting parse state is shown below:

Parse Stack:	[capital(_,_):[], answer(_,_):[the,is,what]]
Input Buffer:	[capital,of,texas,?]

Actions	*Description*
INTRODUCE(TERM, PHRASE, SO, S1)	Put TERM on the parse stack of S0 if PHRASE occurs at the beginning of the input buffer of S0 (S0 is the input parse state and S1 is the output parse state)
COREF_VARS(T1, AR1, N1, T2, AR2, N2, S0, S1)	Unify the N1-th argument of the term T1 with the N2-th argument of the term T2 if T1 and T2 are on the parse stack of S0 having arity AR1 and AR2 respectively
DROP_CONJ(T1, AR1, T2, AR2, N2, S0, S1)	Place the term T1 in the N2-th argument of the term T2 to form a new conjunct if T1 comes before T2 on the parse stack of S0 having arity AR1 and AR2 respectively
LIFT_CONJ(T1, AR1, T2, AR2, N2, S0, S1)	Similar to DROP_CONJ except that the term T2 comes before the term T1 on the parse stack of S0
SHIFT(S0, S1)	A word at the beginning of the input buffer of S0 will be shifted into the buffer of the top predicate on the parse stack of S0 if the input buffer is not empty

Table 5: A summary of the parse actions

The next action is a COREF_VARS. We have two possible choices here: coref_vars(capital, 2, 1, answer, 2, 1, S0, S1) or coref_vars(capital, 2, 2, answer, 2, 1, S0, S1).[4] Since the question is asking about the capital, the first one is the proper choice and we will pick it here. In general, the knowledge required for properly selecting a COREF_VARS action is learned. The resulting parse state is shown below:

Parse Stack: [capital(C,_):[], answer(C,_):[the,is,what]]
Input Buffer: [capital,of,texas,?]

The next sequence of steps are two SHIFT's followed by an INTRODUCE which is instantiated to introduce(const(_,stateid(texas)), [texas], S0, S1). The resulting parse state is:

Parse Stack: [const(_,stateid(texas)):[], capital(C,_):[of,capital], answer(C,_):[the,is,what]]
Input Buffer: [texas,?]

Notice that instead of looking ahead into the input buffer for 'Texas' and introducing 'capital(_,stateid(texas))', we introduced 'capital(_,_)' and its second argument is left to be instantiated by a COREF_VARS when the parser comes to the term 'Texas'. This helps avoid the problem of having to combine different disambiguation decisions at the same point. For instance, if the question was "What is capital of the state that borders Texas?", we would have to make a decision between introducing 'capital(C,stateid(texas))' or 'capital(C,_)' precisely at the point where 'capital' was at the beginning of the input buffer. It would be easier for the parser to make such decisions when the relevant context become available on the parse stack at a later point.

The next sequence of actions are COREF_VARS which is instantiated to coref_vars(const, 2, 1, capital, 2, 2, S0, S1) and two more SHIFT. Again, we have two possible COREF_VARS instantiations here, the proper one was chosen. The resulted parse state is shown below:

Parse Stack: [const(S,stateid(texas)):[?,texas], capital(C,S):[of,capital], answer(C,_):[the,is,what]]
Input Buffer: []

[4]A choice like coref_vars(capital, 2, 1, answer, 2, 2, S0, S1) is eliminated by inspecting if one of the predicates is a meta-predicate and which argument positions hold variables.

Now, the next steps would be two DROP_CONJ. (Remember that a DROP_CONJ can be applied only if all the variables of the predicate being dropped are instantiated by previous actions from COREF_VARS.) They are drop_conj(const, 2, answer, 2, 2, S0, S1) and drop_conj(capital, 2, answer, 2, 2, S0, S1). The resulted parse state is:

Parse Stack:	[answer(C, (capital(C,S), const(S,stateid(texas)))):[?,texas,of,capital,the,is,what]]
Input Buffer:	[]

We have reached the final parse state at this point since none of the parser actions can be applied. The logical query constructed is then read off from the parse stack.

2.3.3 The CHILL approach

The CHILL (Constructive Heuristic Induction for Language Learning) framework is based on an empirical approach to parser construction integrated in a symbolic knowledge acquisition framework for both the learning and the representation of semantic knowledge. Since the parser is to construct logical queries from natural language input, it would be natural to implement the parser as a logic program where the parsing operators are actually Horn clauses.

Given a corpus of sentence and query pairs, the task is to induce a parser that can translate these sentences to the appropriate queries (which can be further mapped to a target database access language). If we consider inducing a parser directly from these pairs, the space of possible parsers would be too large and unfortunately we still have not developed enough in machine learning to handle a task of this complexity. However, if we begin with an initial parser generated by instantiating the action templates given the examples (which can also be viewed as an overly general initial domain theory), the problem of inducing a parser could be reduced to learning control rules for it. Induction Logic Programming (ILP) techniques for learning search control knowledge will be used since a (first-order) logical knowledge representation framework is employed. ILP is a growing paradigm in machine learning and it will be further discussed in the next section. The idea of learning control rules for a parser can also be traced back to earlier work in acquiring syntactic knowledge for parsing (Berwick, 1985). Figure 1 shows the architecture for CHILL.

The working of CHILL is divided into four phases as indicated in the figure: 1) generating the initial parser, 2) analysing the examples, 3) inducing the control rules, and 4) specializing the initial parser. We will briefly describe each of them here.

Initial Parser Generation. Given the set of training examples and the lexicon, the initial parser is generated by instantiating each template $X \in \{$ INTRODUCE, COREF_VARS, DROP_CONJ, LIFT_CONJ, SHIFT $\}$ to a set of specific parsing actions. Given a template X and an example, a specific parsing action can be obtained by inspecting the logical query and the sentence. For instance, in the previous example "What is the capital of Texas?", a specific action can be obtained from INTRODUCTION by instantiating TERM to capital(_,_) and PHRASE to [capital] given the lexicon, which gives us introduce(capital(_,_), [capital], S0, S1). Actions that are not used in parsing an example will also be generated due to natural language ambiguities. For example, if 'capital' was mapped to 'money(M,G)' (e.g. the amount of money M the state government G has) in the lexicon as well, we would

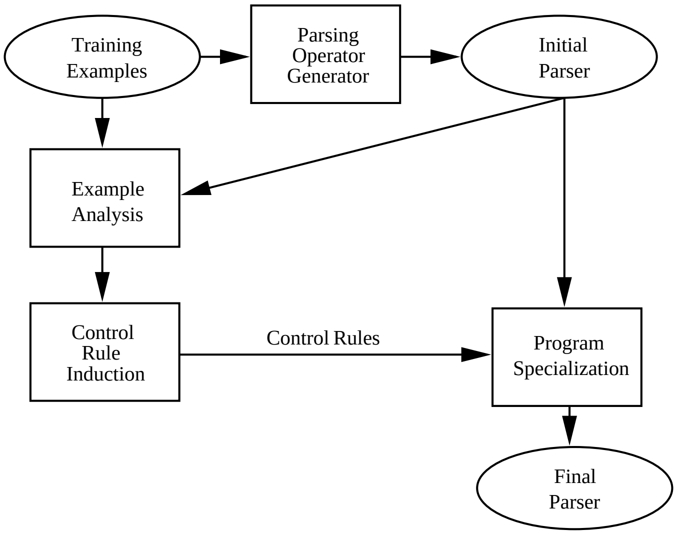

Figure 1: The architecture for CHILL

have generated the additional (redundant) action introduce(money(_,_), [capital], S0, S1). The set of actions generated for all the training examples become the initial *overly general* parser. The initial parser is guaranteed complete with respect to the examples.

Example Analysis. The initial parser will produce many spurious parses for the training examples since it may contain redundant actions and there may be ambiguities among which actions should be employed if more than one can be applied given a particular parse state. The job of example analysis is to determine the right set of actions to be used and exactly which action should be applied to which parse state to correctly parse an example (i.e. construct the right logical query). The result of example analysis is a set of actions that were used to parse the given training example paired with the corresponding parse states to which they were applied. For instance, in the previous example, the parse state that consists of the parse stack [answer(_,_):[the,is,what]] and the input buffer [capital,of,texas,?] would be paired with introduce(capital(_,_), [capital], S0, S1). The training exmaples are analysed one by one and all of the results are collected together into a final set. These parse states with the actions will be useful for inducing control rules for the parser.

Control Rule Induction. After example anaylsis is done, we will have a set of actions in some order[5] where each action is coupled with the set of parse states to which it was successfully applied. All these parse states will become the positive examples for the control rule induction component and any parse states applicable to one action but paired with any other actions below this action are considered negative examples. Parse states that appear above the action can be ignored since the parser is implemented as a decision list and therefore if an action applicable to some of these states is applied, the parser will not backtrack (through Prolog's backtracking mechanism) to any other actions further down in the decision list. Since in general there could be more than one possible derivation for a sentence (i.e. two or more different ways of getting the correct logical query from a

[5]Actually, they will be grouped in the order of COREF_VARS, INTRODUCE, DROP_CONJ, LIFT_CONJ, and SHIFT.

sentence), we can have "false" negative examples which are those to which more than one parsing action can be applied and yet the different resulting states would lead to the same final state. Checking for all these false negative examples is expensive computationally and we simply treat them as noise in the training data and rely on the induction algorithm for noise handling. We will discuss more about noise handling in Section 3.7. In principle, any ILP algorithm can be used here to induce the control rules. The particular algorithm employed by CHILL will be discussed in the next section.

Program Specialization. The control rule produced for each action can be viewed as some kind of "guard" for that action. The rule would (hopefully) capture all the contextual knowledge present in the parse states necessary for classifying which (future) parse states the action should apply to. These control rules will be incorporated into the initial overly general parser to specialize it.

2.4 Inductive Logic Programming

Inductive Logic Programming (ILP) is a growing subfield in AI at the intersection of machine learning and logic programming. The problem is defined as follows. Given a set of examples $\xi = \xi^+ \cup \xi^-$ consisting of positive and negative examples of a target concept, and background knowledge B, find an hypothesis $H \in \mathcal{L}$ (the language of hypotheses) such that the following conditions hold[6] (Muggleton & Raedt, 1994).

Prior Satisfiability. $B \wedge \xi^- \not\models \Box$
Posterior Satisfiability. $B \wedge H \wedge \xi^- \not\models \Box$
Prior Necessity. $B \not\models \xi^+$
Posterior Sufficiency. $B \wedge H \models \xi^+$

The sufficiency criterion is also called *completeness* with regard to positive examples and the posterior satisfiability criterion is also known as *consistency* with the negative examples. Due to the use of a more expressive first-order formalism, ILP techniques are proven to be more effective in tackling problems that require learning relational knowledge than traditional propositional approaches (Quinlan, 1990).

There are two major approaches in the design of ILP learning algorithms: top-down and bottom-up. Both approaches can be viewed more generally as a kind of *set covering algorithm*. However, they differ in the way a clause is constructed. In a top-down approach, one builds a clause in a general to specific order where the search usually starts with the most general clause and successively specializes it with background predicates according to some search heuristic. A representative example of this approach would be the FOIL algorithm (Quinlan, 1990; Cameron-Jones & Quinlan, 1994). In a bottom-up approach, the search begins at the other end of the space where it starts with the most specific hypothesis, the set of examples, and constructs the clauses in a specific to general order by generalizing the more specific clauses. A representative example of this approach would be the GOLEM algorithm (Muggleton & Feng, 1992). These two ILP systems are briefly reviewed here.

[6] This problem setting is also called the *normal semantics* of ILP.

Procedure *Foil*
Input:
$R(V_1, V_2, ..., V_k)$: the target concept
ξ^+: the $\oplus$ examples
ξ^-: the $\ominus$ examples
Output:
H: the set of learned clauses

```
H := ∅
Positives-To-Cover := ξ⁺
While Positives-To-Cover is not empty Do
    /*
      Search for a clause that covers a preferably large subset of Positives-To-Cover
      but covers no examples in ξ⁻. And add the clause to the building definition.
    */
    C := R(V1, V2, ..., Vk) ←
    T := Positives-To-Cover ∪ ξ⁻
    While T contains negative tuples Do
        Find the literal L that maximizes Gain(L) to add to the clause C
        Form a new set T' by extending each tuple t in T that satisfies L with its new variable bindings
    Replace T by T'
    End While
    Add C to H
    Remove examples covered by C from Positives-To-Cover
End While
Return H
End Procedure
```

Figure 2: The FOIL algorithm

2.4.1 Foil

FOIL learns a function-free first-order Horn clause definition of a target concept given the *background predicates* which are defined extensionally. FOIL contains an outer loop which finds clauses that covers a portion of the positive examples and are consistent with the negative examples. The loop stops when the set of clauses found covers all the positive examples. The inner loop builds a single clause, starting with the most general hypothesis and adding literals to it until it covers no negative examples. Literals are ranked using the information gain metric and the literal that maximizes gain is chosen. More formally, let T_+ denote the number of positive tuples in the set T, the information of T is defined as:

$$I(T) = -\log_2(T_+ / \mid T \mid). \tag{2}$$

And, the information gain of a literal L is defined as:

$$Gain(L) = s \cdot (I(T) - I(T')) \tag{3}$$

where s is the number of tuples in T that have extensions in T' (i.e. the number of current positive tuples covered by L) and T' is the new training set created from T and L. Figure 2

Procedure *Golem*
Input:
ξ^+: the set of positive examples
ξ^-: the set of negative examples
Pairs: random sampling of pairs from ξ^+
RLggs: $\{C : \langle e, e' \rangle \in Pairs$ and $C = RLGG(e, e')$ and C consistent wrt ξ^- $\}$
S: the pair $\{e, e'\}$ whose RLGG has the greatest cover in *RLggs*
Output:
H: the set of learned clauses

$H := \emptyset$
Do
  $Examples$:= a random sampling of examples from ξ^+
  $RLggs := \{C$: $e' \in Examples$ and $C = RLGG(S \cup e'))$ and C consistent wrt ξ^- $\}$
  Find e' which produces greatest cover in *RLggs*
  $S := S \cup e'$
  Add $RLGG(S)$ to H
  $Examples := Examples - cover(RLGG(S))$
While increasing-cover
Return H
End Procedure

Figure 3: The GOLEM algorithm

summarizes the FOIL algorithm.

2.4.2 Golem

GOLEM also contains an outer loop that find a set of consistent clauses covering the positive examples like FOIL. However, it builds a clause by considering the *relative least general generalization* (RLGG) of the pairs of positive examples. It is called *relative* because the background knowledge is taken into account when performing the *least general generalization* (LGG) on a pair of positive examples; each example has a sequence of ground background literals added as its body. The LGG of two terms $f_1(l_1, ..., l_n)$ and $f_2(m_1, ..., m_n)$ is a new variable v if $f_1 \neq f_2$. Otherwise, it is $f_1(lgg(l_1, m_1), ..., lgg(l_n, m_n))$. The LGG of two clauses $C_1 = \{l_1, \ldots, l_k\}$ and $C_2 = \{m_1, \ldots, m_n\}$ is defined as: $lgg(C_1, C_2) = \{lgg(l, m) \mid l \in C_1$ and $m \in C_2$ and $lgg(l, m)$ is defined$\}$. As a simple example, consider the clauses $C_1 : p(a) \leftarrow q(b)$ and $C_2 : p(c) \leftarrow q(d)$. The $lgg(C_1, C_2)$ would be the clause $p(X) \leftarrow q(Y)$. A more detailed discussion on LGG can be found in (Plotkin, 1970).

The RLGG of two examples can produce a very large clause with a lot of redundant literals in its body. To reduce the clause size, the search only considers a restricted model of the background knowledge K, the h-easy model which is the set of all Herbrand instantiations of h-easy atoms of K. (An atom a is h-easy with respect to K if there is a derivation of a from K involving at most h resolutions.)

GOLEM starts by taking a sampling of RLGGs of pairs of uncovered positive examples and chooses the one that has the greatest coverage for further generalization. It stops building the clause when this RLGG cannot be further generalized (i.e when any further generalization produces inconsistent clauses.) Figure 3 is a summary of the algorithm.

2.5 Integrating statistical and relational methods for learning

In recent years, there has been several methods which combine statistical and relational learning approaches to produce classifiers that are more accurate than using either approach alone (Slattery & Craven, 1998; Ali & Pazzani, 1996). More precisely, we have take an approach which uses a relational learning algorithm to learn multiple models from the training data and combines them using a statistical method. It has been shown that learning multiple models can improve classification accuracy (Ali & Pazzani, 1996; Kwok & Carter, 1990; Baxt, 1992; Breiman, 1996; Quinlan, 1996a). We are going to briefly review a system which is most similar to our approach called HYDRA which uses a form of Bayesian learning to combine multiple models acquired by a relational learning algorithm (Ali & Pazzani, 1995).

The HYDRA system uses a relational learning algorithm very similar to FOIL (Quinlan, 1990), which is used to learn multiple descriptions from a given set of training data. Two particular approaches were employed by the system, namely stochastic hill-climbing (Kononenko & Kovacic, 1992) and k-fold partition learning (Gams, 1989). In stochastic hill-climbing, one stores a set of literals that are within some margin σ of the best literal ranked according a certain metric on literals (like info-gain) and then chooses a literal randomly from that set. The probability of a literal being chosen is propotional to its goodness according to the metric (like the gain of the literal). In k-fold partition learning, one generates k models by partitioning the training data in k sets of equal size and then learn a model for all but the ith partition. (There are totally k ways of excluding a partition.) After learning a set of models using either methods, HYDRA then combines all the learned models into a single model by Bayesian combination (Buntine, 1990).

3 Learning Multiple Models via TABULATE

In this section, we are going to discuss the details of the new induction algorithm used by CHILL. Since its design is strongly motivated by the CHILLIN induction algorithm (Zelle & Mooney, 1994), we are going to provide a brief overview of the algorithm here. Then, we will proceed to explain the motivation for the new algorithm and discuss its details and various ILP issues it addresses.

3.1 Combining Top-down and Bottom-up Methods in CHILLIN

Top-down or bottom-up approaches to ILP have their own strength and weaknesses. For instance, GOLEM requires the use of extensional background knowledge (i.e. background knowledge expressed by listing ground facts) and could result in building clauses with a lot of redundant literals. Specific constraints like using only the h-easy model of the background knowledge have to be enforced to deal with some of these efficiency problems. On the other hand, FOIL requires the target hypothesis to be function-free and therefore needs specific constructor functions as part of the background knowledge. For instance, to learn the concept member(X, Y) where X is an element of the (non-empty) list Y. One needs to provide FOIL with a background predicate like head(Y, H) which is (essentially) a function that returns the first element H of the list Y. The size of the background knowledge grows with the number of such constructor functions used by the learner and so does the complexity of the hypothesis space to search (i.e. a larger branching factor). A combination of the two different methods which takes advantage of the strength of each approach may therefore open up new approaches to ILP that could perform better than either alone. CHILLIN[7] was an attempt at combining both approaches and it was used in CHILL for learning control rules. Figure 4 shows the outline of CHILLIN.

Procedure *Chillin*
Input:
ξ^+: the $\oplus$ examples
ξ^-: the $\ominus$ example
Output:
DEF: the set of learned clauses

$DEF := \{E \leftarrow \mid E \in \xi^+\}$
Repeat
  $PAIRS :=$ a sampling of pairs of clauses from DEF
  $GENS := \{G \mid G = Find\_A\_Clause(C_i, C_j, DEF, \xi^+, \xi^-)$ for $\langle C_i, C_j\rangle \in PAIRS\}$
  $G :=$ Clauses in $GENS$ yielding most compaction
  $DEF := (DEF -$ (Clauses subsumed by G)) $\cup\ G$
Until no-further-compaction
Return DEF
End Procedure

Figure 4: Outline of the CHILLIN algorithm

Unlike set-covering alogrithms like FOIL, CHILLIN consists of a compaction outer loop that builds a more general hypothesis with each iteration. Each iteration tries to find a clause that maximizes the coverage of the set of positive examples (i.e. most compaction). A clause is built by finding the LGG of a random pairs of *clauses* in the building definition (i.e. *DEF*) and if the LGG is overly general, it will be specialized by adding literals to its body in a way very similar to FOIL. The search for a hypothesis is therefore done in a bottom-up manner since it begins with the most specific hypothesis (i.e. the set of positive examples) and continues to generalize it through the compaction loop. The specialization of a clause, however, resembles that of a top-down algorithm as literals are added to its body for specialization and therefore heuristics like information gain can be used to discriminate between literals.

Once a clause is found, it will be incorporated into the current theory. Any clause covered (subsumed) by it will be removed from the theory. A novel kind of subsumption is adopted here which is called *empirical subsumption*. A clause $C \preceq_e$ (empirically subsumes) D if the set of ground unit instances covered by the clause C is a subset of that of the clause

[7]CHILLIN stands for *the CHILL INduction algorithm*.

D.

3.2 Motivation for the New Algorithm

CHILLIN has been tested on learning simple list processing programs (like append/3) and was shown to be more effective than either FOIL or GOLEM (Zelle & Mooney, 1994). While CHILLIN was shown to be a successful attempt to combining top-down and bottom-up approaches in ILP, it suffers from several weaknesses. First, the search is basically a hill-climbing search in the hypothesis space which may sometimes get stuck on a local minimum. It may be interesting to consider other search methods like beam search which attempt to partially overcome local minima. Using beam search, however, requires the design of a metric for evaluating each hypothesis in the beam. We explain the details of our metric in Section 3.3. The goal of the search is to find the simplest consistent hypothesis (or a queue of hypotheses) guided by an explicitly defined theory evaluation metric. The new algorithm is described in more detail in Section 3.4.

Second, CHILLIN relies on the LGGs of the positive examples to capture "theory constants" (logical constants that appear in the definition) that may be useful. However, using LGGs to learning these theory constants is not a very reliable method. For instance, if a relevant constant like a certain word appears at different locations in the input buffers of two different parse states, the result of the LGGs of the two parse states would replace the word by a variable, and therefore could fail to capture useful lexical information. Thus, we consider making literals with theory constants an explicit process and incoporate appropriate predicates for these literals in the background knowledge for the learner. This is further described in Section 3.5.

Third, we have considered adding the capability of handling noisy data which is lacking in CHILLIN. As we mentioned before, noisy data can arise due to the presence of "false negative" examples. We will further discuss the issue of noise handling in Section 3.7.

Finally, the new algorithm is designed to learn multiple models from the training data so that they can be integrated via statistical learning methods to produce probabilistic models for the semantic parser. All these observations were the motivation behind the design of TABULATE.[8]

3.3 Compression and Accuracy

The "ideal" solution to an ILP problem is the hypothesis that has the minimum size and the most predictive power. In practice, however, this is hardly achievable due to two reasons: 1) finding a minimum program is undecidable (because the minimum encoding function is not computable) and 2) it is uncertain whether the minimum program will have the most predictive power. Some form of bias that would lead us "close" to this ideal would still be desirable as the goal of the search is to find hypotheses that perform well. Despite arguments on its generality (Webb, 1996), the Occam's Razor Principle[9] has been the most widely used form of bias in many learning algorithms and it has its basis in algorithmic

[8]TABULATE stands for *Top-down And Bottom-Up cLAuse construction with Theory Evaluation.*

[9]The famous Occam's Razor Princple of Willam of Ockham says that "entities should not be multiplied beyond necessity."

complexity theory. The Occam's Razor Principle is perhaps best followed in the Rissanen's *Minimum Description Length (MDL)* approach (Rissanen, 1978) where the hypothesis H which minimizes the theoretical minimum encoding $K(H|E)$[10] is chosen given $H \models E$, E is the set of examples and H is a hypothesis in a well defined hypothesis space. (The theoretical minimum encoding is not computable and in practice one employs a "reasonable" encoding scheme as an approximation to this notion.) However, it has been found (Califf, 1998) that approaches like *MDL* can sometimes give "too much" weight to the complexity of the hypothesis over its accuracy during the course of the search and may not be the best guide in a heuristic search like hill-climbing or beam search. This leads us to the idea of modifying a metric like *MDL* to one that puts more emphasis on the accuracy of a hypothesis. A reasonable choice of the accuracy metric is the *m-estimate* (Cestnik, 1990). The m-estimate of the expected accuracy is given by[11]

$$accuracy(H) = \frac{s + m \cdot p^{+}}{n + m} \tag{4}$$

where s is the number of positive examples covered by the hypothesis H, n is the total number of examples covered, p^{+} is the prior probability of the class $\oplus$, and m is a parameter.

A metric of calculating the program size similar to one in (Muggleton and Buntine, 1988) is used. The size of a clause C having head H and body B is defined as follows:

$$size(C) = 1 + termsize(H) + termsize(B) \tag{5}$$

$$termsize(T) = \begin{cases} 1 & \text{if } T \text{ is a variable} \\ 2 & \text{if } T \text{ is a constant} \\ 2 + \sum_{i=1}^{arity(T)} termsize(arg_i(T)) & otherwise \end{cases} . \tag{6}$$

The size of a clause roughly counts the number of symbols in it where a symbol can be a variable, a constant or a predicate. The size of a hypothesis which is a finite set of clauses is the sum of the size of the clauses.[12] The metric $M(H)$ as our search heuristic is defined as

$$M(H) = \frac{accuracy(H) + C}{\log_2 size(H)} \tag{7}$$

where H is a hypothesis in the search space and C is a constant. The constant C is used as a balancing factor between the accuracy and complexity of the hypothesis H. The meaning of $M(H)$ is that we are trying to find a hypothesis that optimizes compression and accuracy. Given a hypothesis H in the search space (including inconsistent hypotheses), we want to find the one that compresses the examples the most without losing "too much" classification information (or accuracy). However, we need a balancing factor to set the *a priori* bias between accuracy and complexity in a particular hypothesis space. The constant C is used to set the bias on the accuracy of a hypothesis; the larger the value of C, the more accuracy is emphasized. The constant C may be determined in different ways. However, we make the

[10] $K(\cdot)$ is the Kolmogorov complexity function (Kolmogorov, 1965).

[11] The original definition was given in a more general setting where the classification problem can be n-ary, $n \geq 2$.

[12] A more refined scheme like the one in (Muggleton, Srinivasan, & Bain, 1992) could be employed to estimate the number of bits required for an encoding.

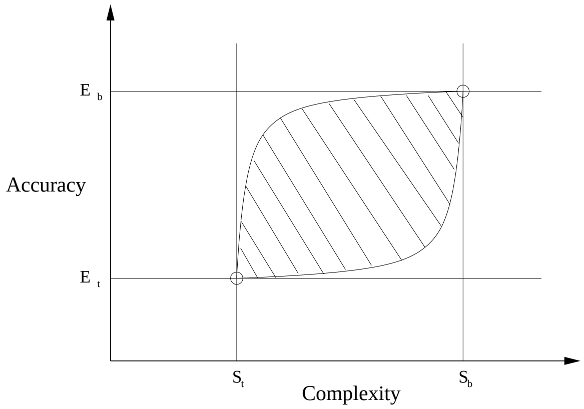

Figure 5: A possible projection of the hypothesis space onto the metric space

assumption that the most general hypothesis $t(X_1, \ldots, X_n) \leftarrow$ is as poor as the most specific hypothesis ξ^+ in terms of their overall quality.[13] Figure 5 shows a possible projection of the hypothesis space onto the metric space. The metric space represents a reduction of the hypothesis space; programs that have the same score are considered equivalent. A more refined definition of the metric M would of course give a better model of the hypotheses (in terms of their goodness). The two extreme points (S_t, E_t) and (S_b, E_b) represent the most general hypothesis and the most specific hypthesis respectively where E_t, E_b are the m-estimates of the most general hypothesis and the most specific hypothesis respectively, and S_t, S_b are their sizes respectively. (More generally, they represent the classes of programs that have the same size and accuracy.) These two points are equally good according to our assumption which is more formally expressed as:

$$M(t(X_1, \ldots, X_n) \leftarrow) = M(\xi^+). \tag{8}$$

According to the definition of M, it is expanded to

$$\frac{E_t + C}{S_t} = \frac{E_b + C}{S_b}. \tag{9}$$

Thus, C can be resolved to

$$C = \frac{E_b S_t - E_t S_b}{S_b - S_t}. \tag{10}$$

We are now ready to describe the new algorithm.

3.4 The Tabulate Algorithm

While top-down or bottom-up methods normally begin the search for hypothesis at one end of the hypothesis space, Tabulate explores both parts of the search space, akin to a bi-directional search. More precisely, it considers both refinements of the most general hypothesis and least general generalizations of existing clauses in a theory.

[13]While in general it may not be the case that they have the same amount of information content (i.e. $I(t(X_1, \ldots, X_n) \leftarrow) \neq I(\xi^+)$), they do represent the worst cases of the two ends of the search space.

TABULATE also has a compaction outer loop. The search starts with the most specific hypothesis which is the set of positive examples.[14] Each iteration of the loop attempts to compact each of the hypotheses in the current search queue. (Each iteration, thus, corresponds to a single search step of generating the children nodes.) The search employed is a modified version of the standard beam search we call w-beam search. A standard beam search considers all the children nodes when selecting the best B from them. However, if the size B is not big "enough" (due to time or space restriction), the search can run into a situation where the best few search nodes (which could look "similar" to each other) will eventually dominate the entire search queue and the search essentially gets stuck on a local minimum. For instance, when a hypothesis has many good refinements (i.e. a clause that is being built in the hypothesis has many good refinements[15]), then all its children will occupy a large part of the search queue and in a few iterations the entire queue will consist only of its descendants. (This problem is not particular to a beam search. In the case of using a genetic algorithm, one could get into the so-called problem of "lack of diversity in the population", which is similar to this problem.) In a w-beam search, a *window* of size w is posted as an upper bound on the number of children nodes the search will consider for a given parent node. In other words, a parent can produce at most w children in the next search queue. This is to partially avoid the *domination* problem mentioned above by "sharing" the search queue among the parents. Notice a w-beam search is reduced to a standard beam search if w is defined as the maximum branching factor of the search (or infinity). There are two cases in each iteration: 1) an existing clause is being refined or 2) a new clause is begun. Since a search node can be refined in a top-down or bottom-up manner, hypotheses found by the algorithm may consist of clauses that are built in either fashion. The TABULATE algorithm has also taken a "compaction-based" approach like that of CHILLIN or CIGOL (Muggleton & Buntine, 1988) in that a clause found by the search is not put aside but included as part of the theory that is being refined. Compaction can sometimes give a more general (consistent) hypothesis than set covering since clauses that are built during the search may be further combined to form more general clauses at a later point in the search. The outline of the TABULATE algorithm is given in Figure 6.

The outline of the refinement operator *Refine_Clause* is given in Figure 7. We have incorporated noise-handling and predicate invention within the refinement operator, both will be discussed further in Section 3.7. There are three possible outcomes in a refinement: 1) the current clause satisfies the noise-handling criterion and does not need to be refined and it will simply be returned ($next_j$ is set to *empty*), 2) the current clause does not satisfy the noise-handling criteria and all possible refinements are returned ($next_j$ is set to *empty* if the jth refined clause satisfies the noise-handling criterion; otherwise, it is set to the refined clause itself), and 3) the current clause does not satisfy the noise-handling criterion but there are no further refinements[16] and an empty set of clauses is returned ($next_j$ is set to *fail*). In practice, there may be too many possible refinements and we decided to choose the

[14] We actually start with a set of LGGs that are correct with respect to the examples and proceed with the search from there. This helps boost the bottom up search a bit when there is a large number of positive examples.

[15] For instance, there may be many literals that yield similar scores on a particular literal selection metric.

[16] We have posted a predicate arity bound K on the predicate invention routine to avoid straight memorization of examples. This case could happen when it requires an invented predicate of arity bigger than K to further refine the clause.

```
Procedure Tabulate
Input:
t(X_1, ..., X_n):   the target concept to learn
ξ+:                 the ⊕ examples
ξ−:                 the ⊖ examples
Output:
Q:                  a queue of learned theories

Theory_0 := {E ←| E ∈ ξ+}              /* the initial theory */
T(N_0) := Theory_0                      /* the theory of the starting search node */
C(N_0) := empty                         /* the current clause which is being built */
Q := [N_0]                              /* the search queue */
Repeat
    CQ := ∅
    For each search node N_i ∈ Q Do
        /*
           If it is time to build a new clause, then start with the most general hypothesis and LGGs
           of existing clauses. Otherwise, keep refining the current clause until it needs no further
           refinement. R_i is the set of clause refinements of node i.
        */
        If C(N_i) = empty or C(N_i) = fail Then
            Pairs := sampling of S pairs of clauses from T(N_i)
            Find the LGG G in Pairs that has the greatest cover in ξ+
            R_i := Refine_Clause(t(X_1, ..., X_n) ←) ∪ Refine_Clause(G ←)
        Else
            R_i := Refine_Clause(C(N_i))
        End If
        /*
           If current clause cannot be refined but is not accurate enough, then we need to build a new
           clause. Otherwise, incorporate the refined clause into the theory of the current search node and
           check if the refined clause is acceptable or still needs to be further refined. CQ is the set
           of children nodes and CQ_i is the children nodes produced from parent node i.
        */
        If R_i = ∅ Then
            CQ_i := {⟨T(N_i), fail⟩}
        Else
            CQ_i := {⟨Complete(T(N_i), G_j, ξ+), next_j⟩ | for each G_j ∈ R_i, next_j = empty if G_j
                                                           satisfies the noise criteria; otherwise, G_j}
        End If
        CQ := CQ ∪ CQ_i
    End For
    Q := the B best nodes from Q ∪ CQ ranked by metric M
Until termination-criteria-satisfied
Return Q
End Procedure
```

Figure 6: The TABULATE algorithm

Procedure *Refine_Clause*
Input:
C: the clause to be refined
K: the arity bound for an invented predicate
BG: the set of background knowledge
N_i: the current search node
Output:
S: the set of refinements of C

$R := T(N_i) - \{C\}$
/*
Check if the clause C is already accurate enough wrt the noise-handling criterion. If so, it is returned. Otherwise, it needs to be refined. If the clause cannot be refined using background literal, it be will checked if inventing predicate for it is suitable (i.e. whether the invented predicate requires arity $\leq K$). An empty set of clauses will be returned if a refinement is not possible, indicating that the search should try to build a different clause.
*/
If C satisfies the noise-handling criterion **Then**
    $S := \{C\}$
Else
    If C can be usefully refined using BG **Then**
        $S := \{G_j \mid G_j := Add\_Literal(L_j, C), \forall L_j$ from $BG\}$
    Else
        If invented predicate requires arity $\leq K$ **Then**
            $\xi^{+\prime} := \{e^+ \in \xi^+ \mid R \not\models e^+\}$
            $\xi^{-\prime} := \{e^- \in \xi^- \mid C \models e^-\}$
            $L_{ip} := Invent\_Predicate(\xi^{+\prime}, \xi^{-\prime}, C)$
            $S := \{Add\_Literal(L_{ip}, C)\}$
        Else
            $S := \emptyset$
        End If
    End If
End If
Return S
End Procedure

Figure 7: The clause refinement operator *Refine_Clause*

best n based on their positive information gains to avoid having to deal with the complexity of a full-fledged search which has to evaluate each of the possible refinements according to the hypothesis evaluation metric.

There two possible cases after each refinement of a clause. In the first case, the refinement is a new clause being built for the theory. In this case, we will check which clauses in the current theory are empirically subsumed by the new refinement. Those that are subsumed are removed from the theory. Existing clauses that are empirically subsumed by the newly refined clause are considered "redundant" and hence eliminated from the theory. In the second case, it is a specialization of an existing clause in the theory. Positive examples that are not covered by the resulting theory, due to specializing the clause, are added back to it. Therefore, the theory is maintained complete during the entire course of the search. These are all done in the *Complete* procedure.

Different possible termination criteria can be defined depending on the goal of the search. For instance, if one wants to emphasize the quality of the solution, one may impose more requirements for termination but at the expense of using more resources. The termination criterion in this case checks for two conditions to be satisfied. First, we have taken the sum of the metric M over each hypothesis in the entire search queue which represents the quality of the queue of hypotheses. The first condition is satisfied if the next search queue does not improve the sum. This is like a hill-climbing search in the space of search queues. Second, we check to make sure that there is no clause currently being built for the theory in each of the search nodes in the search queue and the last finished clause of each theory satisfies the noise-handling criterion (i.e. $C(Node) = empty$ for every $Node$ in the search queue). Finally, a committee of hypotheses found by the algorithm is returned. This set of hypotheses can be viewed as multiple models of the target concept learned by the algorithm.

3.5 Learning with Theory Constants

In domains like language learning, an inductive learner that can utilize or even synthesize theory constants is desirable since this kind of knowledge is valuable to the learner as discriminating features. In parsing, one needs contextual information for disambiguating different possible parses of a sentence and this information can be represented as theory constants to the inductive learner. For example, in our previous sample trace of parsing the sentence "What is the capital of Texas?", if the phrase "capital" was mapped to money(M,G) (the amount of money M the state government G has) in the lexicon as well, we would need to disambiguate between introducing capital/2 or money/2 on the parse stack. In this case, the context that would be helpful for disambiguating between the two cases could be the *absence* of the word government in the input buffer. If we have the predicate phrase_in_buffer(P,S) (the phrase P appears in the input buffer of the parse state S) in the background knowledge of the learner, the literal not phrase_in_buffer([government],S) would be useful for constructing a control rule for the parsing action introduce(capital(_,_), [capital], S0, S1). [17]

ILP systems like FFOIL (Quinlan, 1996b) do make use of background knowledge that

[17] In all the domains described in this proposal, we used the predicates phrase_in_buffer/2 and predicate_on_stack/2 (which checks if a particular predicate is on the parse stack) as the two basic predicates in the background knowledge. In some of the domains, however, more background predicates were used but we are not going into all the details here.

can handle theory constants (e.g. checking if the value of a variable equals zero). However, it requires *a prior* knowledge of the set of constants that will be relevant or necessary for the learning task. This may be possible for domains like learning functional definitions where it would be relatively easier to identify a set of "important" constants that may be relevant to a number of learning tasks (like 0 or 1). In other domains like language learning, however, identifying a set of useful constants that is reasonably comprehensive would be rather difficult as one would be required to have enough prior knowledge of the *relevant* contextual information but this is what the learning system is suppose to find out. Handcrafting some possibilities or throwing in an entire dictionary would be either too ineffective or inefficient. Therefore, instead of engineering them as prior knowledge to the system, we obtain possible theory constants from the training data. This, however, requires the system to generate or extract theory constants from examples given a set of background predicates.[18]

More precisely, the idea is that given a set of positive and negative examples of the target concept and a set of background knowledge in which each background predicate comes with mode declaration and argument-type definition, we generate a new set of literals which would use "constants" that appear in the set of examples for the learner. Using phrase_in_buffer/2 as an example where the sentence "What is the capital of Texas?" is a positive example and "What fraction of the Texas government state capital is spent on highway construction?" is a negative example for the parsing action introduce(capital(_,_), [capital], S0, S1), we generate the following set of literals if we restrict ourselves to considering only one-word phrases:

```
phrase_in_buffer([what],S)
...
phrase_in_buffer([texas],S)
phrase_in_buffer([?],S)
not phrase_in_buffer([what],S)
...
not phrase_in_buffer([texas],S)
not phrase_in_buffer([government],S)
...
not phrase_in_buffer([spent],S)
not phrase_in_buffer([on],S)
not phrase_in_buffer([highway],S)
not phrase_in_buffer([construction],S)
not phrase_in_buffer([?],S)
```

3.6 Predicate Invention

While earlier ILP systems assume a sufficient amount of background predicates to start with, more recent approaches have abandoned that assumption and allow the learner to

[18] The set of background knowledge would have to contain predicates that can take theory constants as their arguments. The set of relevant background predicates, however, are much easier to choose than the set of constants.

extend its vocabulary when the given set of background knowledge is not sufficient for the learning.

The purpose of inventing a literal for a clause is to constrain some of the variables in the clause so as to exclude some negative examples the clause is covering. In general, separating the positive and negative examples may require constraining multiple variables. We have adopted the approach similar to CHAMP (Kijsirikul, Numao, & Shimura, 1992) for predicate invention in TABULATE. A greedy algorithm is employed to find the smallest set of variables that would separate the positive and negative examples; the set of instantiations of the positive and negative examples form two disjoint sets. Such a set always exists as long as the set of positive and negative examples are mutually disjoint. The search begins with an empty set of variables. A variable is added to the set one at a time; the one that separates the most positive examples from the negative examples is chosen first. The set of instantiations of these variables forms the new set of positive and negative examples for the new literal. The predicate invention algorithm (in CHILLIN) can then be recursively called to learn a new concept.

3.7 Noise Handling

In the task of parsing sentences into logical queries, noise could arise when there exists more than one possible consistent logical queries for a given sentence[19]. Different ways of annotating the training data could give rise to ambiguities among the parsing actions which are more "artificial" than lexical; in our case, these could be *false* negative examples collected for the learner. Instead of checking for them when collecting the training data or enforcing a particular "style" of annotation which are both time-consuming and impractical, we treat them as noise in the training data and let the learning algorithm handle them.

We have adopted the approach taken in RIPPER (Cohen, 1995) to handle noise. In particular, a clause needs no further refinement when it meets the following criteria:

$$\frac{p-n}{p+n} > \beta \tag{11}$$

where p is the number of positive examples covered by the clause, n is the number of negative examples covered and $-1 \leq \beta \leq 1$ is the noise parameter. Unlike in RIPPER where β is a fixed constant and selected according to the assumption on the amount of noise in the data, we set β to some high value (i.e. close to 1) and decrease its value whenever the sum of the metric over each hypothesis in the search queue does not improve but some of the search nodes still have an unfinished clause or a failed clause. In other words, we estimate the value of β and therefore the amount of noise in the data instead of relying on assumptions about the data.

[19]Equivalent logical queries for the same sentence do exist since there may be various ways of structuring a query and all of them mean the same thing. For instance, the meaning of answer(C, (capital(C,S), const(S, stateid(texas)))) is the same as answer(C, (state(S), city(C), capital(C,S), const(S, stateid(texas)))); both queries give exactly the same answer. They are, nevertheless, two distinct queries.

4 Statistical Semantic Parsing

Unlike the approach taken by Miller (1996) where an explicit generative (semantic) grammar is used to build a probabilistic model for the parser, our approach does not involve the use of such a grammar (which may not be readily available for our domains). A different probabilistic model that does not assume the use of such a grammar is developed. The model will be described followed by a discussion of probability estimation.

4.1 The Parsing Model

Before we proceed to describe the statistical parsing model we are going to build, let's first present some definitions essential to the discussion.

A *parser* is a relation $Parser \subseteq Sentences \times Queries$ where $Sentences$ and $Queries$ are the sets of natural language sentences and logical queries respectively. Given a sentence l, the set $Meaning(l) = \{q \in Queries \mid \langle l, q\rangle \in Parser\}$ is the set of logical queries which are possible interpretations of the given sentence l. Our task here is to learn a semantic parser that implements the target relation $Parser$.

A *parse state* is a tuple $\langle stack, buffer\rangle$ where *stack* is a list of *lexicalized*[20] logical forms (the meaning of a lexical entry) and the *buffer* is a list of words from the input sentence. The set $States$ is the set of all syntactically well-formed parse states. A *parsing action* a_i (which is the ith action of our learned parser and $i \geq 1$) is a function $a_i(s) : InStates_i \rightarrow OutStates_i$ where $InStates_i \subseteq States$ is the set of states to which the action is applicable and $OutStates_i \subseteq States$ is the set of states constructed by the action (given the input states). The function $a_0(l) : Sentence \rightarrow InitialStates$ maps each sentence l to a corresponding unique *initial* parse state in $InitialStates \subseteq States$ which is the starting state for our learned parser given l. Suppose our learned parser has n total parsing actions, the *partial* function $a_{n+1}(s) : FinalStates \rightarrow Queries$ is defined as the action that retrieves the logical query from the final state $s \in FinalStates \subseteq States$ if one exists. Right now, we assume that the learned parser is complete with respect to the data; there is always a path from the sentence l to one of the logical queries in $Meaning(l)$. This assumption, however, is not true. Since the learned parser is derived from the training data, it could suffer from missing certain parsing actions which are essential to completing a parse if they are not part of the training data. (This problem is further discussed in Section 6.) Hence, some final states may not "contain" a logical query and therefore the function a_{n+1} is only a partial function. In cases where this actually happens, the parser would simply report a failure. A state is called a *final* state if there exists no parsing action applicable to it. A *path* is a finite sequence of parsing actions.

Given a sentence l, a *good* state s is one such that there exists a path from it to a logical query $q \in Meaning(l)$. Otherwise, it is a *bad* state. Since we lexicalize our parse states, the input sentence l can be constructed from the corresponding given parse state s. Therefore, the set of parse states can be uniquely divided into the set of good states and the set of bad states given l and $Parser$. In other words, we have, $States = States^{+} \cup States^{-}$ where $States^{+}$ denotes the set of good states and $States^{-}$ denotes the set of bad states. Now, we

[20]We lexicalized logical forms by attaching each logical form on the stack with a buffer that contains the words from the input sentence that cuased this form to be introduced. See Section 2.3.2.

are ready to discuss our parsing model.

Given an input sentence l, the goal of the learned parser is to search for a logical query $\hat{q}$ such that the probability of $\hat{q}$ being in $\mathit{Meaning}(l)$ is maximized. In other words, we want to find $\hat{q}$ given l such that

$$\hat{q} = \mathrm{argmax}_q \; P(q \in \mathit{Meaning}(l) \mid l \Rightarrow q) \tag{12}$$

given that there *is* a path from l to q (denoted as $l \Rightarrow q$). Instead of building a probabilistic model for how likely a logical query would be generated given a sentence (as in most cases in statistical syntactic parsing), we build a model to estimate how likely a parse from a given sentence would represent a correct meaning for it.

Now, we need to define what $P(q \in \mathit{Meaning}(l) \mid l \Rightarrow q)$ is. First, we notice that

$$P(q \in \mathit{Meaning}(l) \mid l \Rightarrow q) = P(s \in \mathit{FinalStates}^+ \mid l \Rightarrow s \text{ and } a_{n+1}(s) = q) \tag{13}$$

where $\mathit{FinalStates}^+ = \mathit{FinalStates} \cap \mathit{States}^+$ (the set of good final states). We can drop the conditions and denote the above probabilities as $P(q \in \mathit{Meaning}(l))$ and $P(s \in \mathit{FinalStates}^+)$ respectively as long as they are assumed in the context of the discussion. This is to say that the probability that a given logical query (as a result of parsing l) is the meaning for l is the same as the probability that the final state (from which the logical query can be retrieved) is a good state (by the definition of a *good* state). More precisely, given a final state s produced by parsing l, and $q = a_{n+1}(s)$, we have

$$s \in \mathit{FinalStates}^+ \Leftrightarrow q \in \mathit{Meaning(l)}. \tag{14}$$

Obviously, we need to determine in general the probability of having a good resulting parse state, or more precisely, $P(s_{j+1} \in \mathit{OutStates}_i^+)$ for a parse state s_{j+1} and an action a_i where $\mathit{OutStates}_i^+ = \mathit{OutStates}_i \cap \mathit{States}^+$ (the set of good output states). Given any parse state s_j at the jth step of derivation (a derivation is a path from l to q) and a parsing action a_i such that $s_{j+1} = a_i(s_j)$ (i.e. the action is applicable to s_j producing the next state s_{j+1}), we have

$$\begin{aligned} P(s_{j+1} \in \mathit{OutStates}_i^+) \;\; = & \\ & P(s_{j+1} \in \mathit{OutStates}_i^+ \mid s_j \in \mathit{InStates}_i^+) P(s_j \in \mathit{InStates}_i^+) + \\ & P(s_{j+1} \in \mathit{OutStates}_i^+ \mid s_j \notin \mathit{InStates}_i^+) P(s_j \notin \mathit{InStates}_i^+) \end{aligned} \tag{15}$$

where $\mathit{InStates}_i^+ = \mathit{InStates}_i \cap \mathit{States}^+$ (the set of good input states to the action a_i). So, the meaning of the equation is that the probability of a certain parse state being a good one is the sum of the probability that assuming the *previous* parse state is a good one, the application of the action would produce a good next parse state and the probability that assuming it is a bad state, the parsing action would produce a good one, with these probabilities being weighted by the probabilities that the previous parse state is a good one or a bad one respectively. Since by definition no parsing action could produce a good parse state from a bad one, the second term is simply zero and we have

$$P(s_{j+1} \in \mathit{OutStates}_i^+) = P(s_{j+1} \in \mathit{OutStates}_i^+ \mid s_j \in \mathit{InStates}_i^+) P(s_j \in \mathit{InStates}_i^+). \tag{16}$$

Now, we are ready to derive a method for estimating $P(q \in \mathit{Meaning}(l))$. Suppose $q = a_{n+1}(s_m)$ (i.e. s_m is the final state), we have

$$\begin{aligned}
&P(q \in \mathit{Meaning}(l)) \quad (17)\\
&\quad = P(s_m \in \mathit{FinalStates}^+)\\
&\quad = P(s_m \in \mathit{FinalStates}^+ \mid s_{m-1} \in \mathit{InStates}^+_{a_{m-1}}) P(s_{m-1} \in \mathit{InStates}^+_{a_{m-1}})\\
&\quad \ldots\\
&\quad = P(s_m \in \mathit{FinalStates}^+ \mid s_{m-1} \in \mathit{InStates}^+_{a_{m-1}}) \ldots\\
&\qquad P(s_j \in \mathit{OutStates}^+_{a_{j-1}} \mid s_{j-1} \in \mathit{InStates}^+_{a_{j-1}}) \ldots\\
&\qquad P(s_2 \in \mathit{OutStates}^+_{a_1} \mid s_1 \in \mathit{InStates}^+_{a_1}) P(s_1 \in \mathit{InStates}^+_{a_1})
\end{aligned}$$

where a_k denotes the index of which action is applied at the kth step. Since the initial state is always a good state, we know that

$$P(s_1 \in \mathit{InStates}^+_{a_1}) = 1. \quad (18)$$

So, now we have

$$P(q \in \mathit{Meaning}(l)) = \prod_{j=1}^{m-1} P(s_{j+1} \in \mathit{OutStates}^+_{a_j} \mid s_j \in \mathit{InStates}^+_{a_j}). \quad (19)$$

We will proceed to discuss how to estimate the probability of $P(a_i(s) \in \mathit{OutStates}^+_i \mid s \in \mathit{InStates}^+_i)$ given an action a_i and a parse state s.

4.2 Estimating Probabilities for Parsing Actions

Since in CHILL, the collection of positive and negative examples for learning a control rule for each parsing action facilitates learning a decision list for the set of parsing actions, the probabilities of the parsing actions are, therefore, estimated according to such a model. Alternatively, one could collect negative examples from all the parsing actions and treat the parser as a set of independent parsing actions instead of a decision list. This issue will be further discussed in Section 6. Although the current method may not be the best one, its resemblance to that of learning a decision list makes a direct comparison of the performance of the two approaches of parsing more intuitive since the learner in each system is given exactly the same set of positive and negative examples. Since there will be no learning for the last parsing action a_n, in a decision list (as there will be no negative examples collected for it), we will treat the probability estimation for it differently. Let's begin with a discussion on how probabilities are estimated for the actions $a_1, \ldots, a_{n-1}$.

The result of learning control rules for the parser is that each parsing action has learned a set of hypotheses which serve to identify the conditions under which that parsing action should be applied. These hypotheses can be viewed as the contextual information that would be useful for making the decision of whether to apply the action or not. Some hypotheses may be more "important" than others in the sense that they carry more weight in the decision. The committee of hypotheses, like a set of features, can be used to estimate the probabilities of the goodness of various actions given a parse state. A weighting parameter (or a priority weight) is also included to lower the probability estimate depending on the position of the action a_i in the decision list of actions applicable to the given parse state; the further down an action is in a decision list, the less likely it will be applied. For actions a_i where $1 \leq i \leq n-1$:

$$P(a_i(s) \in OutStates_i^+ \mid s \in InStates_i^+) = \mu^{pos(i)-1} \sum_{h_k \in H} \lambda_k P(a_i(s) \in OutStates_i^+ \mid h_k) \quad (20)$$

where s is a given parse state, $pos(i)$ is the position of the action a_i in the list of actions applicable to state s, λ_k and $0 < \mu \leq 1$ are weighting parameters , and $H \subseteq Q$ is the chosen subset[21] of hypotheses learned by the induction algorithm for the action a_i. The sum of λ_k for all k is equal to one. In other words, the probability of an action is estimated by linearly combining the conditional probability estimates of each of the learned models and then weighting by an exponential decay factor depending on the position of the action in the decision list. We will further discuss how λ_k and $P(a_i(s) \in OutStates_i^+ \mid h_k)$ are determined. Now, we are ready discuss the probability model for the action a_n.

To build the probability model for the last action a_n, we could use the maximum likelihood estimate since we have a relatively large number of examples.[22] However, using just the ML estimate directly would mean we are ignoring *any* contextual information that might be available (particularly since there will be no learning for the last action). Recall that equation (20) will yield an estimate for the goodness of each action a_1 to a_{n-1}. We devise a simple test that checks if the maximum of the set of estimates $A(s)$ obtained from the subset of the actions $a_1, \ldots, a_{n-1}$ applicable to s is less than or equal to a certain threshold α. The intuition is that the goodness of the last action is conditioned on whether the best estimate of any previously applicable action is no greater than a certain threshold. If $A(s)$ is empty, we just take the maximum as zero. More precisely,

$$P(a_n(s) \in OutStates_n^+ \mid s \in InStates_n^+) = \begin{cases} \frac{c(a_n(s) \in OutStates_n^+)}{c(s \in InStates_n^+)} & \text{if } max(A(s)) \leq \alpha \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

where α is the threshold[23], $c(a_n(s) \in OutStates_n^+)$ is the count of the number of good states produced by the last action, and $c(s \in InStates_n^+)$ is the count of the number of good states to which the last action is applicable.

Now, let's discuss how $P(a_i(s) \in OutStates_i^+ \mid h_k)$ and λ are estimated. Suppose that D is the set of positive and negative examples[24] collected for an action a_i. Since the set of training data may have noise, we need to include this factor in our estimation. Noisy data will only appear in the set of negative examples, if any. Positive examples cannot be noisy since they are collected from example analysis and if a parsing operator is wrongly applied to a certain parse state, a correct parse has no chance of being discovered, and therefore it

[21] We require that the cardinality of H should not be more than that of the entire set of positive examples.

[22] The typical number of examples for other actions is usually within the range one to a few hundred. However, the number of examples for the last action could be around a few thousand.

[23] The threshold is set to 0.5 for all the experiments performed in the next section.

[24] The set of negative examples used to induce the hypotheses are collected from parsing operators *below* the current operator in the decision list. However, the set of negative examples used here for probability estimation include those *above* the current operator. This is done primarily for the reason of accuracy of estimation.

will not be considered a positive example of the operator. The *degree* of noise assumed is given by the parameter β stated in equation (11). We need to first compute the *probability* of having noisy negative examples given β and a particular set of examples D. If we think of β as the lower bound on the degree of "noise freeness" in D, then

$$\hat{n_c} = \frac{p(1-\beta)}{1+\beta} \tag{22}$$

is an estimate on the number of negative examples allowed to be covered by a clause where p is the number of positive examples in D. Now, if we define

$$\theta = \begin{cases} \frac{\hat{n_c}}{n} & \text{if } \hat{n_c} \leq n \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

where n is the number of negative examples in D, then θ is an estimate of the probability that a negative example is corrupted. Now, if $h_k \models s$ (i.e. h_k labels s as a good state for this operator), we have

$$P(a_i(s) \in \mathit{OutStates}_i^+ \mid h_k) = \frac{p_c + \theta \cdot n_c}{p_c + n_c} \tag{24}$$

where p_c is the number of positive examples covered by h_k and n_c the number of negative examples covered. Otherwise, if $h_k \not\models s$, we have

$$P(a_i(s) \in \mathit{OutStates}_i^+ \mid h_k) = \frac{p_u + \theta \cdot n_u}{p_u + n_u} \tag{25}$$

where p_u is the number of positive examples rejected by h_k and n_u the number of negative examples rejected.

One could use a variety of linear combination methods to estimate the weights λ_k. However, we have a taken a very simple approach here. λ_k is determined by the complexity of the hypothesis h_k:

$$\lambda_k = \frac{size(h_k)^{-1}}{\sum_{i=1}^{|H|} size(h_k)^{-1}}. \tag{26}$$

Intuitively, it means the more complex an hypothesis is, the less probable it is and therefore the less weight it carries in a combined decision. We will discuss other linear combination methods in Section 6.

4.3 Searching for a Parse

To find the most probably correct parse, the parser employs a beam search starting with the initial parse state. This is similar to using a beam search to find the most probable parse tree in probabilistic syntactic parsing (Ratnaparkhi, 1999; Collins, 1996). At each step of the derivation, the parser finds all the parsing actions that are applicable to the parse state and calculates the probability of correctness of each of them using equations (20) and (21). It then computes the probability of the correctness of each derivation up to that point using equation (19). The search queue is sorted by the probability of each derivation. If a parse state cannot be further expanded (i.e. the parser cannot find any applicable action to the

Procedure *ProbParse*
Input:
s: the input sentence
B: the beam size
Q: the parse queue
Ops: the set of parsing actions
Output:
$Query$: the resulted logical query

S_0 := the initial parse state created from input sentence s
$Q := [S_0]$
While $Head(Q)$ does not contain a query **Do**
    For each parse state $S_i \in Q$ **Do**
        Find the sublist of parsing action A_i from Ops which are applicable to S_i
        If $A_i \neq \emptyset$ **Then**
            Expand S_i by each action in A_i to get a list of children parse states in Q_i
        Else
            If S_i contains a query **Then**
                $Q_i := [S_i]$
            Else
                $Q_i := \emptyset$
            End If
        End If
    End For
    Q := the best B parse states from $\bigcup_i Q_i$
End While
$Query$:= the query contained in $Head(Q)$
Return $Query$
End Procedure

Figure 8: The parsing algorithm using a beam search

parse state), the parser checks if a logical query can be obtained from the parse state. If so, the parse state remains in the parse queue. Otherwise, it is removed. The parser stops when a complete parse is found on the top of the parse queue or a certain time limit is passed (in which case a failure is reported by the parser). A summary of the parsing algorithm is shown in Figure 8 (assuming a parse can always be found).

5 Experimental Results

5.1 The Domains

Three different domains are used for demonstrating the performance of the new approach. The first one is the United States Geography domain. The database contains about 800 facts implemented in Prolog as relational tables containing basic information about the U.S. states like population, area, capital city, neighboring states, major rivers, major cities, and highest and lowest points and their elevation. It also contains information about rivers

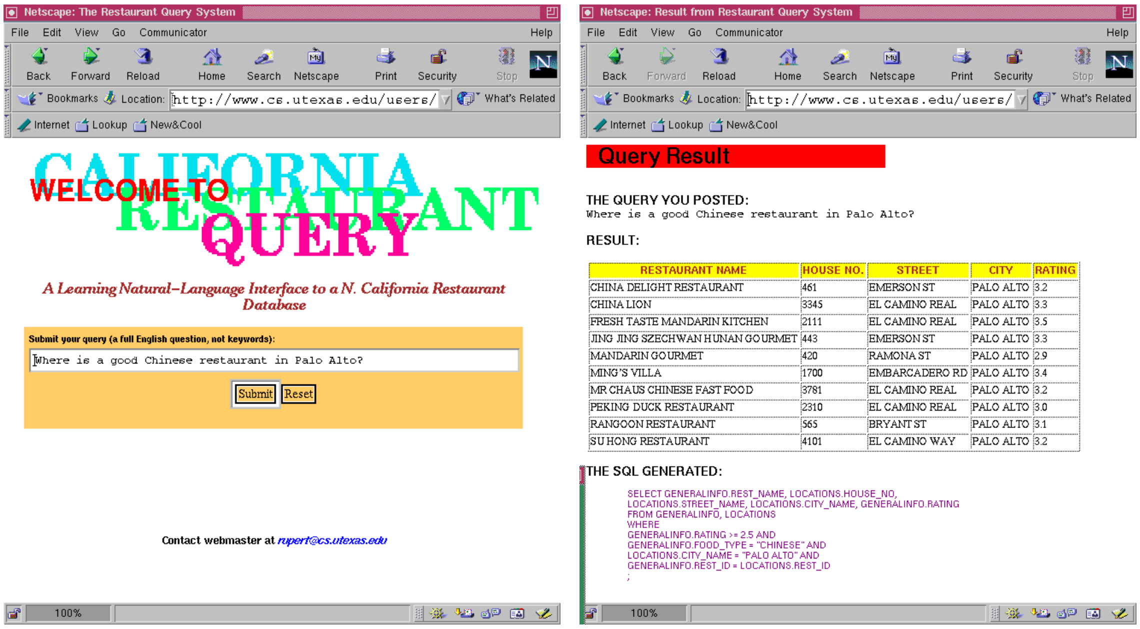

Figure 9: Screenshots of a user's request in English

and the population of cities. A handcrafted parser called Geobase was constructed for the domain by people who built the database. We used this parser in our experiments for the comparison of results.

The second application is a restaurant database query system. The database contains information about restaurants in the Northern California including the Bay area. The types of information in the database are the name of the restaurant, its location, the specialty of the restaurant and a rating on the quality of the restaurant given by customers. The database currently contains more than nineteen thousand entries (or rows in a relational table). The purpose of the application is to provide tourists with restaurant information like many of the online dinning guides on the Web. While most of them require a user to select entries (from a possibly large tables) or enter them in some kind of HTML forms to access the information in their databases, we have provided a natural language interface to the database so that a user can ask questions in English. Sample screenshots of the Web interface developed for the system is shown in Figure 9.

The third domain consists of a set of 300 computer-related job postings, such as job announcements, from the USENET newsgroup `austin.jobs`. Information from these job postings are extracted to create a database (Califf & Mooney, 1999) which contains the following types of information: 1) the job title, 2) the company, 3) the recruiter, 4) the location, 5) the salary, 6) the languages and platforms used, and 7) required or desired years of experience and degrees. This database is updated on a daily basis and it currently has several thousand entries.

5.2 Experimental Design

The U.S. Geography domain has a corpus of 560 sentences. Approximately 100 sentences were collected from user requests[25] on the Web for the first few months[26] since the system was put into use and the rest were collected from undergraduate students in our department. To compare the new system to the original one, we included the test result for the subset of 250 sentences which were used in (Zelle & Mooney, 1996) as well. The restaurant database query system has a corpus of 250 sentences and they are all artificially generated by hand (and partly based on user requests). The job database information system has a corpus of 400 sentences. Again, they are all artificially made with the help of a simple grammar that generates certain obvious types of questions people may ask.

The experiments were conducted using 10-fold cross validation. The corpus is divided up into ten partitions of equal size, each trial uses a particular partition as the test data and the rest of the partitions are used for training. The test result is the average of the results of the ten trials. In each test, the recall (a.k.a. accuracy) and the precision of the parser are reported. Recall and precision are defined as follows:

$$\mathit{Recall} = \frac{\#\text{ of correct queries produced}}{\#\text{ of sentences}} \tag{27}$$

$$\mathit{Precision} = \frac{\#\text{ of correct queries produced}}{\#\text{ of successful parses}}. \tag{28}$$

The recall is the number of correct queries produced divided by the total number of sentences in the test set. The precision is the number of correct queries produced divided by the number of sentences in the test set from which the parser produced a query (i.e. a successful parse). A query is considered correct if it produces the same answer set as that of the correct logical query.

5.3 Results

A beam size of five and a window size of two were used by the TABULATE algorithm for all the experiments. The results of running the 10-fold cross validation test on the corpora are shown in Table 6. Different beam sizes were also used by the probabilistic parser to illustrate their effects on the performance of the parser.

In the experiment with 250 sentences from the U.S. Geography domain, we reported the following results: 1) the recall (and precision) of CHILL using TABULATE (only the best hypothesis from the committee of hypotheses was used as the control rule for a parsing action), 2) the recall (and precision) of CHILL with probabilistic parsing (ProbParse) using different beam sizes, 3) the recall of the original CHILL (which uses the CHILLIN induction algorithm), and 4) the recall of Geobase which is a hand-crafted parser built for the domain.

The best recall and precision of CHILL with probabilistic parsing were 80.40% and 88.16% respectively using a beam size of twelve while that of CHILL using TABULATE were 75.60% and 92.65% respectively. The performance of the probabilistic parser degraded when decreasing the beam size. The drop in performance, however, was not rapid except

[25] We have a log of all online user requests. The current total amounts to nearly four thousand.

[26] The system has been deployed for almost two years.

	Geo250		Geo560		Jobs400		Rest250	
	R	P	R	P	R	P	R	P
CHILL+ProbParse(12)	80.40	88.16	71.61	78.94	80.50	86.56	99.20	99.60
CHILL+ProbParse(8)	79.60	86.90	71.07	79.76	78.75	86.54	99.20	99.60
CHILL+ProbParse(5)	78.40	87.11	70.00	79.51	74.25	86.59	99.20	99.60
CHILL+ProbParse(3)	77.60	87.39	69.11	79.30	70.50	87.31	99.20	99.60
CHILL+ProbParse(1)	67.60	90.37	62.86	82.05	34.25	85.63	99.20	99.60
CHILL+TABULATE	75.60	92.65	69.29	89.81	68.50	87.54	99.20	99.60
CHILL+CHILLIN	68.50	–	–	–	–	–	–	–
Handcrafted Parser	56.00	–	–	–	–	–	–	–

Table 6: Results on all the experiments. Geo250 consists of 250 sentence from the U.S. Geography domain. Geo560 consists of 560 sentences from the same domain. Jobs400 consists of 400 sentences from the job postings domain. Rest250 consists of 250 sentences from the Northern California restaurant domain. R = Recall and P = Precision. ProbParse(B) is the probabilistic parser using a beam size of B. The handcrafted parser used for the U.S. Geography domain is Geobase.

for the case when the beam size was one. The recall of the original system using CHILLIN was around 68.5% and that of Geobase was 56.0%.

In the experiment with 560 sentences from the same domain, the same set of results were reported except those of the original CHILL and the Geobase. The best recall and precision of CHILL with probabilistic parsing were 71.60% and 78.94% respectively using a beam size of twelve while that of CHILL using TABULATE were 69.29% and 89.81% respectively. Again, the drop in performance of the probabilistic parser due to decreasing the beam size was not rapid (around -2% for -9 in the beam size) except when the beam size was one.

The same set of results were reported for the experiment with 400 sentences from the USENET newsgroup job postings domain. The best recall and precision of CHILL with probabilistic parsing were 80.50% and 86.56% respectively while that of CHILL using TABULATE were 68.50% and 87.54% respectively. This time, however, the drop in performance of the probabilistic parser using a smaller beam size than twelve was very significant (around -10% for -9 in the beam size and around -50% for -11 in size).

In the last experiment with the 250 sentence corpus from the Northern California restaurant domain, the best recall and precision of CHILL with probabilistic parsing were 99.20% and 99.60% respectively which is the same as that of CHILL using TABULATE. Figure 10 and Figure 11 show the performance of the probabilistic parser using various beam sizes in the different domains.

5.4 Discussion

Before we proceed to discuss the results, let's consider some of the potential advantages and disadvantages of the present approach. Then, we can discuss the results with an understanding on the conditions under which the system would perform well.

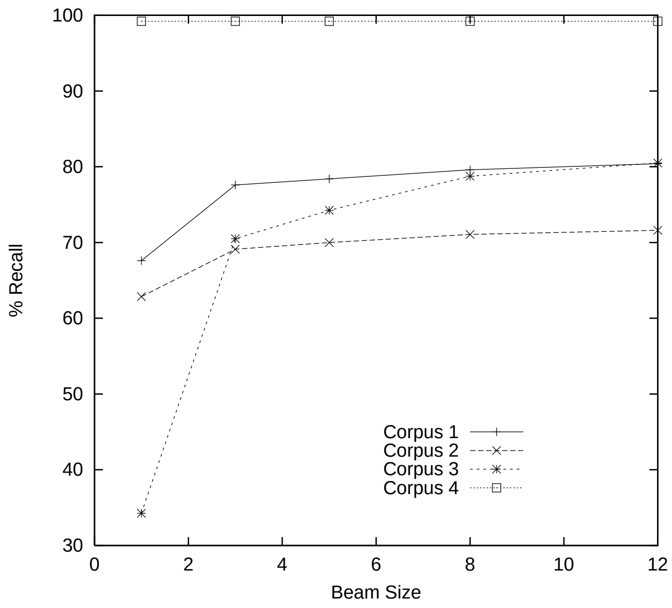

Figure 10: The recall of the parser using various beam sizes in the different domains

First, the present approach is largely motivated by the observation that the original system is very brittle; it takes only one overly general (or specialized) control rule to miss a correct parse. The use of a probabilistic approach in parsing adds a layer of robustness to the system by considering a set of parses where the overall quality of a parse is considered. This allows room for making some mistakes during the course of parsing a sentence because of the presence of imperfect control rules. The probabilistic parser will perform better than the original approach if the correct parse has the best overall quality (i.e. the most probable) and the beam size is big enough to overcome local minima which are places where the parser makes mistakes. The second advantage is the use of a probabilistic classifier (by combining multiple learned models). Using a committee of hypotheses can usually perform better than using just a single hypothesis. This has been demonstrated in other domains as well.

While a probabilistic approach has potential advantages in the robustness of the system, it relies on the precision of probability estimation. If there is a lack of data in a domain, it could be hard to achieve a certain level of accuracy required for good performance of the system and very often this means replying on some default models may be necessary. A second potential disadvantage would be the decrease in precision since considering a set of parses would most likely lead to outputing more spurious parses by the system. A third potential disadvantage would be the amount of time needed to successfully find a parse.

Now, let's consider the general performance of the different systems in all the domains. The experimental results from the different domains demonstrated that the system did gain robustness because of considering a set of parses; the performance of the system increased monotonically with the beam size in most of the domains. It was most apparent in the third experiment (400 sentences from the job postings domain) where the system signifi-

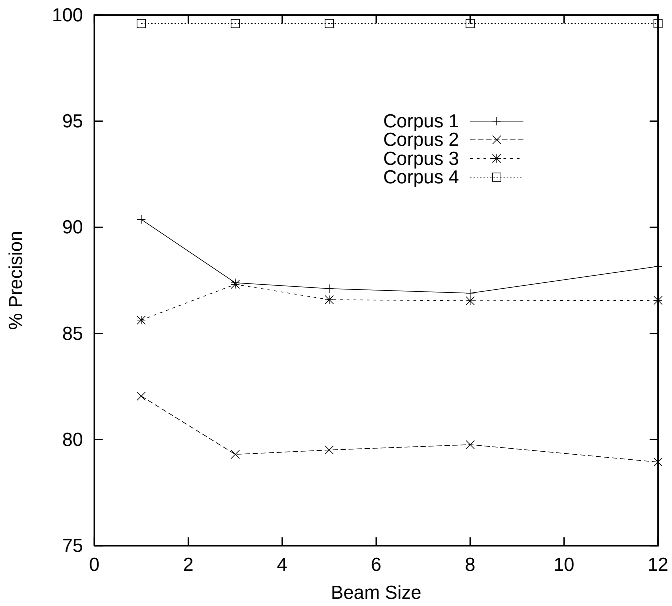

Figure 11: The precision of the parser using various beam sizes in the different domains

cantly outperformed the original system. However, it was also the case that there might be a lack of data (at least in some of the domains) since using a beam size of one (which rendered the search effectively a best first search like that of using a decision list) would lead to significantly worse performance than the original system; the probabilistic classifier alone was not performing better than that of a decision list. Precision of the system somewhat decreased but not by a significant amount except in the second experiment. All the experiments were run on a 167MHz UltraSparc work station in Sicstus Prolog. Although results on the parsing time of the different systems were not formally reported here, it was noted that the difference between using a beam size of three and the original system was less than two seconds (i.e. the average amount of time required to find a successful parse) in all the domains but significantly increased to around twenty seconds when using a beam size of twelve.

While there was an overall improvement using the new approach (as far as recall is concerned), its performance varied greatly from one domain to another. In the second experiment, the best improvement in recall was merely around 2%. Using the same beam size, however, the system had a +10% improvement in recall over the original system. This suggests that there maybe other factors in the picture specific to a particular domain. Currently, it is not certain that what these factors might be but it is suspected that factors like the quality of the lexicon used by the system, the relative amount of data available for calculating probability estimates, and the problem of "learning an incomplete parser" with respect to test data might all be potential bottlenecks in the picture. Finally, the performance of the two systems were very close to each other in the last domain and their performance were near perfect in both the recall and the precision. This is primarily because the domain was relatively easier; the systems achieved 90+% in recall and precision given only roughly 30% of the training data. (These results on training the systems with various amount of data are not shown here since they have not been collected for all the domains.)

The recall of CHILL using TABULATE was significantly better than that of using CHILLIN. This shows that the use of background knowledge with theory constants and beam searching the hypothesis space builds better control rules for the parsing actions. Our experience is that (by inspecting the learned control rules) hypotheses discovered by TABULATE tend to be more general and readable than those of CHILLIN (given the same set of positive and negative examples). Finally, Geobase performed the worst among the results which was largely due to hand-crafting rules that do not scale up with the task.

6 Future Work

There are several areas where we would like to look into as possible directions for future work. Each of these areas will be discussed here. More precisely, we would like to explore the first eight listed areas below and accomplish them as part of the future work of this proposal. The last two suggested areas for future work are interesting directions for a more long term goal of the project.

6.1 Performing More Extensive Experiments

Currently, the biggest corpus we have experimented with is the one in the U.S. Geography domain with 560 sentences. It is interesting to expand some of the corpora in the different attempted domains and see if the approach would scale up well with the tasks. In all the domains, we have implemented online demos accessible through the World Wide Web to collect real world data. For instance, in the Geography domain, the NLI system has been deployed for more than two years and more than 4,000 requests to the database have been made so far.

6.2 Dealing with Incompleteness

The assumption that the parser is *complete* with respect to unseen data (which means there is at least one correct parse for each input sentence that can be parsed into a query given the parser) simply does not hold. In other words, there are cases where the parser just cannot proceed because the “right” parsing operator is missing. This is due in large part to the fact that the parser is acquired from training examples; only parsing operators used to parse the training examples are learned but there may be parsing operators required to parse the test cases that are not used for parsing the training data. One way to solve this problem is to allow the parser to “invent” parsing operators when it comes to an impasse due to missing the right operators by inspecting the current parse state. More precisely, we can invent operators for COREF_VARS, DROP_CONJ, and LIFT_CONJ by checking if the remaining predicates on the parse stack can be further processed. For instance, if there is a predicate on the parse stack that can be dropped into the body of a meta predicate, we can try dropping it and see if a correct query would be produced. We cannot invent operators for INTRODUCE since we do not have the target logical query for the sentence. We do not need to invent operators for SHIFT.

The estimation of probabilities for these "invented" parsing operators, however, are different from the current approach since it cannot be based on any training data at all. One potential method of estimating their probabilities is by estimating the probability that given, for instance, a DROP_CONJ operator in the existing parser, the application of the operator to a parse state would give a good resulting parse state. This method is similar to estimating the probability of a grammar rule in a probabilistic context-free grammar.

We incorporated the ability to invent operators in the existing parser used by CHILL. Preliminary results of the different domains were around 8% to 10% improvement in recall and precision. (The control rules were learned using TABULATE.)

6.3 Improving Precision

Second, we would like to look into ways to improve the precision of the parser. One possible way is to implement a test on the resulted parse queue and see if the probability of "best" parse is significantly higher than, say, the second parse. A set of parses with similar probabilities could suggest that no correct parse has been found (or perhaps the question is truly ambiguous). Problems with parsing precision can also be partially overcome by tackling the problem of parser incompleteness since improving the recall of a parser (significantly) would probably improve the precision as well.

Another possible avenue is to consider mapping all queries found by the parser back to English (which can be done by reversing the mapping of the lexicon and inspecting the structure of the logical query) and let the user pick the right interpretation (if any) in cases where the best query found is not significantly better than the second query. This potentially allows online learning from user input which will be further discussed in Section 6.8 below.

6.4 Exploring Various Linear Weighted Combination Methods

Various methods for estimating the weights in a linear evidence combination model have been developed: using a EM algorithm for iterative weight updating, Bayesian combination (Buntine, 1990), and Likelihood combination (Duda, Gaschnig, & Hart, 1979). In particular, we would like to explore the Bayesian combination method since it is closest to the optimal Bayes classification. More precisely, the method estimates the probability of a class c given an example x, the training data D, and the set of hypotheses H produced by the learning algorithm by:

$$P(c \mid x, H) = \sum_{h_k \in H} P(c \mid x, h_k) P(h_k \mid D) \tag{29}$$

$P(c \mid x, h_k)$ can be estimated according to equations (24) and (25), or better, using appropriate smoothing techniques when there is a lack of training data which will be further discussed below. The estimation of $P(h_k \mid D)$ is not going to be discussed here since it involves a heavy amount of details. It would also be interesting to improve the existing method so that it would be tolerant to the presence of noise.

6.5 Improving Probability Estimation Using Smoothing Techniques

Currently, some conditional probabilities are overestimated (e.g. the results are too close to 1.0) for poorly learned models resulting in worse classification performance. The m-estimate

(Cestnik, 1990) method of smoothing the ML estimates for the conditional probabilities is not entirely satisfying due to the extreme lack of data in some cases. New smoothing techniques that base on *a priori* assumptions about the quality of the learned models may need to be developed. One potential method is to linearly interpolate an additional model (Stanley & Goodman, 1996; Jelinek & Mercer, 1980):

$$P_{interpolated}(c \mid h_k) = \lambda P(c \mid h_k) + (1 - \lambda) Q(h_k) \tag{30}$$

where c is a class, h_k is a hypothesis in the set of hypotheses produced by the learning algorithm, $0 \leq \lambda \leq 1$ is a parameter, $P_{interpolated}(\cdot \mid \cdot)$ is the interpolated probability estimate, $P(\cdot \mid \cdot)$ is the ML estimate, and $Q(h_k)$ represents a default model of h_k, for instance, the *a priori* probability of h_k. So, if h_k is a poorly learned model, $Q(h_k)$ would be a smaller value than the ML estimate. Therefore, the interpolated estimate is less than the (overestimated) ML estimate.

6.6 Abandoning The Decision List Framework

The current approach is modelling the semantic parser as a decision list. Negative examples for a parsing action are collected only from parsing actions further down in the list, an exponential decay factor is used to model the *a priori* probability of a parsing action depending on its position in the decision list, and an explicit model for the last parsing action is built to improve performance. However, such a restriction on modelling a decision list is not necessary. A new way of modelling the semantic parser is to consider the parsing actions as a *set* instead of a decision list, and negative examples will be collected from all other parsing actions. The potential benefit of this new way of collecting examples is that the learning algorithm can learn a more discriminative set of hypotheses. In the current approach, parsing actions that are closer to the end of the decision list have fewer training examples and therefore are more prone to poor probability estimation and very often suffer from no learning at all. However, in practice, there may be a lack of training data for some parsing actions, and a more realistic approach would be combining decision list modelling with this new framework since some parsing actions may not have any negative examples in the training data (and therefore no learning is possible).

6.7 Meta Learning

While there is a lack of data for some parsing actions, learning the default model for the last parsing action SHIFT is not an easy task, however, since one normally can have up to several thousand training examples for the last parsing action. This suggests that directly learning models that capture contextual information for these great number of parse states may not be a very effective approach to the problem. However, if one considers learning models that capture the results of the decisions made by other parsing actions, a potentially much more compact set of hypotheses can be learned for the default parsing action. This is similar to meta learning in the sense that a new set of examples have to be generated for the last parsing action by including information on parsing decisions made by other parsing actions.

6.8 Online Learning

The idea of online learning is that given that the parser produces a set of queries and the correct one is somewhere in the set but not the best one (i.e. not the most probably correct one) and that the user can give feedback to the system, for instance, by picking out the right one for the system from the set of queries, the parser can learn online with this new piece of information by appropriately adjusting the weights of the multiple learned models for some subset of parsing actions. More precisely, suppose S is the set of queries found by parsing the sentence l, q is the best query in S, q^* is the correct query in S, and the following are the derivations of $l \Rightarrow q$ and $l \Rightarrow q^*$ respectively:

$$l \to s_1 \cdots s_k \to x_1 \cdots x_n \to q$$
$$l \to s_1 \cdots s_k \to y_1 \cdots y_m \to q^*.$$

A *transition* from state a to b, $a \to b$, represents a single step in a derivation. We will use the term *transition probability* in this context to represent the conditional probability $P(a^+ \mid b^+)$ of the transition from state a to b where s^+ represents that the state s is a good state. Since we know that $P(q^* \in \mathit{Meaning}(l)) < P(q \in \mathit{Meaning}(l))$, some of the transition probabilities in the derivation $l \Rightarrow q$ are overestimated, or some of the transition probabilities in the derivation $l \Rightarrow q^*$ are underestimated. We assume that both cases occur here. Let X be the set of transitions whose probabilities are overestimated in the derivation $l \Rightarrow q$ and Y be the set of transitions whose probabilities are underestimated in the derivation $l \Rightarrow q^*$. The goal of the learning is to reduce the classification errors of the classifiers that produce the probabilities from the sets X and Y. Before we proceed to discuss how this may be achieved, we need to first discuss how the sets X and Y are determined.

Since all the transitions from s_1 up to s_k are shared by both derivations, they will not be part of X or Y. (Altering their probabilities will not make $P(q^* \in \mathit{Meaning}(l)) > P(q \in \mathit{Meaning}(l))$.) Let's consider the derivation $l \Rightarrow q$. We know that the states s_1 to s_k are all good states and the rest are bad states. Since s_k is a good state and x_1 is a bad state, the probability, $P(x_1^+ \mid s_k^+)$, of the transition $s_k \to x_1$ is overestimated. (In fact, it should be zero.) The set X simply consists of this transition. The other transitions are not included in X since we are not certain if their probabilities are overestimated. The set Y can be determined heuristically by removing a transition one at a time from s_k to y_m and checking if the product of the rest of the transition probabilities is now greater than $P(q \in \mathit{Meaning}(l))$. Transitions with lower probabilities are considered first. The goal is to find a minimal set of transitions whose probability estimates if improved would yield the most improvement to the estimation of $P(q^* \in \mathit{Meaning}(l))$. Now, we are ready to discuss how the classifiers that produce these probability estimates can be improved.

We will restrict our discussion to the case where the transition in the set X or Y is *not* done by a SHIFT operator. (The case where a SHIFT operator is involved will be a little more complicated.) For any transition in the set X, since the corresponding classifier produces an overestimation of the transition probability, we need to adjust some of the parameters of the classifier so that this probability can be lowered. Similary, we need to adjust the parameters of the classifiers for the set Y so that the probability estimates of the transitions are increased. More precisely, suppose the classifier has the set of models h_i with the set of weights λ_i. We can change the weights λ_i so that the probability estimates are maximized (or minimized). However, to avoid overfitting one single training example, one needs to define a *learning parameter* (similar to that of a perceptron learning algorithm) and update the weights according to the amount of errors made. In this case, we can assume that the target is one if the goal is to maximize the probability estimates; otherwise, the target is zero. However, we will perform only one step of the computation since we do not want the weight updating procedure to overfit a single training example. This is similar to performing one step of weight updating on one particular misclassified example in a perceptron learning algorithm (or other linear weight updating algorithms).

6.9 Probabilistic Relational Models

Recently, there has been research work on learning *probabilistic relational models* (PRMs) for databases (Friedman, Getoor, Koller, & Pfeffer, 1999) which generalizes the traditional approach of learning Bayesian networks (Pearl, 1988). Like the learning of Bayesian networks, a PRM learns a structured representation of the probabilistic model; structural dependency between attributes (or random variables) are captured and a complex probability distribution is built over them. However, in PRM learning, one can go beyond the bound of a propositional (or attribute-value based) representation and build the learning framework on a first-order representation which captures relational knowledge present in the domain. We would like to explore this relatively new type of learning approach to a problem where both relational knowledge and probabilistic models are necessary for the task and see if we can build our classifiers using techniques from this new approach.

6.10 Discourse Processing

The current approach to database information retrieval through a natural language interface assumes that all the necessary information to interpret a user database access is present in the context of the sentence. In reality, however, information exchange happens more often through a dialog. This suggests that extending the current approach to allow a dialog between the user and the computer would be desirable. However, this requires processing of discourse context and information and integrating them with the meaning of the sentence to fully interpret the meaning of a user database access (Miller et al., 1996; Koppelman, Pietra, Epstein, & Roukos, 1995). More precisely, given a discourse history H (which contains all the previous user input sentences in the current session of a database access), we need to find the query $\hat{q}$ given a sentence l such that

$$\hat{q} = \text{argmax}_q \; P(q \in \mathit{Meaning}(l, H) \mid \exists s_{m+1} : l \Rightarrow s_m \wedge r_H(s_m, s_{m+1}) \wedge s_{m+1} \Rightarrow q) \qquad (31)$$

where $\mathit{Meaning}(l, H)$ is set of correct queries given the sentence l and the discourse history H, s_m is the final state of parsing the sentence l using the acquired semantic parser (without operator invention), r_H is a relation that relates a (final) state s_m to a possible *completion* s_{m+1} of the state given H which is further parsed into the query q (possibly with operator invention). A completion of a state given the discourse context is the result of a transformation of the state using the information from the context. Since the discourse context H is now part of the probabilistic model and it needs to be included in the new model.

The definition of a good state also needs to be extended as well since we now have a *pre-discourse* and a *post-discourse* model. The pre-discourse model is a probabilistic model of parsing the sentence l into a certain good final state s_m. Discourse processing is primarily handled by properly constructing the relation r_H and building a probabilistic model for it. The post-discourse model is a probabilistic model of parsing the resulting completed state s_{m+1} into the target query q. It is immediately obvious that our current approach is a special case where $H = \emptyset$, r_H is the identity relation (which relates everything to itself), and $s_m = s_{m+1}$. Suppose $R_H(s_m) = \{s \mid r_H(s_m, s)\}$ (i.e. the set of possible completions of s_m given H). A good state x in the pre-discourse model is one such that there exists a "path" from it to the target query, or more precisely, $x \Rightarrow y$ and there exists $z \in R_H(y)$ such that $z \Rightarrow q$. A good state in the post-discourse model can simply be the same as the definition of a good state in our current framework. It is interesting to verify that the definition of a good state in the pre-discourse model is reduced to that of the post-discourse model when $H = \emptyset$ and r_H is the identity relation. Now, we can construct a probabilistic model of $P(q \in \mathit{Meaning}(l, H))$ (the conditions are dropped for the simplicity of discussion):

$$P(q \in \mathit{Meaning}(l, H)) = \tag{32}$$
$$max_{s_{m+1} \in R_H(s_m)} \left\{ \prod_{i=1}^{m-1} P(s_{i+1}^{+} \mid s_i^{+}) \cdot P(s_{m+1}^{+} \mid s_m^{+}, H) \cdot \prod_{j=m+1}^{n-1} P(s_{j+1}^{+} \mid s_j^{+}) \right\}$$

where s_n is a final state in the post-discourse model and x^+ represents generally the notion that the state x is a good state in either models. The three terms on the right side of the above equation (from left to right) represent the pre-discourse model, the discourse processing model, and the post-discourse model respectively. The estimation of probabilities in the post-discourse is the same as that of our current approach. Intuitively, the above equation means that the probability of a query q being a correct meaning of a sentence l given the discourse context H is the product of the probability that l can be parsed to some good final state s_m (in the pre-discourse model), the probability that the resulting transformed state (given s_m and the discourse context) s_{m+1} is a good state (in the post-discourse model), and the probability that this state is parsed to some good final state s_n (in the post-discourse model). Since in general there could be more than one path from l to q, we just choose the one that has the maximum probability. Of course, more work still needs to be done to verify that the probabilistic model defined above is a reasonable one. Let's go through a trace of the following example to illustrate the task of discourse processing. We assume that the relation r_H has been properly constructed already. Suppose we have the following scenerio in a database access session:

USER1: What is the capital of Texas?
SYSTEM1: < Display the answer Austin>
USER2: What about California?
SYSTEM2: < Display the answer Sacramento>

The system starts with an empty discourse $H = \emptyset$ when the session begins. Therefore, the first question asked by the user is simply parsed to a logical query like we did before. More precisely, we have the following parse state and logical query at the end of the parsing:

Parse Stack:	[answer(C, (capital(C,S), const(S,stateid(texas)))):[?,texas,of,capital,the,is,what]]
Input Buffer:	[]
Query:	answer(C, (capital(C,S), const(S,stateid(texas))))

Now, the user asks the second question and the discourse history contains the final parse state and the logical query as shown above. The system first parses the sentence to some final state and suppose we have the following:

Parse Stack:	[const(_,stateid(california)):[?,california], answer(_,_):[about,what]]
Input Buffer:	[]

At this point, the system constructs a possible completion of the final state given the discourse context. Let's assume that there is only one completion. The following is a reasonble possible completion given the discourse context:

Parse Stack:	[capital(_,_):[], const(_,stateid(california)):[?,california], answer(_,_):[about,what]]
Input Buffer:	[]

In this case, we assume that the system recognizes that the user is asking for information about the capital city of California and it therefore adds the necessary predicate on the parse stack. After that, the system proceeds to parse the resulting state and we finally have:

Parse Stack:	[answer(C, (const(S,stateid(california)), capital(C,S))):[?,california,about,what]]
Input Buffer:	[]
Query:	answer(C, (const(S,stateid(california)), capital(C,S)))

Now, we have a good final parse state and the system can proceed to retrieve the answer from the database. The above scenerio is, of course, an imagined example of what a real database access session could look like. Future work would be focusing on developing a method for constructing the relation given the discourse context and the probabilistic model for it.

7 Related Work

Our approach is novel in the sense that it combines the strength of a relational and statistical learning approach to the problem of semantic parsing. There are other (relatively recent) approaches to the same problem as well, and we are going to briefly review two related but different pieces of work on semantic parsing here. Both were applied to the task in the Air Travel Information Service (ATIS) project (Miller, Bobrow, Ingria, & Schwartz, 1994). The goal of the project is to develop a spoken natural language interface for air travel information like obtaining flight information.

7.1 Semantic Parsing with Discourse Modelling

Miller (1996) describes a system that maps an input user request for flight information to a corresponding frame structure which will be translated to SQL given a discourse context. Instead of logical queries, the meaning of a sentence is represented as a parse tree whose non-terminal nodes contain syntactic and semantic information for the components of the sentence. For example, the word *in* in the user request "When do the flights that leave from

Boston arrive in Atlanta?" (Miller et al., 1996) is represented as a tree node labelled with `prep` (a preposition) and `location` (a location indicator). A standard frame-slot representation is used to capture information expressed in a parse tree. The parse tree will be used to decide what frame type should be used and the slots contain necessary for constructing the target SQL which hopefully would retrieve the relevant flight information. The parse tree is also tagged with specific frame and slot contructors. The discourse processing part only focused on resolving "hidden" constraints in a user request. (Problems like pronoun resolution, however, are not addressed since they are less relevant to the task.) For instance, a possible "hidden" constraint in the user request "Which flights are available on Tuesday?" given the context that the user asked about flight information from Boston to Denver would be that *flights* may be referring to only those from Boston to Denver. The statistical model built for the parser can be broken down into two components: 1) a model for the "pre-discourse" meaning of a sentence and 2) a model for the "post-discourse" meaning. The "pre-discourse" model is built for the grammar which is encoded in a recursive transition network which resembles more to statistical syntactic parsing. In our approach, since we do not have an explicit grammar constructed for our domains, the model is built directly for the parsing actions learned from the training data. We do not have a "post-discourse" model either since our tasks do not assume the database querying is to be done in a dialogue orient environment. Ambiguity resolution of possible different frames is done by selecting the most likely one (the product of the two statistical models) given the current discourse context.

7.2 Using Semantic Classification Trees for Semantic Interpretation

Another system called CHANEL was developed to handle the same task (Kuhn & De Mori, 1995). An intermediate representation language is used to represent the meaning of an input sentence which can be automatically translated to SQL. The intermediate representation language has two components: 1) a list of displayed (database) attributes and 2) a set of constraints over those attributes. This is very similar to the approach we have taken in the Northern California restaurant domain where we also use a representation language that can be mapped to either SQL or logical query. In CHANEL, semantic interpretation is handled by using a semantic classification tree (SCT). An SCT is similar to the standard decision tree (Quinlan, 1986). The internal nodes, however, contain patterns expressed in a simplied form of regular expression for matching against an input sentence. These patterns check for specific combination of words in an input sentence. Training up an SCT is also very similar to decision tree learning, using the information gain metric. An SCT is built for each attribute; each input sentence is classified as either positive or negative to whether the particular database attribute should be included in the displayed attribute list. Creating a list of constraints is also done by using SCTs. However, it is more involved and the details will not be explained here. To map a user input to an SQL, the system first creates a partial parse using a hand-crafted chart parser. The output is transferred to a robust matcher using these SCTs to create the displayed attribute list and the set of constraints which will be further translated into an SQL.

8 Conclusion

Semantic parsing is an important area in natural language processing and has been proved useful for tackling realistic NLP tasks like building natural language interfaces. However, handcrafting such applications is both time-consuming and ineffective, resulting in poor performance and scalability. Automated parser acquisition is therefore desirable. However, to avoid robustness problems of building a deterministic parser, one may want to consider a more probabilistic framework of parsing.

In this proposal, a probabilistic framework for semantic parsing is presented. The framework models a parser directly and does not assume the use of a generative semantic grammar which may not be available to the domain of interest. A new ILP learning system, TABULATE, is also introduced which can be used to learn multiple models from a given set of training data. These multiple learned models are integrated via statistical techniques like linear weighted evidence combination to produce probabilistic models for learning a semantic parser. Experimental results show that such an approach could outperform a purely logical approach in terms of accuracy of the parser.

While existing research on integrating statistical and relational learning methods puts more emphasis on producing a classifier for a certain classification task, the emphasis here is not just to produce a classification system but also a precise probability model that would allow good parsing performance. Precision of probability estimation, therefore, takes an even higher priority. The current system suffered from some of these problems with probability estimation and therefore performed relatively close to a deterministic parsing approach in some of the domains where data may not be readily available. Future research direction would be focusing on further improving some of the problems encountered by the existing approach and demonstrating its improved ability to automate the construction of natural language interfaces to databases.

9 Acknowledgements

I would like to extend my gratitude to my supervising professor Dr. Raymond Mooney for his patience during the course of the work and his timely and critical comments without which the work could not have come this far. I would also like to thank everyone else in the committee for their time and effort. Special thanks to the people in the DaimlerChrysler Research and Technology Center who made the experimental database in the Northern California restaurant domain available for research purposes. The research was supported by the National Science Foundation under grant IRI-9704943 and a grant from the DaimlerChrysler Research and Technology Center at Palo Alto in California.

References

Abramson, H., & Dahl, V. (1989). *Logic Grammars.* Springer-Verlag, New York.

Ali, K., & Pazzani, M. (1995). Hydra-mm: Learning multiple descriptions to improve classification accuracy. *International Journal on Artificial Intelligence Tools*, *4*.

Ali, K., & Pazzani, M. (1996). Error reduction through learning multiple descriptions. *Machine Learning Journal*, *24:3*, 100–132.

Allen, J. F. (1995). *Natural Language Understanding (2nd Ed.).* Benjamin/Cummings, Menlo Park, CA.

Armstrong-Warwick, S. (1993). Preface (to the special issue on using large corpora). *Computational Linguistics*, *19*(1), iii–iv.

Bahl, L. R., Jelinek, F., & Mercer, R. (1983). A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *5*(2), 179–190.

Baxt, W. G. (1992). Improving the accuracy of an artificial neural network using multiple differently trained networks. *Neural Computation*, *4*, 772–780.

Berwick, B. (1985). *The Acquisition of Syntactic Knowledge.* MIT Press, Cambridge, MA.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Brown, J. S., & Burton, R. R. (1975). Multiple representations of knowledge for tutorial reasoning. In Bobrow, D., & Collins, A. (Eds.), *Representation and Understanding.* Academic Press, New York.

Buntine, W. (1990). *A theory of learning classification rules.* Ph.D. thesis, University of Technology, Sydney, Australia.

Califf, M. E., & Mooney, R. J. (1999). Relational learning of pattern-match rules for information extraction. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pp. 328–334 Orlando, FL.

Califf, M. E. (1998). *Relational Learning Techniques for Natural Language Information Extraction.* Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, TX. Also appears as Artificial Intelligence Laboratory Technical Report AI 98-276 (see http://www.cs.utexas.edu/users/ai-lab).

Cameron-Jones, R. M., & Quinlan, J. R. (1994). Efficient top-down induction of logic programs. *SIGART Bulletin*, *5*(1), 33–42.

Cestnik, B. (1990). Estimating probabilities: A crucial task in machine learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pp. 147–149 Stockholm, Sweden.

Charniak, E. (1996). Tree-bank grammars. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1031–1036 Portland, OR.

Charniak, E., Hendrickson, C., Jacobson, N., & Perkowitz, M. (1993). Equations for part-of-speech tagging. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 784–789 Washington, D.C.

Church, K. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the Second Conference on Applied Natural Language Processing*, pp. 136–143 Austin, TX. Association for Computational Linguistics.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 115–123.

Collins, M. J. (1996). A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pp. 184–191 Santa Cruz, CA.

Collins, M. J. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pp. 16–23.

Duda, R., Gaschnig, J., & Hart, P. (1979). Model design in the prospector consultant system for mineral exploration. In Michie, D. (Ed.), *Expert systems in the micro-electronic age*. Edinburgh University Press, Edinburgh, England.

Friedman, N., Getoor, L., Koller, D., & Pfeffer, A. (1999). Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)* Stockholm, Sweden.

Gams, M. (1989). New measurements highlight the importance of redundant knowledge. In *European Working Session on Learning (4th: 1989)* Montpeiller, France.

Hendrix, G. G., Sacerdoti, E., Sagalowicz, D., & Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, *3*(2), 105–147.

Hirschberg, J. (1998). Every time I fire a linguist, my performance goes up, and other myths of the statistical natural language processing revolution. Invited talk, Fifteenth National Conference on Artificial Intelligence (AAAI-98).

Jelinek, F., & Mercer, R. L. (1980). Interpolated estimation of markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice* Amsterdam, The Netherlands.

Kijsirikul, B., Numao, M., & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 44–49 San Jose, CA.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, *1*, 4–7.

Kononenko, I., & Kovacic, M. (1992). Learning as optimization: Stochastic generation of multiple knowledge. In *Machine Learning: Proceedings of the Ninth International Workshop* Aberdeen, Scotland.

Koppelman, J., Pietra, S. D., Epstein, M., & Roukos, S. (1995). A statistical approach to language modeling for the atis task. In *Eurospeech 1995* Madrid.

Kuhn, R., & De Mori, R. (1995). The application of semantic classification trees to natural language understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *17*(5), 449–460.

Kwok, S., & Carter, C. (1990). Multiple decision trees. *Uncertainty in Artificial Intelligence*, *4*, 327–335.

Litman, D. J. (1996). Cue phrase classification using machine learning. *Journal of Artificial Intelligence Research*, *5*, 53–95.

Manning, C. D., & Carpenter, B. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the Fifth International Workshop on Parsing Technologies*, pp. 147–158.

Merialdo, B. (1994). Tagging English text with a probabilistic model. *Computational Linguistics*, *20*(2), 155–172.

Miller, S., Bobrow, R., Ingria, R., & Schwartz, R. (1994). Hidden understanding models of natural language. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pp. 25–32.

Miller, S., Stallard, D., Bobrow, R., & Schwartz, R. (1996). A fully statistical approach to natural language interfaces. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pp. 55–61 Santa Cruz, CA.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339–352 Ann Arbor, MI.

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 281–297. Academic Press, New York.

Muggleton, S. H., Srinivasan, A., & Bain, M. E. (1992). Compression, significance and accuracy. In *Proceedings of the Ninth International Conference on Machine Learning.*

Muggleton, S., & Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19*, 629–679.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, Inc., San Mateo,CA.

Pereira, F., & Shabes, Y. (1992). Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pp. 128–135 Newark, Delaware.

Plotkin, G. D. (1970). A note on inductive generalisation. *Machine Intelligence*, *5*, 153–163.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Quinlan, J. R. (1996a). Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 725–730 Portland, OR.

Quinlan, J. R. (1996b). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Ratnaparkhi, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, *34*, 151–176.

Reeker, L. H. (1976). The computational study of language acquisition. In Yovits, M., & Rubinoff, M. (Eds.), *Advances in Computers*, Vol. 15, pp. 181–237. Academic Press, New York.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*, 465–471.

Siklossy, L. (1972). Natural language learning by computer. In Simon, H. A., & Siklossy, L. (Eds.), *Representation and meaning: Experiments with Information Processsing Systems*. Prentice Hall, Englewood Cliffs, NJ.

Slattery, S., & Craven, M. (1998). Combining statistical and relational methods for learning in hypertext domains. In Page, D. (Ed.), *Proceedings of the 8th International Workshop on Inductive Logic Programming*, pp. 38–52. Springer, Berlin.

Stanley, C., & Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*.

Thompson, C. A., & Mooney, R. J. (1999). Automatic construction of semantic lexicons for learning natural language interfaces. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pp. 487–493 Orlando, FL.

Tomita, M. (1986). *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston.

Waltz, D. L. (1978). An English language question answering system for a large relational database. *Communications of the Association for Computing Machinery*, *21*(7), 526–539.

Warren, D. H. D., & Pereira, F. C. N. (1982). An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, *8*(3-4), 110–122.

Webb, G. I. (1996). Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, *4*, 397–417.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the Association for Computing Machinery*, *13*, 591–606.

Zelle, J. M. (1995). *Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers*. Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, TX. Also appears as Artificial Intelligence Laboratory Technical Report AI 96-249.

Zelle, J. M., & Mooney, R. J. (1994). Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 343–351 New Brunswick, NJ.

Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1050–1055 Portland, OR.