Unpublished Technical Note

An Inductive Logic Programming Method for Corpus-based Parser Construction

John M. Zelle
Dept. of Math and Computer Science
Drake University
Des Moines, IA 50311
zelle@zelle.drake.edu

Raymond J. Mooney
Dept. of Computer Sciences
University of Texas
Austin, TX 78712
mooney@cs.utexas.edu

March 18, 1997

Abstract

Empirical methods for building natural language systems has become an important area of research in recent years. Most current approaches are based on propositional learning algorithms and have been applied to the problem of acquiring broad-coverage parsers for relatively shallow (syntactic) representations. This paper outlines an alternative empirical approach based on techniques from a subfield of machine learning known as Inductive Logic Programming (ILP). ILP algorithms, which learn relational (first-order) rules, are used in a parser acquisition system called Chill that learns rules to control the behavior of a traditional shift-reduce parser. Using this approach, Chill is able to learn parsers for a variety of different types of analyses, from traditional syntax trees to more meaning-oriented case-role and database query forms. Experimental evidence shows that Chill performs comparably to propositional learning systems on similar tasks, and is able to go beyond the broad-but-shallow paradigm and learn mappings directly from sentences into useful semantic representations. In a complete database-query application, parsers learned by Chill outperform an existing hand-crafted system, demonstrating the promise of empricial techniques for automating the construction certain NLP systems.

1 Introduction

One of the enduring goals of natural language processing is the design of parsers that can translate natural language inputs into an internal representation suitable for computer manipulation. Traditional work in NLP has focused on a *rationalist* approach to the parsing problem, searching for perspicuous, rule-based representations of the knowledge required for language processing. Despite progress, traditional NLP has not yet succeeded in producing accurate, robust, broad-coverage systems for understanding English or other natural languages. Even the construction of limited-domain applications remains difficult and time-consuming, yielding systems that are often inefficient and incomplete. Fielded systems often have an ad hoc quality and porting them to new domains can mean essentially starting from scratch. Partially in response to these difficulties, there has been increasing recent interest in *empirical* approaches to natural language processing.

The empirical alternative, which has been very successful in speech recognition, replaces hand-generated rules with models obtained automatically by training over language corpora. Corpus-based methods may be used to augment the knowledge of a traditional parser, for example by acquiring new case-frames for verbs (Manning, 1993) or acquiring models to resolve lexical or attachment ambiguities (Lehman, 1994; Hindle & Rooth, 1993). More radical approaches attempt to replace the hand-crafted components altogether, extracting all required linguistic knowledge directly from suitable corpora. The more ambitious approach will be the main focus of this paper, but much of the discussion applies equally well to the more piecemeal approaches.

Empirical methods effectively divide the building of natural language systems into two tasks: annotation (corpus building), and acquisition. Annotation, is the province of human experts (perhaps with mechanical help). They must devise a training corpus that demonstrates the type of NL analysis required. For example, if the desired system is a syntactic parser, then the required corpus is a large sampling of text paired with the desired syntactic parse trees. Such a corpus is sometimes called a *treebank* (Marcus, Santorini, & Marcinkiewicz, 1993). Although some systems have used raw (unannotated) text for grammar acquisition, those employing annotations have produced more accurate parsers (Periera & Shabes, 1992).

The second task, acquisition, is a machine learning problem. Given a suitable training corpus, learning algorithms are employed to automatically construct a parser that can map subsequent inputs into the de-

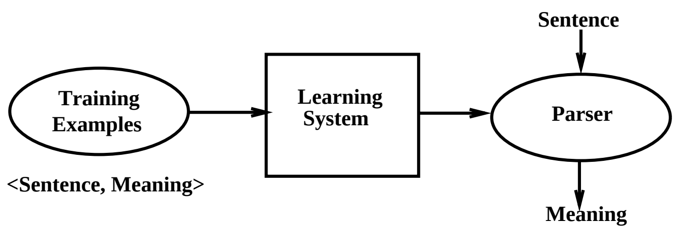

Figure 1: The parser acquisition problem.

sired representation. The parser acquisition problem is depicted in Figure 1. Development of a successful parser acquisition system would allow human designers to concentrate on engineering useful representations, relegating the difficult issue of constructing a parser for such representations to the acquisition system.

So far, empirical techniques have been employed largely in the attempt to create broad-coverage parsers for relatively shallow representations, as in part-of-speech tagging (Merialdo, 1994; Charniak, Hendrickson, Jacobson, & Perkowitz, 1993) and the induction of stochastic context-free grammars (Periera & Shabes, 1992) or transition networks (Miller, Bobrow, Ingria, & Schwartz, 1994). Following in the footsteps of speech recognition research, these methods eschew traditional, symbolic parsing in favor of statistical and probabilistic methods. Although several current methods learn some symbolic structures such as decision trees (Black, Jelineck, Lafferty, Magerman, Mercer, & Roukos, 1993; Magerman, 1995) and transformations (Brill, 1993), more traditional statistical methods dominate.

In fact, the term "parsing" is frequently restricted to the process of determining the *syntactic* structure of a sentence as a hierarchy of labeled constituents. Sometimes only the hierarchy itself is considered, resulting in unlabeled bracketings. However, parsing in this narrow sense represents only a small part of the understanding problem. Going beyond syntax to uncover important content, such as "who did what to whom", requires a more semantically oriented representation. Traditional NLP systems often employ case-role representations or deeper logical forms amenable to manipulation by automated deduction to capture such content. Dealing with this richer structure within an empirical framework presents an interesting challenge.

A common thread in previous empirical approaches is that the acquired knowledge is represented in a *propositional* form (perhaps with associated probabilities). This means, for example, that a decision about how to label a node in a parse tree is made by considering a fixed set of properties (e.g., syntactic category) about a fixed context of surrounding nodes (e.g., parent and immediate left sibling). The specifics of the decision policy are determined by the acquisition algorithm but the context over which the policy is formed, and the exact properties that can be tested, are determined *a priori* by the designer of the acquisition system. In machine learning, such approaches are often called feature-vector representations, as each decision context can be specified by a finite vector of atomic values associated with the various features of interest.

One advantage of propositional approaches is that the corresponding learning algorithms can be implemented very efficiently, allowing systems to be trained on very large corpora. This is an important consideration for the construction of broad-coverage, syntactic parsers. However, propositional techniques can be difficult to apply in recursive or richly relational domains, for example the kind of logical forms that are often the target of NLP research utilizing unification grammars.

This paper presents an approach to empirical parser construction that employs learning algorithms for structured (relational) knowledge representations of the type often employed in traditional NLP. Inductive Logic Programming (ILP) is a growing subfield of machine learning that addresses the problem of learning first-order logic descriptions (Prolog programs) from examples(Lavrač & Džeroski, 1994; Muggleton, 1992). Due to the expressiveness of first-order logic, ILP methods can learn relational and recursive concepts that cannot be represented in the feature-based languages assumed by most machine-learning algorithms. ILP methods have successfully induced small programs for sorting and list manipulation (Quinlan & Cameron-Jones, 1993) as well as produced encouraging results on important applications such as predicting protein secondary structure (Muggleton, King, & Sternberg, 1992). Detailed experimental comparisons of ILP and feature-based induction have demonstrated the advantages of relational representations in two language related tasks, text categorization (Cohen, 1995) and generating the past tense of an English verb (Mooney & Califf, 1995). Our research attempts to bridge the gap between rational and empirical approaches to NLP by applying ILP to the problem of parser acquisition.

The main advantage of this approach is its flexibility. Given the power of first-order rules, there is less need to hand-engineer appropriate features and contexts over which a system learns. The induction algorithm can automatically extract the relevant portions of structured contexts and construct new predicates to represent novel syntactic and/or semantic lexical and phrasal categories that it needs to make parsing decisions. With this flexibility, a single learning system is able to learn to parse sentences into a wide variety of representations including syntax trees, case-role mappings, and logical formulations based on predicate calculus. In contrast to statistical approaches that emphasize broad-coverage with shallow parsing, we are more interested in using empirical approaches to rapidly construct domain-specific parsers that produce usable semantic representations.

In this paper, we describe and discuss an initial implementation of this approach, called Chill (Constructive Heuristics Induction for Language Learning). Chill uses ILP techniques to learn heuristic rules for controlling a deterministic shift-reduce parser written in Prolog. The next section provides an introduction to ILP and how it may be used for parser acquisition. Section 3 gives a detailed description of Chill by way of a simple case-role parsing example. Section 4 presents the details of the ILP induction algorithm used in Chill. Section 5 presents experimental results demonstrating Chill' performance on case-role mapping tasks, constructing parse trees for the ATIS corpus from the Penn Treebank, and logical forms for database queries. Sections 6 and 7 discuss related and future work, and section 8 presents our conclusions.

2 Using ILP for Parser Construction

2.1 Introduction to ILP

ILP research considers the problem of inducing a first-order, definite-clause logic program from a set of examples and given background knowledge. As such, it stands at the intersection of the traditional fields of machine learning and logic programming.

As a sample ILP task, consider learning the concept of list membership. The input to the learning system consists of a number of positive and negative instances of the predicate, `member/2`. [1] Some positive instances might be: `member(1, [1,2])`, `member(2, [1,2])`, `member(1,[3,1])`, etc..[2] While instances such as `member(1,[])`, `member(2,[1,3])`, would serve as negative examples. Additional information is provided in the form of background relations in terms of which the desired concept is to be learned. In the case of list membership, this information might include a definition of the concept, `components/3` which

[1] We use the standard notation ⟨**name**⟩/⟨**number**⟩ to indicate the name and arity (number of arguments) for a predicate.

[2] The square brackets are standard Prolog notation for list construction. Thus, `[1,2]` is the list containing `1` and `2`.

decomposes a list into its component head and tail. This type of "constructor" predicate is typically used, as many ILP systems learn function-free clauses; using `components/3` eliminates the need for list constructions (e.g. `[X|Y]`) within learned clauses.[3] Given this input, an ILP system attempts to construct a concept definition which entails the positive training examples, but not the negatives. In this case, we hope to learn the correct definition of `member`, namely:

member(X, List) :- components(List, X, Tail).

member(X, List) :- components(List, Head, Tail), member(X, Tail).

2.1.1 Top-down ILP Algorithms

ILP research has clustered around two basic induction methods, top-down and bottom-up. Top-down ILP algorithms learn program clauses by searching a space of possible clauses from general to specific in a manner analogous to traditional machine-learning approaches for inducing decision trees. Perhaps the best-known example is Quinlan's Foil (Quinlan, 1990; Cameron-Jones & Quinlan, 1994) which uses an information heuristic to guide search through the space of possible program clauses.

Foil learns a program one clause at a time using a greedy-covering algorithm that can be summarized as follows:

```
Let positives-to-cover = positive examples.
While positives-to-cover is not empty
    Find a clause, $C$, that covers a preferably large subset of positives-to-cover
        but covers no negative examples.
    Add $C$ to the developing definition.
    Remove examples covered by $C$ from positives-to-cover.
```

A clause covers an example if the head of the clause can be unified with the example, and the body of the clause is subsequently provable with the background relations.

[3]The notation `[X|Y]` denotes a Prolog list with the head being `X` and the tail `Y`, i.e. `X` is the first element of the list, and `Y` is a list of the remaining elements.

The "find a clause" step is implemented by a general-to-specific hill-climbing search that adds antecedents to the developing clause one at a time. At each step, it evaluates possible literals that might be added and selects one that maximizes an information-gain heuristic based on the sets of positive and negative *tuples* covered by the clause. A tuple is simply an instantiation of the variables appearing in the clause when it is used to cover an example. A single example may give rise to multiple tuples if there are distinct proofs of the example producing different variable bindings.

FOIL considers adding literals for all possible variablizations of each predicate as long as at least one of the arguments is an existing variable bound by the head or a previous literal in the body. Literals are evaluated based on the number of positive and negative tuples covered, preferring literals that cover many positives and few negatives. Let T be the set of tuples covered by a clause, and T' the tuples covered after extending the clause with a literal, L. Let T_+ denote the number of positive tuples in the set T and define:

$$I(T) = -\log_2(T_+/|T|). \tag{1}$$

The chosen literal is then the one that maximizes:

$$gain(L) = s \cdot (I(T) - I(T')), \tag{2}$$

where s is the number of of tuples in T that have extensions in T' (i.e. the number of current positive tuples covered by L). Here I computes the information content of "knowing" that a tuple covered by the current clause represents a positive example. Gain is the difference in the residual information content of the subset of tuples still covered after adding the antecedent. Many positive tuples in higher concentration (compared to negative ones) results in better gain.

As an illustration, consider learning the concept of `member/2`. Initially, *positives to cover* contains all of the positive examples provided of list membership. FOIL starts with the most general clause: `member(A,B) :- true`. This clause covers all of the positive and negative examples; FOIL will attempt to make it more specific by adding literals. Since the only background predicate provided is `components/3`, FOIL evaluates all possible literals which can be formed from this predicate and also recursive literals using `member/2`. The possible literals for components take the form: `components(A,C,D)`, `components(B,C,D)`, `components(A,B,C)`, `components(B,A,C)`, etc. There are 26 unique variablizations to try for this predicate. It is obvious that the literal, `components(B,A,C)` will show positive gain, since it covers a number of positive examples (those asserting membership of the first element of a list) and covers no negative examples. Hence, this literal may be chosen to generate the clause, `member(A,B) :- components(B,A,C)`. Since the clause covers no negative examples, it is complete. FOIL then uses this as the first clause of the learned definition, and the covered examples are removed from *positives-to-cover*.

In the next iteration, FOIL again starts with the clause, `member(A,B) :- true` and examines all possible next literals. The previously chosen literal, `components(B,A,C)`, will have no gain since there are no longer any examples in *positives-to-cover* which it can cover. However, `components(B,C,D)` will have some positive gain. This gain comes from the fact that all positive examples must have a non-empty list for B, but some of the negative examples will have empty lists or perhaps non-lists in the second argument position. Hence, the literal covers all of the positives, but excludes some of the negatives. Choosing this literal produces the clause, `member(A,B) :- components(B,C,D)`. Clearly this clause might still cover many negative examples (e.g. `member(1,[2])`), so FOIL continues to add literals. Adding the literal `member(A,C)` makes the clause consistent with all examples in *positives-to-cover* and excludes any negative examples. At this point, the desired two-clause definition has been learned.

As described in the above referenced papers, FOIL also includes many additional features such as: heuristics for pruning the space of literals searched, methods for including equality, negation as failure, and useful literals that do not immediately provide gain (*determinate literals*), pre-pruning and post-pruning of clauses to prevent over-fitting, and methods for ensuring that induced programs will terminate.

2.1.2 Bottom-up ILP Algorithms

Bottom-up methods search for program clauses by starting with very specific clauses and attempting to generalize them. In logic programs, general clauses may be used to prove specific consequences through resolution theorem proving. Bottom-up induction inverts the resolution process to derive general clauses from specific consequences. The overall effect is a compression of the concept definition, replacing many specific instances with a few general clauses from which the instances can be derived. A successful representative of this class is Muggleton and Feng's GOLEM (Muggleton & Feng, 1992).

Like FOIL, GOLEM may be viewed as a greedy set covering algorithm, where new clauses are hypothesized by considering *least-general generalizations* (LGGs) of more specific clauses (Plotkin, 1970). A clause G *subsumes* a clause C if there is a substitution for the variables in G that make the literals in G a subset of the literals in C. Informally, we could turn C into G by dropping some conditions and changing some constants to variables. If G subsumes C, anything that can be proved from C could also be proved from G, since G imposes fewer conditions. Hence G is said to be more general than C (assuming C does not also subsume G, in which case the clauses must be equivalent except for renaming of variables).

The LGG of clauses C_1 and C_2 is defined as the least general clause that subsumes both C_1 and C_2. An LGG is easily computed by "matching" compatible literals of the clauses; wherever the literals have differing structure, the LGG contains a variable. When identical pairings of differing structures occurs, the same variable is used for the pair in all locations.

For example, consider two specific clauses concerning the concept `uncle` in the context of some known familial relationships:

```
uncle(john,deb) :-
  sibling(john,ron), sibling(john,dave),
  parent(ron,deb), parent(ron,ben),
  male(john), male(dave), female(deb).
uncle(bill,jay):-
  sibling(bill,bruce)
  parent(bruce,jay), parent(bruce,rach),
  male(bill), male(jay).
```

The LGG of these clauses yields the rather complicated result:

```
uncle(A,B):-
  sibling(A,C), sibling(A,D),
  parent(C,B), parent(C,E), parent(C,F), parent(C,E)
  male(A), male(G), male(H), male(I).
```

Here `A` replaces the pair ⟨`john,bill`⟩, `B` replaces ⟨`deb,jay`⟩, `C` replaces ⟨`ron,bruce`⟩, etc. Note that the result contains four `parent` literals (two of which are duplicates) corresponding to the four ways of matching the pairs of `parent` literals from the original clauses. Similarly, there are four literals for `male`. In the worst case, the result of an LGG operation may contain n^2 literals for two input clauses of length n. The example LGG contains no `female` literal since the second clause does not contain a compatible literal. Straightforward simplification of the result by removing redundant literals yields:

```
uncle(A,B):-
    sibling(A,C) parent(C,B), male(A).
```

This is one of the two clauses defining the general concept, `uncle/2`.

The construction of the LGG of two clauses is in some sense "context free." The resulting generalization is determined strictly on the form of the input clauses, there is no consideration of potential background knowledge. In order to take background knowledge into account GOLEM produces candidate clauses by considering *Relative* LGGs (RLGGS) of positive examples with respect to the background knowledge. The idea is to start with the assumption that any and all background information might be relevant to determining that a particular instance is a positive example. Thus, each positive example is represented by a clause of the form: `E :-` ⟨`every ground fact`⟩ where ⟨`every ground fact`⟩ is a conjunction of all true ground literals that can be derived from the background relations. In the case of `member/2`, this would include facts such as `components([1],1,[])`, `components([1,2],1,[2])`, `components([2],2,[])`, etc. An RLGG of two examples is simply the LGG of the examples' representative clauses. The LGG process serves to generalize away the irrelevant conditions.

One difficulty of this approach is that interesting background relations will give rise to an infinite number of ground facts. For example, there can be no finite set of facts that completely describes the `components/3` relation, since lists may be indefinitely long. GOLEM builds initial representative clauses for examples by considering a finite subset corresponding to the facts that can be derived from the background predicates through a fixed number of binary resolutions. The greedy clause construction algorithm of GOLEM takes the following form:

```
Let $Pairs$ = random sampling of pairs of positive examples
Let $RLggs = \{C : \langle e, e' \rangle \in Pairs$ and $C = RLGG(e, e')$ and $C$ consistent$\}$
Let S be set of the pair $e, e'$ with best cover RLgg in $RLggs$
Do
    Let $Examples$ be a random sampling of positive examples
    Let $RLggs = \{C$: $e' \in Examples$ and $C = RLGG(S \bigcup e'))$ and $C$ consistent$\}$
    Find $e'$ = which produces greatest cover in $RLggs$
    Let $S = S \bigcup e'$
    Let $Examples = Examples - cover(RLGG(S))$
While increasing-cover
```

Golem starts by taking a sampling of RLGGs of pairs of uncovered positive examples. The RLGG that covers the most positive examples without covering any negatives is then further generalized through RLGG with other random samplings of positive examples. The process terminates when all subsequent RLGGs fail to cover more examples consistently (i.e. without covering negative examples). Like Foil, Golem also includes a number of filtering and clause–pruning heuristics to make the search process more efficient and prevent over-fitting.

2.2 Overview of CHILL

A straight-forward application of ILP to parser acquisition would be to simply present a corpus of sentences paired with representations (e.g. parse trees) as a set of positive examples to an ILP system. After learning a Prolog definition for the predicate `parse(Sentence, Representation)`, it can be used to prove goals having the second argument uninstantiated, thereby producing parses of input sentences. However, there is no convenient set of negative examples or an obvious set of background relations to provide for use in the learned clauses. Also, since parsers are very complex programs, it is unlikely that any existing ILP system could induce a complete parser from scratch that generalizes well to new inputs. The space of logic programs is simply too large, and the learning problem too unconstrained.

Chill attempts to address this problem by considering parser acquisition as a control-rule learning problem. Rather than using ILP techniques to directly learn a logic grammar, Chill begins with a well-defined parsing framework and uses ILP to learn control strategies within this framework. Treating language acquisition as a control-rule learning problem is not in itself a new idea. Berwick (1985) used this approach to learn grammar rules for a Marcus-style deterministic parser. When the system came to a parsing impasse, a new rule was created by inferring the correct parsing action and creating a new rule using certain properties of the current parser state as trigger conditions for its application. In a similar vein, Simmons and Yu (1992)

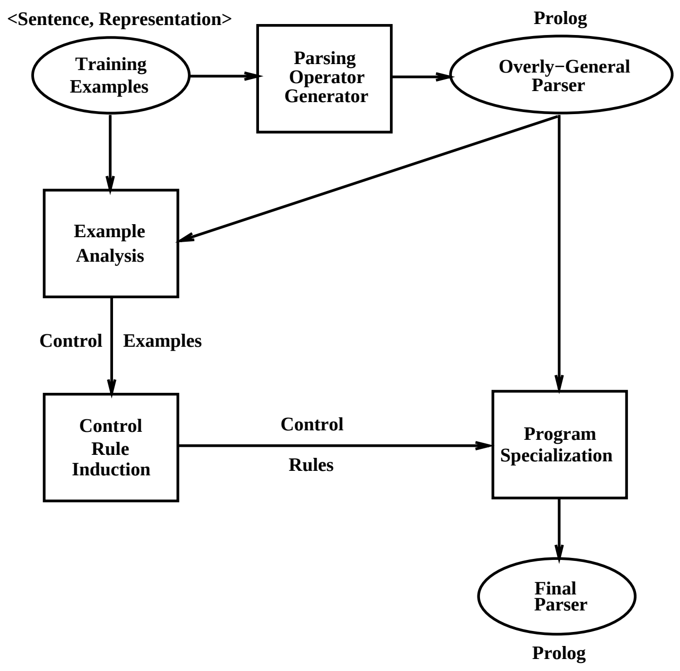

Figure 2: The CHILL Architecture

controlled a simple shift-reduce parser by storing example contexts consisting of the syntactic categories of a fixed number of stack and input buffer locations. New sentences were parsed by matching the current parse state to the stored examples and performing the action corresponding to the best matching previous context. More recently, some probabilistic parsing methods have also employed frameworks for learning to prefer parsing operators based on context (Briscoe & Carroll, 1993; Magerman, 1995). However, all of these systems use feature-vector representations that have only a limited, fixed-size access to the parsing context. In CHILL, parser states are simply represented as structured logical terms. The exact form of a parser state depends on the type of representation that the parser is expected to produce but includes the complete contents of the stack and the remaining input string. The burden of discovering the relevant structural differences in this complex, unbounded context is placed on the ILP learning algorithm.

The input to CHILL is a set of training instances consisting of sentences paired with the desired parses. The output is a shift-reduce parser in Prolog that maps sentences into parses. Figure 2 shows the basic components of the system. During Parser Operator Generation, the training examples are analyzed to formulate an overly-general shift-reduce parser that is capable of producing parses from sentences. The initial parser is overly-general in that it produces a great many spurious analyses for any given input sentence. In Example Analysis, the overly-general parser is used to parse the training examples to extract contexts in which the various parsing operators should and should not be employed. Control-Rule Induction then employs a general ILP algorithm to learn rules that characterize these contexts. Finally, Program Specialization "folds" the learned control-rules back into the overly-general parser to produce the final parser.

The following section explains the details of Chill in the context of a simple case-role mapping problem. The concreteness and simplicity of this problem allow for a more comprehensible and complete explanation of the mechanisms employed in Chill. Additional information on how the same general framework is applied to produce syntactic parse trees and logical queries is provided in section 5.

3 Acquiring a Case-role Parser with Chill

3.1 The Mapping Problem

Case theory (Fillmore, 1968) decomposes a sentence into a proposition represented by the main verb and various arguments such as agent, patient, and instrument, represented by noun phrases. The basic mapping problem is to decide which sentence constituents fill which roles. Though case analysis is only a part of the overall task of sentence interpretation, the problem is nontrivial even in simple sentences.

Consider these sentence/case-analysis examples given by McClelland and Kawamoto (1986):

1. The boy hit the window.	[hit agt:boy pat:window]
2. The hammer hit the window.	[hit inst:hammer pat:window]
3. The hammer moved.	[moved pat:hammer]
4. The boy ate the pasta with the cheese.	[ate agt:boy pat:pasta accomp:cheese]
5. The boy ate the pasta with the fork.	[ate agt:boy pat:pasta inst:fork]

In the first sentence, the subject, `boy`, is an agent. In the second, the subject, `hammer`, is an instrument. The role played by the subject must be determined on the grounds that boys are animate and hammers are not. In the third sentence, the subject, `hammer`, is interpreted as a patient, illustrating the importance of the relationship between the surface subject and the verb. In the last two sentences, the prepositional phrase could be attached to the verb (making `fork` an instrument of `ate`) or the object (`cheese` is an accompaniment of `pasta`), and semantic knowledge is required to make the correct assignment.

3.2 Constructing the Overly-General Parser

Our system adopts a simple shift-reduce framework for case-role mapping (Simmons & Yu, 1992). The process is best illustrated by way of example. Consider the sentence: "The man ate the pasta." Parsing

Action	Stack Contents
	[]
(shift)	[the]
(shift)	[man, the]
(1 det)	[[man, det:the]]
(shift)	[ate, [man, det:the]]
(1 agt)	[[ate, agt:[man, det:the]]]
(shift)	[the, [ate, agt:[man, det:the]]]
(shift)	[pasta, the, [ate, agt:[man, det:the]]]
(1 det)	[[pasta, det:the], [ate, agt:[man, det:the]]]
(2 obj)	[[ate, obj:[pasta, det:the], agt:[man, det:the]]]

Figure 3: Shift-Reduce Parsing of "The man ate the pasta."

begins with an empty stack and an input buffer containing the entire sentence. At each step of the parse, either a word is shifted from the front of the input buffer onto the stack, or the top two elements on the stack are popped and combined to form a new element which is pushed back onto the stack. The sequence of actions and stack states for our simple example is shown in Figure 3. The action notation *(x label)*, indicates that the stack items are combined via the role, *label*, with the item from stack position, x, being the head.

An advantage of assuming such a constrained parsing mechanism is that the form of structure building actions is limited. The operations required to construct a given case representation are directly inferable from the representation. In general, a structure building action is required for each unique case-role that appears in the analysis. The set of actions required to produce a set of analyses is the union of the actions required for each individual analysis.

A shift-reduce parser is easily represented as a logic program. The state of the parse is reflected by the contents of the stack and input buffer. Each distinct parsing action becomes an operator that takes the current stack and input and produces new ones. Figure 4 shows an overly-general program sufficient to parse the above example. The `parse/2` predicate takes a list of words representing a sentence and returns a case structure. The `parse/4` predicate maps a stack and input buffer in its first two arguments into a new stack and buffer in the third and fourth arguments. The mapping is performed by zero or more applications of simple actions represented by `op/4`, which takes the current stack and input string and produces an updated stack and input string. For example, the first clause of `op/4` implements the *(1 agt)* action. The `reduce/4` predicate simply attaches a value to a head via some label to produce a new structure.

Extending the program to parse further examples is accomplished by adding additional clauses to the `op/4` predicate. Each clause is a direct translation of a required parsing action. As was mentioned earlier, the identification of the necessary actions is straight-forward. A particularly simple approach is to include two actions (e.g., *(1 agt)* and *(2 agt)*) for each role used in the training examples; any unnecessary operator clauses will be removed from the program during the subsequent specialization process.

3.3 Example Analysis

The overly-general parser produces a great many spurious analyses for the training sentences because there are no conditions specifying when it is appropriate to use the various operators. In order to eliminate spurious parses, the appropriate context in which each operator should be applied needs to be characterized. Therefore, Chill specializes the parser by including control heuristics that guide the application of operators. For each clause of `op/4` Chill constructs a definition of the concept "subgoals for which this clause is useful". The definition of this concept comprises a set of clauses which examine the stack and input buffer and are satisfied by exactly those states to which the given operator should be applied. The job of example analysis

```
parse(S, Parse) :- parse([], S, [Parse], []).

parse(Stack, Input, Stack, Input).
parse(Stack0, In0, Stack, In) :-
        op(Stack0, In0, Stack1, In1), parse(Stack1, In1, Stack, In).

op([Top,Second|Rest],In,[NewTop|Rest],In) :- reduce(Top,agt,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :- reduce(Top,det,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :- reduce(Second,obj,Top,NewTop).
op(Stack,[Word|Words],[Word|Stack],Words).  % Shift operation.
```

Figure 4: Overly-General Parser for "The man ate the pasta."

is to construct sets of positive and negative *control examples* from which the appropriate control rules can be subsequently induced.

A control example is a specific subgoal to which a particular operator clause may be applied in the course of parsing an example. For a given operator, positive control examples represent parse states to which the operator should be applied. Such examples are generated by finding the first correct parsing of each training pair with the overly-general parser; any subgoal to which an operator is applied in this successful parse becomes a positive control example for that operator.

The extraction of negative control examples (subgoals to which an operator should *not* be applied) is performed under the assumption that the training corpus is *output complete*. This means that the set of training examples includes a pair for every correct parsing for each unique sentence appearing in the set. In other words, if a sentence used in training is to be treated as having N different readings, then that sentence must appear N times in the training set, paired once with each possible representation. Whether sentences should be allowed to have multiple parses is up to the NL system designer. If only the single, best parsing is preferred, then only one pair per sentence should be provided. If, however, it is desired that the system be able to produce multiple parsings for a given sentence, output completeness dictates that all readings of the training sentences be included in the training set.

The set of positive control examples along with the assumption of output completeness implicitly defines a set of negative control examples. Knowing the set of clauses which should be applied to a given subgoal identifies other clauses as those that should *not* be applied. The exact mechanism for generation of negative control examples is dependent on whether the final parser is intended to produce multiple parses. For parsers returning only a single parse, the positive examples for a given operator clause are considered negative examples for all *prior* clauses which do not have the same positive example. Since only one solution is being computed, subsequent clauses will not have a chance to match against this particular subgoal, and it need not be included in their negative example sets. For multiple-output parsers, a positive example for one clause is considered a negative example for *all* matching clauses that do not also have that subgoal as a positive example. This is necessary because subsequent clauses may be matched against this subgoal when backtracking for more solutions. These subsequent clauses should not be applied unless they lead to a correct alternative parse (in which case they will also have this subgoal as a positive control example extracted from the proof of a different training pair).

For the *(1 agt)* clause of `op/4` and the example sentence,"the man ate the pasta," Example Analysis would extract the single positive control example: `op([ate,[man, det:the]], [the,pasta], A, B)`. This is the only subgoal to which the *(1 agt)* reduction is applied in the correct parsing of the sentence. Notice that `A` and `B` are uninstantiated variables since they are outputs from the `op/4` clause and are not yet bound at the time the clause is applied. Allowing for multiple parses, all contexts where the first clause of `op/4` were not applied would become negative control examples. Thus, the following negative control examples would be generated for this operator:

```
op([man,the],[ate,the,pasta], A, B)

op([the,[ate,agt:[man,det:the]]], [pasta], A,B)

op([pasta,the],[ate,agt:[man,det:the]], [], A, B)

op([pasta,det:the],[ate,agt:[man,det:the]], [], A, B)
```

3.4 Control-Rule Induction

Once sets of positive and negative control examples have been extracted, the task of the induction component is to generate a definite-clause concept definition which covers the positive examples, but not the negative. This NLP task puts several demands on an ILP algorithm. First, the algorithm must deal gracefully with highly structured examples; the resulting rules are operating over arbitrarily large parse states. Second, the algorithm must be able to invent new predicates to make the distinctions that are necessary for accurate parsing in realistic domains. Experience suggests that it is unlikely a hand-crafted feature set will be complete enough on its own. Finally, the algorithm must be efficient enough to deal with thousands of examples. A single sentence gives rise to many control examples over which induction is performed. Furthermore, each parsing operator gives rise to a control-rule induction problem. Acquiring a parser over a corpus of even several hundred sentences gives rise to dozens of induction problems over hundreds or thousands of examples.

At the time Chill was being developed, no existing ILP algorithm met all of these requirements. Chill employs a novel ILP algorithm which combines elements of both top-down and bottom-up methods. Rules are initially generated by forming LGGs of clause pairs. Overly-general rules are then specialized by the addition of literals. In addition, the algorithm includes demand-driven predicate invention which allows it to create new concepts when necessary to discriminate the positive and negative examples. The details of the algorithm will be taken up in the next section, for now we'll concentrate on the results of the induction.

Given our example, a control rule that might be learned for the *(1 agt)* reduction is:

```
op([X,[Y,det:the]], [the,Z], A, B) :- animate(Y).

animate(man). animate(boy). animate(girl) ....
```

Here the system has invented a new predicate to help explain the parsing decisions. Of course, the new predicate would have a system generated name and is called "animate" here for clarity. This rule may be roughly interpreted as stating: "the agent reduction applies when the stack contains two items, the second of which is a completed noun phrase whose head is animate."

The output of the Control-Rule Induction phase is a suitable set of control-rules for each clause of `op/4`. These control rules are then passed on to the Program Specialization phase.

3.5 Program Specialization

The final step is to “fold” the control information back into the overly-general parser. A control rule is easily incorporated into the overly-general program by unifying the head of an operator clause with the head of the control rule for the clause and adding the induced conditions to the clause body. The definitions of any invented predicates are simply appended to the program. Given the program clause:

```
op([Top,Second|Rest],In,[NewTop|Rest],In) :- reduce(Top,agt,Second,NewTop).
```

and the control rule:

```
op([X,[Y,det:the]], [the,Z], A, B) :- animate(Y).
animate(man). animate(boy). animate(girl) ....
```

the resulting clause is

```
op([A,[B,det:the]],C,[D],C) :- animate(B), reduce(A,agt,B,D).
animate(boy). animate(girl). animate(man). animate(lion) . . .
```

The final parser simply consist of the overly-general parser with each operator clause suitably constrained. Because of the way the control rules are induced, the resulting parser is guaranteed to produce all and only the correct parses for each of the training examples.

3.6 Discussion of Shift-Reduce Parsing

This is an appropriate point to discuss a number of implications/motivations for the CHILL framework. Some might consider the shift-reduce framework as too limiting for NLP, citing well-known results about the power of LR(k) grammars. It is important to note, however, that the potential lookahead in our parser is unlimited since the entire state of the parser (current stack contents and the remaining input) may be examined in determining which action to perform. Furthermore, the control-rules that are learned to select operators in a given context are essentially arbitrary logic programs, therefore the class of languages recognized is, in principle, Turing complete.

Another potential confusion is the labeling of the learned parsers as “deterministic.” What does it mean to say that a parser with (potentially) unbounded look-ahead that can produce multiple parses of a single input sentence is deterministic? By this, we mean that the parser does not backtrack to undo an operation in the search for any single parse. Each action taken during parsing should be a correct step toward finding an acceptable interpretation. However, if there are multiple acceptable parses, the parser can backtrack to find an alternative correct action at some point that results in another appropriate analysis. Parsing is deterministic in the sense that an incorrect action is not applied that later needs to be undone. Although the parser can, in principle, examine unbounded context, the rule-induction framework favors simple rules resulting in a bias for the portions of the context sufficient to do accurate parsing. One might consider the learning problem as a search for the "most deterministic" parser than can correctly parse the training sentences.

In principle, the general mechanisms of CHILL could be used in conjunction with any parsing mechanism suitably encoded as a logic program. However, deterministic shift-reduce parsing is a very attractive choice for a number of reasons. First, it is one of the simplest, most constrained mechanisms that has promise for parsing significant portions of English (Marcus, 1980; Tomita, 1986). Second, it fits very well with the overall architecture of acquisition as control-rule learning; creating a deterministic parser is basically an exercise in identifying relevant control rules. Third, the resulting parsers are very efficient, increasing their potential utility for general NLP applications; in the experiments reported in section 5, sentences were parsed very quickly with no analysis requiring more than a second of CPU time.

Finally, shift-reduce parsing provides a principled control-structure that is representation neutral. Different styles of analysis can be produced by simply inserting different parsing operators. Nevertheless, certain general properties of language processing such as left-to-right scanning and compositionality are always maintained. The language processing mechanism provides bias that should simplify the work of the inductive learning component. Instead of having to learn how to build output structures, the learning problem reduces to identifying "states" of the parser. If these states capture generalities of language processing, then it is hoped that the inductive learning mechanism can successfully learn to parse into any consistent compositional representation scheme.

4 The Induction Algorithm

4.1 Overview

The input to the induction component is a set of positive and negative examples of a concept (in this case, control examples) expressed as facts, and a set of background predicates expressed as definite clauses. In the experiments conducted with Chill so far, the initial background knowledge has been empty. The output of the induction is a definite-clause concept definition which entails the positive examples, but not the negative. A summary of the induction algorithm is given below; Zelle and Mooney (1994a) and Zelle (1995) presents additional details along with empirical results on a number of benchmark ILP problems.

Chill employs a compaction algorithm which tries to construct a small, simple program that covers the positive examples. The algorithm starts with a most specific definition (the disjunction of the specific positive examples) and introduces generalizations that make the definition more compact as determined by a simple measure of the syntactic size of the program. The search for more general definitions is carried out in a hill-climbing fashion. At each step, a number of possible generalizations are considered; the one producing the greatest compaction is implemented, and the process repeats.

Generalizations are produced under the notion of *empirical subsumption*. Intuitively, the algorithm attempts to construct a clause that, when added to the current definition, renders other clauses superfluous. The superfluous clauses are then eliminated to produce a more compact definition. Formally, we define empirical subsumption as follows: Given a set C of Clauses $\{C_1, C_2, \ldots, C_N\}$ and a set of positive examples E provable from C, a clause G empirically subsumes C_i iff $\forall e \in E : [(C - C_i) \cup G \vdash e]$. That is, all examples in E are still provable if C_i is replaced by G. Throughout this description of the induction algorithm, unless otherwise noted, the term "subsumption" should be interpreted in this empirical sense.

Figure 5 shows the basic compaction loop. As in Golem, generalizations are constructed from a random sampling of pairs of clauses in the current definition. The best generalization produced from these pairs, `G`, is used to reduce the current definition, `DEF`. The reduction is performed by adding `G` at the top of the definition and then using the standard Prolog proof strategy to find the first proof of each positive example; any clause which is not used in one of these proofs is then deleted from the definition since it is subsumed by `G`.

```
DEF := {E :- true | E ∈ Pos}
Repeat
    PAIRS := a sampling of pairs of clauses from DEF
    GENS := {G | G = build_gen($C_i$,$C_j$,DEF,Pos,Neg) for $\langle C_i, C_j \rangle$ ∈ PAIRS}
    G := Clause in GENS yielding most compaction
    DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction
```

Figure 5: Basic Induction Algorithm

The algorithm for constructing generalized clauses is outlined in Figure 6. There are three basic processes involved. First is the construction of a LGG of the input clauses. If this generalization covers no negative examples, it is returned. If the initial generalization is too general, an attempt is made to specialize it by adding antecedents. If the expanded clause is still too general, it is passed to a routine which invents a new predicate that further specializes the clause so that it covers no negative examples. These three processes are explained in detail in the following subsections.

4.2 Constructing an Initial Generalization

The initial generalization of the input clauses is computed by finding the LGG of the clauses. For example, when learning control rules for agent reduction, some initial clauses might be the following:

```
op([ate,[man,det:the]], [the,pasta], A, B) :- true.
op([hit,[boy,det:the]], [the,man], A, B) :- true.
op([ate,[boy,det:the]], [the,chicken], A, B) :- true.
```

The first and third clauses yield an LGG as follows:

```
1) op([ate,[man,det:the]], [the,pasta], A, B) :- true.
3) op([ate,[boy,det:the]], [the,chicken], A, B) :- true.
-------------------------------------------------------
LGG: op([ate,[X,det:the]],[the,Y], A, B):- true.
```

This LGG is a consistent generalization (it covers no negative examples) and no further processing is required. Of course, such generalizations are not always consistent. Consider the LGG of the first and second clauses:

```
1) op([ate,[man,det:the]], [the,pasta], A, B) :- true.
2) op([hit,[boy,det:the]], [the,man], A, B) :- true.
-------------------------------------------------------
LGG: op([X,[Y,det:the]],[the,Z],A, B):- true.
```

This generalization covers potential negatives such as: `op([hit,[hammer,det:the]], [the, window], A, B)` where hammer should be attached as an instrument rather than an agent. The generalization requires further refinement to prevent coverage of such examples.

Although the initial definitions consist of unit clauses (the only antecedent being `true`), as the definition becomes more compact, the clauses from which LGGs are constructed may contain non-trivial conditions. However, the construction of clause LGGs is still straight-forward. Unlike the RLGGs used by Golem, the LGGs in Chill are independent of background knowledge and, since the input clauses are generally small, they are efficiently computable. The resulting clause is guaranteed to be at least as general as either input clause, but may also cover negative examples. This process also effectively introduces relevant variables which decompose the functional structures appearing in the examples. These variables may then be constrained by adding antecedents to the clause.

4.3 Adding Antecedents

As its name implies, `add_antecedents` attempts to specialize an initial genearlization, `GEN`, by adding new literals as antecedents. The goal is to minimize coverage of negative examples while insuring that the clause still subsumes existing clauses. `Add_antecedents` employs a Foil-like mechanism that adds literals, using background or previously invented predicates. Antecedents are added one at a time using a hill-climbing process; at each step a literal is added that maximizes a heuristic gain metric.

The gain metric employed in Chill is a modification of the Foil information-theoretic gain metric. Good generalizations for Chill are those that subsume many existing clauses. Therefore, the count of positive tuples (loosely, the number of covered positive examples) in the Foil metric is replaced by an estimate of

```
Function build_gen($C_i$, $C_j$, DEF, Pos, Neg)
GEN := LGG($C_i$,$C_j$)
CNEGS := Negatives covered by GEN
if CNEGS = {} return GEN

GEN := add_antecedents(Pos, CNEGS, GEN)
CNEGS := negatives covered by GEN
if CNEGS = {} return GEN

REDUCED := DEF - (Clauses subsumed by GEN)
CPOS := $\{e \mid e \in Pos \wedge REDUCED \not\vdash E\}$
LITERAL := invent_predicate(CPOS, CNEGS, GEN)
GEN := GEN $\cup$ LITERAL
return GEN
```

Figure 6: Build_gen Algorithm

the number of clauses in `DEF` which are subsumed by `GEN`. This estimate is obtained by associating each positive example with the first clause in `DEF` that covers it. If `GEN` covers all of the examples associated with any clause, that clause is counted as subsumed by `GEN`.

As an example of this process, consider learning the control rule for the *(1 agt)* reduction in the presence of suitable background relations regarding word categories such as `person/1` and `animate/1`. Initially, `DEF` contains unit clauses representing the positive control examples. Each example will be associated with the unit clause which was constructed from it. A sampling of pairs of clauses would then be used to construct LGGs. As illustrated above, one generalization might be: `op([ate,[X,det:the]],[the,Y], A, B):- true`. This clause is not overly-general, and no antecedents need to be added. If this were the best compacting generalization found from the sampling, it would be added at the top of the definition, and all of the more specific clauses subsumed by this generalization would be removed from `DEF`. Of course, some unit clauses would remain to cover examples having verbs other than "ate."

In the next cycle of the compaction loop, the LGG of two clauses may produce `op([X,[Y,det:the]], [the,Z], A, B) :- true`. This clause is overly-general, and must be specialized before it can be considered. The FOIL-like component will consider possible new antecedents such as `person(X)`, `person(Y)`, `person(Z)`, etc. Trying the literal, `person(Y)` produces the clause: `op([X,[Y,det:the]], [the,Z], A, B) :- person(Y)`. Presumably, this clause is consistent and will subsume numerous unit clauses in the current definition; therefore, it will have some positive gain. This clause, however, will not subsume the generalization found in the previous iteration, as some of the examples associated with the "ate" generalization will have non-person, animate agents (e.g., `op([ate,[lion,det:the]], [the,sheep], A, B)`). In contrast, the clause: `op([X,[Y,det:the]], [the,Z], A, B) :- animate(Y)` subsumes all of these example as well as those subsumed by `person(X)`. Hence, `animate(X)` is a superior literal according to the gain metric. When this clause is added to `DEF` the previous generalization as well as the remaining unit clauses become superfluous. At this point, `DEF` collapses to this single clause, and the induction is complete.

This discussion has assumed that `add_antecedents` has predicates available which will allow it to completely discriminate between the positive and negative examples; however, this is not always the case. In such situations, `add_antecedents` may or may not add a few antecedents before it is unable to extend the clause further because no literal has positive gain. This partially completed clause is then passed to `invent_predicate` for completion.

4.4 Inventing New Predicates

A clause passed to `invent_predicate` covers both some positive and some negative examples. The purpose of inventing a new predicate is to constrain some of the variables appearing in the clause so as to exclude the negative examples.

Suppose that we are trying to learn the control rule for the (1 agt) reduction, but do not have suitable background knowledge. Then `add_antecedents` will be unable to specialize an LGG such as `op([X,[Y,det:the]], [the,Z],A, B) :- true`. Using this clause to cover positive and negative examples might result in a set of variable bindings shown here in tabular form:

Set	X	Y	Z
Pos	ate	man	pasta
	hit	boy	sheep
	moved	girl	fork
Neg	hit	hammer	window
	hit	ball	pasta
	broke	bat	plate

Note that, while the domains of `X` and `Z` have certain values which appear in both positive and negative examples, the values taken by `Y` are disjoint. A new concept representing "values of `Y` that appear in positive examples" could be used to specialize this clause so as to insure that it does not cover negative examples.

In general, separating the positive and negative examples may require simultaneously constraining multiple variables. In the table above, `X` and `Z` taken together do separate the examples even though they do not do so individually. Chill uses an approach similar to Champ (Kijsirikul, Numao, & Shimura, 1992), employing a greedy algorithm to find a small set of variables that are sufficient for separation. The search begins with an empty set of variables and adds variables one at a time, preferring variables that eliminate overlap with negative examples and minimize the cardinality of the set of postive tuples. The instantiations of these variables determine sets of positive and negative examples for the new concept. The induction algorithm is then recursively invoked with these examples to learn a definition of the new concept.

Returning to the example, `Invent_predicate` will select the single variable `Y` and create positive examples `p1(man)`, `p1(boy)` and `p1(girl)`, along with the negative examples `p1(hammer)`, `p1(ball)`, and `p1(bat)`. Calling the top-level induction algorithm on these examples produces no compaction, so the learned definition of `p1` will just be a listing of the positive examples. Finally, `build_gen` completes its clause by adding the final literal `p1(Y)` which is the newly invented predicate representing `animate`. Once a predicate has been invented and found useful for compressing the definition, it is made available for use in further generalizations. This enables the induction of clauses having multiple invented antecedents.

As discussed by Zelle and Mooney (1994a) and Zelle (1995), the actual implementation of Chill is somewhat more complex since it includes numerous enhancements to improve efficiency. An exact analysis of the computational complexity is difficult; like all practical ILP systems, the search performed here uses many heuristics. All of the fundamental components have polynomial time complexity except that the top-down FOIL-like component is exponential in the arity of the background predicates. Since predicate arity is generally small (1 to 4 arguments is typical) this is generally not a significant problem. CHILL is implemented in Prolog and utilizes the Prolog inference engine to test example coverage. This use of general theorem proving does result in relatively high constant time and space factors. This probably makes the current implementation of CHILL impractical for corpora containing many thousands of sentences. It is, however, quite tractable for hundreds of sentences. CHILL running on a SPARCstation 2, was able to induce parsers for the experiments reported below in CPU times ranging from several minutes to a few hours.[4].

5 Experimental Results

CHILL has been tested on a number of parsing tasks ranging from simple case-role mapping problems to the construction of a complete database-query application. The following subsections discuss several of these experiments.

5.1 Case-Role Mapping

CHILL was first tested on artificial data for case-role mapping previously used to demonstrate certain language processing abilities of artificial neural networks (McClelland & Kawamoto, 1986; Miikkulainen & Dyer, 1991). The corpus consists of 1475 sentence/case-structure pairs produced from a set of 19 sentence templates such as "The HUMAN ate the FOOD with the UTENSIL", where the capitalized items are replaced with words of the given category. The corpus actually contains 1390 unique sentences, some of which admit two meaningful analyses. Therefore, the parser was trained to produce multiple outputs where each unique sentence was treated as a single example and the training corpus was insured to be output complete.

Training and testing followed the standard paradigm of generating learning curves by first choosing a random set of test examples (in this case 740) and then creating parsers using increasingly larger subsets of the remaining examples. The reported results are averages over five random training/test splits. During

[4]The code for CHILL is being made available electronically at `ftp://ftp.cs.utexas.edu/pub/mooney/chill/`

testing, the learned parser was used to enumerate all analyses for a given test sentence. Parsing of a sentence can fail in two ways: an incorrect analysis may be generated, or a correct analysis may not be generated. In order to account for both types of errors, we measured $Accuracy = (\frac{C}{P} + \frac{C}{A})/2$, where P is the number of distinct analyses produced, C is the number of the produced analyses which were correct, and A is the number of correct analyses possible for the sentence. This measure can be viewed as an average of the parser's precision and recall for a given sentence.

Chill performs very well on this learning task as demonstrated by the learning curve shown in Figure 7. The system achieves 92% accuracy on novel sentences after seeing only 150 training sentences. Training on 650 sentences required less than an hour of CPU time and generated a set of parsing operators comprising about 60 lines of Prolog code that achieved 98% accuracy.

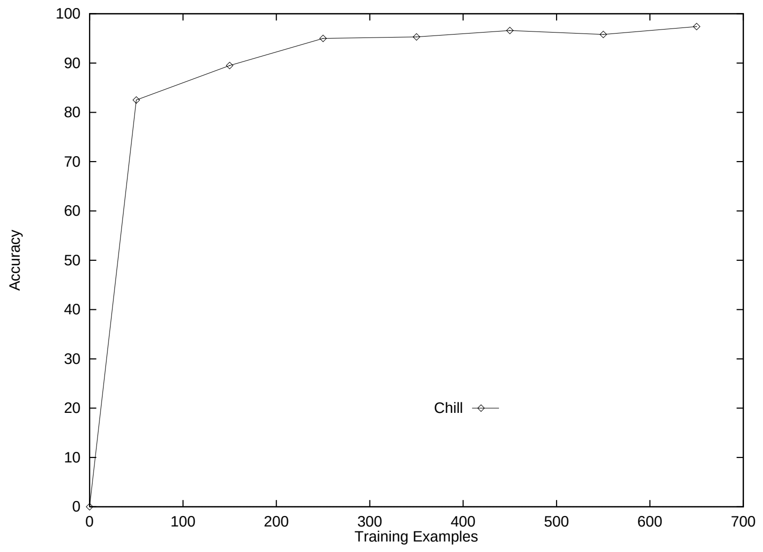

Figure 7: Learning Curve for Artificial Case-Role Corpus

Direct comparison with previous results is difficult, since the neural-network systems produce only a single parse for each sentence, and overall sentence accuracy is often not reported. The closest comparison can be made with the results of Miikkulainen and Dyer (1991) where an accuracy of 95% was achieved in assigning words to case slots after training with 1439 of the 1475 pairs. Chill produces totally correct analyses for a greater percentage of complete sentences with far less training data and CPU time.

It is also noteworthy that CHILL consistently invented interpretable word classes. One example, the invention of `animate`, has already been presented. This concept is implicit in the analyses presented to the system, since only animate objects are assigned to the agent role. Other invented classes clearly picked up on the distribution of words in the input sentences. The system regularly invented semantic classes such as `human`, `food`, and `possession` which were used for noun generation in the corpus.

Phrase classes useful to making parsing distinctions were also invented. For example, the structure `instrumental_phrase` was invented as:

```
instr_phrase([]).
instr_phrase([with, the, X]) :- instrument(X).
instrument(fork). instrument(bat). ...
```

Where the class, `instrument` was itself an invented category. It was not necessary in parsing to distinguish between instruments of different verbs, hence instruments of various verbs such as hitting and eating are grouped together. Where the semantic relationship between words is required to make parsing distinctions, such relationships can be learned. CHILL created one such relation: `can_possess(X,Y) :- human(X), possession(Y)`; which reflects the distributional relationship between humans and possessions present in the corpus. Notice that this invented rule itself contains two invented word categories.

5.2 Syntactic Parsing of the ATIS Corpus

In another set of experiments, CHILL was used to generate syntactic parsers from an extant treebank. The purpose of our experiments was to investigate whether the approach was sufficiently general and robust to handle syntactic parsing of realistic data. There are two facets to this question, first is whether the parsers learned by CHILL generalize well to novel text. An additional issue is whether the induction mechanism can efficiently process the numbers of examples required to achieve adequate performance on relatively large, complex corpora.

5.2.1 Applying CHILL to ATIS

We selected as our test corpus a portion of the ATIS dataset from a preliminary version of the Penn Treebank (specifically, the sentences in the file `ti_tb`)(Marcus et al., 1993). We chose this particular data because it represents realistic input from human-computer interaction, and because it has been used in several other studies on automated grammar acquisition (Brill, 1993; Periera & Shabes, 1992) that can serve as a basis for comparison to Chill. The corpus contains 729 sentences with an average length of 10.3 words.

```
s:[np:[*],
   vp:[show,
       np:[me],
       np:[np:[np:[the, flights],
               sbar:[that,
                     s:[np:[t],
                        vp:[served,
                            np:[lunch]]]]],
           vp:[departing,
               pp:[from,
                   np:[san, francisco]],
               pp:[on,
                   np:[april, '25th']]]]]]
```

Figure 8: Example ATIS Treebank Analysis

Experiments were actually performed with three variations of this corpus in order to test the benefit of adding information in the form of part-of-speech tags. The three versions were: untagged, tagged and tags-only. In the untagged corpus, words appeared in sentences without any attached part-of-speech labeling. The tagged version associated part-of-speech tags with each word. Finally, the tags-only version replaced words with their corresponding tags; parsing of tag sequences is common in previous corpus-based syntactic parsing.

The ATIS corpus concerns queries regarding air-travel information. An example analysis of the sentence, "Show me the flights that served lunch departing from San Francisco on April 25th" is shown in Figure 8. The analysis is a fairly sophisticated syntactic parse tree representation. As illustrated by this example, analyses may contain various types of empty constituents such as the implied subject of a command or the trace left by NP movement.

One complication in using this data is that sentences are parsed only to the phrase level, leaving the internal structure of NPs unanalyzed and allowing arbitrary-arity constituents. Rather than forcing the parser to learn reductions for arbitrary length constituents, Chill was restricted to binary-branching structures. This simplifies the parser and allows for a more direct comparison to previous experiments that use binary bracketings (Brill, 1993; Periera & Shabes, 1992).

Making the treebank analyses compatible with the binary parser required completion of the parses into binary-branching structures. This was accomplished by automatically introducing special internal nodes in a right-linear fashion. For example, the noun-phrase, `np:[the,big,orange,cat]`, would be elaborated as: `np:[the,int(np):[big, int(np):[orange, cat]]]`. The special labels (`int(np)` for noun phrases, `int(s)` for sentences, etc.) permits restoration of the original structure by merging internal nodes. Using this technique, the resulting parses can be compared directly with treebank parses.

The learning component of Chill remained exactly the same as in the case-role experiments except that the initial Parsing Operator Generator was modified to produce operators appropriate for the syntactic analyses of the Penn Treebank. As in case-role parsing, building an overly-general parser from a set of training examples is accomplished by constructing clauses for the `op/4` predicate. As an example, consider a phrase, $[_{np}[_{np}$a trip$]$ $[_{pp}$to $[_{np}$ dallas$]]]$. We represent such an analysis as a Prolog term of the form: `np:[np:[a, trip], pp:[to, np:[dallas]]]`. The operations and associated clauses required to parse the phrase are as follows (the notation, **reduce(**N**)** *Cat*, indicates that the top N stack elements are combined to form a constituent with label, *Cat*):

reduce(2) pp:

```
    op([S1,S2|Ss], Words, [pp:[S2,S1]|Ss], Words).
```

reduce(2) np:

```
    op([S1,S2|Ss], Words, [np:[S2,S1]|Ss], Words).
```

reduce(1) np:

```
    op([S1|Ss], Words, [np:[S1]|Ss], Words).
```

shift: `op(Stack, [Word|Words], [Word|Stack], Words).`

If the sentence analyses include empty categories (detectable as lexical tokens that appear in the analyses, but not in the sentence), each empty marker is introduced via its own shift operator which does not consume a word from the input buffer.

Our initial experiments used this simple representation of parsing actions (Zelle & Mooney, 1994b). However, improved results were obtained by specializing the operators, effectively increasing the number of operators, but reducing the complexity of the control-rule induction task for each operator. The basic idea was to index the operators based on some relevant portion of the parsing context. For example, in the experiments where lexical tags were used, the operators were indexed according to the syntactic category at the front of the input buffer. A single operator like `op(Stack, [Word|Words], [Word|Stack], Words)` becomes multiple operators in slightly differing contexts such as:

```
    op(Stack, [det|Ws], [det|Stack], Ws)
    op(Stack, [nn|Ws], [nn|Stack], Ws)
    op(Stack, [to|Ws], [to|Stack], Ws)
    op(Stack, [np|Ws], [np|Stack], Ws)
```

In experiments without lexical categories, the operators were indexed according to the top two phrase categories on the stack. Words which had not yet been reduced to a phrase were indexed as having the category `word`.

Since, the corpus provides a single preferred parse for each sentence, Chill was applied in single-parse mode, as explained in section 3.3. The operators were placed in order of increasing frequency of use as determined from the training set. This allows for learning control rules that take advantage of defaults, where the typically simpler conditions for applying more exceptional operators are tested first and control falls through to more generally applicable rules by default.

5.2.2 ATIS Results

Obviously, the most stringent measure of accuracy is the proportion of test sentences for which the produced parse tree exactly matches the treebanked parse for the sentence (*exact-match* accuracy). Sometimes, however, a parse can be useful even if it is not perfectly accurate. A parse with a few incorrect attachments or labelings may still convey the information required for some tasks. Indeed, the treebank itself is not completely consistent in the handling of some structures.

To better gauge the partial accuracy of the parser, we adopted a procedure for returning and scoring partial parses. If the parser runs into a "dead-end" while parsing a test sentence, the contents of the stack at the time of impasse is returned as a single, flat constituent labeled S. Since the parsing operators are ordered and the shift operator is invariably the most frequently used, shift serves as a default when no reduction action applies. Therefore, at the time of impasse, all of the words in the sentence will be on the stack, and partial constituents will have been built. The contents of the stack reflect the partial progress of the parser in finding constituents.

The resulting parse is then scored based on the overlap between the computed parse and the correct parse as recorded in the treebank. Two constituents are said to match if they span exactly the same words in the sentence. If constituents match and have the same label, then they are identical. The overlap between the computed parse and the correct parse is computed by trying to match each constituent of the computed parse with some constituent in the correct parse. If an identical constituent is found, the score is 1.0, a matching constituent with an incorrect label scores 0.5. The sum of the scores for all constituents is the overlap score, O, for the parse. The *partial-match* accuracy of the parse is then computed as $(\frac{O}{Found} + \frac{O}{Correct})/2$ where $Found$ is the number of constituents in the computed parse, and $Correct$ is the number of constituents in the correct tree. The result is an average of the proportion of the computed parse that is correct and the proportion of the correct parse that was actually found.

Another accuracy measure, which has been used in evaluating systems that bracket the input sentence into unlabeled constituents, is the proportion of constituents in the generated parse that do not cross any constituent boundaries in the correct tree (Black & et. al., 1991). Of course, this measure only allows for

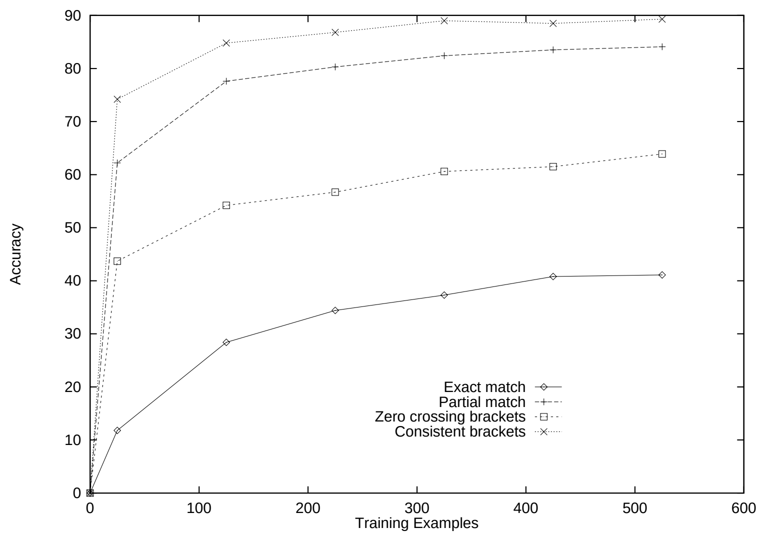

Figure 9: Tags-only ATIS Learning Curves

direct comparison of systems that generate binary-branching parse trees.[5] After ensuring that the output of the parser is binary-branching using the method described above, we computed the percentage of sentences with parses containing no crossing constituents (*zero crossing brackets* accuracy), as well as the proportion of constituents that are non-crossing over all test sentences (*consistent brackets* accuracy). This gives a basis of comparison with previous bracketing results, although it should be emphasized that CHILL is designed for the harder task of actually producing complete labeled parses, and is not directly optimized for the bracketing task.

Learning curves for the ATIS corpus showing these various accuracy measures are shown in Figures 9 and 10. Results are averaged over five random training/test splits. The learning curves for the tagged and tags-only versions were not significantly different, so only the latter results are shown.

The results for the tags-only version are very encouraging. After training on 525 sentences, CHILL constructed a parser comprising over 1500 lines of Prolog that generated completely correct parses for 41% of the novel testing sentences. Using the partial-match metric, CHILL'S parses garnered an average accuracy of over 84%. Even though CHILL is designed for the much more difficult task of building complete, labeled

[5]A tree containing a single, flat constituent covering the entire sentence always produces a perfect crossing score.

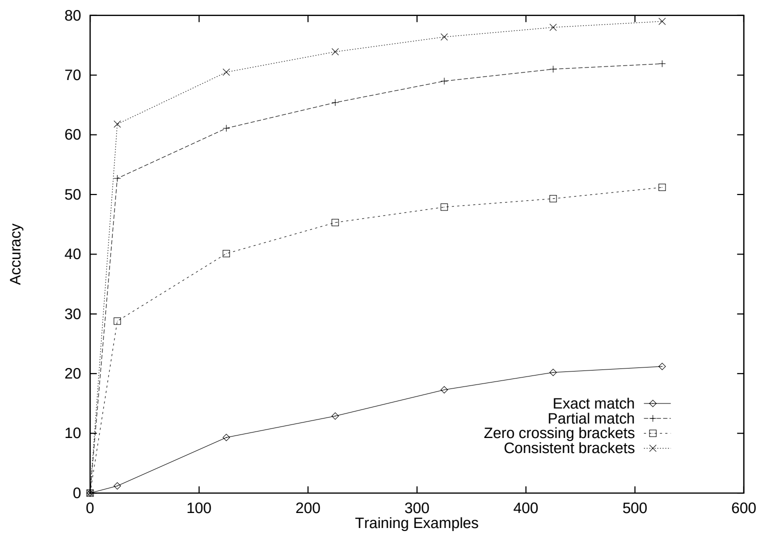

Figure 10: Untagged ATIS Learning Curves

parses, the figures for zero crossing brackets and consistent brackets compare favorably with those reported in previous studies with the ATIS corpus. Brill (1993) reports 60% and 91%, respectively. CHILL scores higher on zero crossing brackets (64%) and slightly lower (90%) on consistent brackets. This is understandable as Brill's transformation learner tries to optimize the latter value, while CHILL's preference for sentence accuracy might tend to improve the former (since correctly parsed sentences have no crossing brackets). CHILL's bracketing figures are also very similar to the recent results reported by Goodman (1996a) in his replication study of two stochastic grammar approaches with the ATIS corpus. With a larger set of 700 training examples, he also reports 64% zero crossing brackets and 90% consistent brackets for the approach of Periera and Shabes (1992) and 66% zero crossing and 90% consistent for the approach of Bod (1993). [6]

A comparison with the curves for the untagged version show that considerable advantage is gained by using word-class tags, rather than the actual words. This is to be expected as tagging significantly reduces the variety in the input. The results for untagged use no special mechanism for handling previously unseen words occurring in the testing examples. Achieving 72% partial match accuracy under these conditions

[6]Note that Goodman was unable to reproduce the higher accuracies originally reported by Bod and attributes Bod's results to an extremely fortuitous choice of test set. Goodman, like us and unlike authors of the previous studies, averages over multiple training/test splits and underlines the importance of this methodology.

seems quite good. Statistical approaches relying on n-grams or probabilistic context-free grammars would have difficulty due to the large number of terminal symbols (around 400) appearing in the modest-sized training corpus. The data for lexical selection would be too sparse to adequately train the pre-defined models. Similarly, the transformational approach of (Brill, 1993) is limited to bracketing strings of lexical classes, not words. CHILL's ability to learn something useful in the untagged rests on the learning mechanism's ability to automatically construct and attend to just those features of the input that are most useful in guiding parsing. Still, it is clear that considerably more training data would be required to significantly improve accuracy in the untagged case.

It should also be noted that the system created new categories in both situations. In the untagged experiments, CHILL regularly created categories for `preposition, verb, form-of-to-be`, etc.. With tagged input, various tags were grouped into classes such as the verb and noun forms. In both cases, the system also formed numerous categories and relations that seemed to defy any simple linguistic explanation. Nevertheless, these categories were apparently helpful in parsing of new text.

Beyond the created word categories, it is difficult to ascertain significant linguistic insight from the rules learned by CHILL in these experiments. The resulting rule-bases were very large (thousands of lines of Prolog code), and represented choices made at a relatively low level (that of control decisions, not grammar rules). Furthermore, the deterministic, single-parse framework used in these experiments produced sets of rules that were highly inter-related and context dependent.

The ATIS study shows that CHILL achieves results comparable to those of systems designed specifically for learning syntactic bracketings. Somewhat surprisingly, this is accomplished by learning a more traditional symbolic, deterministic parser rather than a probabilistic grammar. The learned parser is particularly efficient since, unlike probabilistic parsers, it performs no real search through the space of possible parses. However, the current computational requirements of ILP training prevents testing the existing system on huge corpora of long sentences, (e.g. the Wall Street Journal portions of the Penn Treebank). However, the construction of large, broad-coverage parsers for relatively shallow representations is not the only, nor perhaps even the most useful, application of empirical techniques. Syntactic parsing is only a small component of the larger understanding problem. In practice, most natural language systems are concerned with deeper, more semantically-oriented issues. Another application of empirical techniques is in developing more specialized parsers for mapping sentences in domain-specific sublanguages into useful meaning representations. The next subsection addresses this possibility.

5.3 Parsing Database Queries into Logical Form

Evaluating parser acquisition systems using artificial metrics such as matching treebank analyses is open to a number of criticisms. While the metrics are certainly useful in making rough comparisons, the results must be interpreted with caution. Often systems are not run on identical corpora, and even when the corpora are the same, they may be "cleaned up" in different ways (Goodman, 1996a). It is also unclear how one should compare systems that are tuned for optimizing different metrics (e.g., focusing on bracketings rather than labeled parse trees) (Goodman, 1996b). It is even less clear that these measures accurately reflect actual differences in performance of real language processing tasks. It is not necessarily the case that a system scoring 80% on some parsing metric will produce better final application results than one achieving 70%. The acid test is whether an empirical approach allows for the construction of better natural language systems, or perhaps allows designers to build comparable systems with less time and expertise.

In this section, we report on experiments using CHILL to engineer a natural language front-end for a simple database. The database task provides a good metric as it is easily evaluable. The gold standard is simply whether the system produces the correct output for a given question, a determination that is straight-forward for many database domains. There is no need to fret about the accuracy of partial metrics or engage in philosophical debates over the usefulness of various intermediate representations.

5.3.1 Overview of the Problem

For the database-query task, the input to CHILL consists of sentences paired with executable database queries. The query language considered here is a logical form similar to the types of meaning representation typically produced by logic grammars (Warren & Pereira, 1982; Abramson & Dahl, 1989). The semantics of the representation is grounded in a query interpreter that executes queries and retrieves relevant information from the database. The choice of a logical query language rather than the more ubiquitous SQL was made because the former provides a more straight-forward, compositional mapping from natural language utterances—a property that is necessary for the CHILL approach. Of course, once a sentence has been parsed into an

What is the capital of the state with the largest population?

```
answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).
```

What are the major cities in Kansas?

```
answer(C, (major(C), city(C), loc(C,S), equal(S,stateid(kansas)))).
```

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), equal(S, stateid(iowa)))).
```

Figure 11: Sample Database Queries

unambiguous logical form, translation into another database-query formalism should be easily automatable.

The domain of the chosen database is United States geography. The choice was motivated by the availability of an existing natural language interface for a simple geography database. This system, called *Geobase* was supplied as an example application with a commercial Prolog available for PCs, specifically Turbo Prolog 2.0 (Borland International, 1988). Having such an example provides a database already coded in Prolog for which a front-end can be built; it also serves as a convenient benchmark against which CHILL's performance can be compared.

The Geobase data contains about 800 Prolog facts asserting relational tables for basic information about U.S. states, including: population, area, capital city, neighboring states, major rivers, major cities, and highest and lowest points along with their elevation. The database also contains information concerning the lengths of rivers and the population of cities.

Figure 11 shows a sample of questions in English and the associated query representations. The queries are expressed using the Prolog conventions of capitalized identifiers representing logical variables and commas representing conjunction.

Development of the database application required work on two components: a framework for parsing into the logical query representations, and a specific query language for the geography database. The first component is domain-independent and consists of algorithms for parsing operator generation and example analysis to infer the required operators and parse the training examples. The resulting parsing framework is quite general and could be used to generate parsers for a wide range of logic-based representations.

The second component, which is domain specific, is a query language having a vocabulary sufficient for expressing interesting questions about geography. The database application itself comprises a parser produced by Chill coupled with an interpreter for the query language. The specific query language for these experiments (hereafter referred to as *Geoquery*) was initially developed by considering a sample of 50 sentences. A simple query interpreter was developed concurrently with the query language, thus insuring that the representations were grounded in the database-query task.

Once the query language and parsing framework were designed, a corpus of sentence/query pairs was developed by having uninformed subjects generate sample questions for the system. An analyst then constructed an appropriate query for each question, resulting in a corpus of 250 pairs which was used to evaluate the performance of Chill on this task.

5.3.2 The Query Language, Geoquery

The query language is basically a first-order logical form augmented with some higher-order predicates or *meta-predicates*, for handling issues such as quantification over implicit sets. This general form of representation is useful for many language processing tasks. The particular constructs of Geoquery, however, were not designed around any notion of appropriateness for representation of natural language in general, but rather as a direct method of compositionally translating English sentences into unambiguous, logic-oriented database queries. Thus Geoquery is an example of an task-specific meaning representation language.

The most basic constructs of the query representation are the terms used to represent the objects referenced in the database and the basic relations between them. The basic forms are listed in Figure 12. The objects of interest are states, cities, rivers and places (either a high-point of low-point of a state). In first-order representations, such objects would typically be represented by unique constants; however, it is easier to treat these objects as terms, for example using `stateid(texas)` to represent the state of Texas. This has the effect of "typing" the basic objects and making it easier to remember the representation for potentially ambiguous names (e.g. `stateid(missouri)` vs. `riverid(missouri)`). Cities are represented

Type	Form	Example
country	`countryid(CountryName)`	`countryid(usa)`
city	`cityid(CityName, StateAbbrev)`	`cityid(austin,tx)`
state	`stateid(StateName)`	`stateid(texas)`
river	`riverid(RiverName)`	`riverid(colorado)`
place	`placeid(PlaceName)`	`placeid(pacific)`

Figure 12: Basic Objects in Geoquery

using a two argument term with the second argument containing the abbreviation of the state. This is done to insure uniqueness, since different states may have cities of the same name (e.g. `cityid(columbus,oh)` vs. `cityid(columbus,ga)`). This convention also allows a natural form for expressing partial information; a city known only by name is given an uninstantiated variable for its second term.

The basic relations are shown in Figure 13. The use of these relations should be self-evident. One possible exception is the relation *equal/2*. It is used to indicate that a certain variable is bound to a ground term representing an object in the database. For example, a phrase like "the capital of Texas" translates to `(capital(S,C), equal(S, stateid(texas)))` rather than the more traditional `capital(stateid(texas),C)`. The use of `equal` allows objects to be introduced at the point where they are actually named in the sentence.

Although the basic predicates provide most of the expressiveness of Geoquery, meta-predicates are required to form complete queries. A list of the implemented meta-predicates is shown in Figure 14. These predicates are distinguished in that they take completely-formed conjunctive goals as one of their arguments.

The most important of the meta-predicates is `answer/2`. This predicate serves as a "wrapper" for query goals indicating the variable whose binding is of interest (i.e. answers the question posed). Executing a query of the form: `answer(X,Goal)` where `X` is a variable appearing in `Goal` results in a listing of all the unique values taken on by `X` for all possible proofs of `Goal` generated through backtracking. Many queries do not require any other meta-predicates for their expression. The other meta-predicates provide for the quantification over and selection of extremal elements from implicit sets.

Form	Predicate
`capital(C)`	`C` is a capital (city).
`city(C)`	`C` is a city.
`major(X)`	`X` is major.
`place(P)`	`P` is a place.
`river(R)`	`R` is a river.
`state(S)`	`S` is a state.
`capital(C)`	`C` is a capital (city).
`area(S,A)`	The area of `S` is `A`.
`capital(S,C)`	The capital of `S` is `C`.
`equal(V,C)`	variable `V` is ground term `C`.
`density(S,D)`	The (population) density of `S` is `P`
`elevation(P,E)`	The elevation of `P` is `E`.
`high_point(S,P)`	The highest point of `S` is `P`.
`higher(P1,P2)`	The elevation of `P1` is greater than that of `P2`.
`loc(X,Y)`	`X` is located in `Y`.
`low_point(S,P)`	The lowest point of `S` is `P`.
`len(R,L)`	The length of `R` is `L`.
`next_to(S1,S2)`	`S1` is next to `S2`.
`size(X,Y)`	The size of `X` is `Y`.
`traverse(R,S)`	`R` traverses `S`.

Figure 13: Basic Predicates in Geoquery

Form	Explanation
`answer(V,Goal)`	`V` is the variable of interest in `Goal`.
`largest(V, Goal)`	`Goal` produces only the solution maximizing size of `V`
`smallest(V,Goal)`	Analogous to `largest`.
`highest(V,Goal)`	Analogous to `largest` (with elevation).
`lowest(V,Goal)`	Analogous to `highest`.
`longest(V,Goal)`	Analogous to `largest` (with length).
`shortest(V,Goal)`	Analogous to `longest`.
`count(D,Goal,C)`	`C` is count of unique bindings for `D` satisfying `Goal`.
`most(X,D,Goal)`	`Goal` produces only the `X` maximizing count of `D`
`fewest(X,D,Goal)`	Analogous to `most`.

Figure 14: Meta-Predicates in Geoquery

5.3.3 A Parsing Framework for Queries

Although the logical representations of Geoquery look very different from parse-trees or case-structures discussed in previous sections, they are amenable to the same general parsing scheme as that used for the shallower representations. Adapting Chill to work with this representation requires only the identification and implementation of suitable operators for the construction of Geoquery-style analyses.

Logical queries are built in the shift-reduce framework using three simple operator types. Initially, a word or phrase at the front of the input buffer suggests that a certain structure should be part of the result. The appropriate structure is pushed onto the stack. For example, the word "capital" might cause the `capital/2` predicate to be pushed on the stack. This type of operation is performed by an `introduce` operator and is very similar to the simple shift operation employed for parse-tree analyses (A direct analogue of "shift" is still used to keep track of the actual words of the sentence. This will be explained in the following example). Initially, the logical structures are introduced with new (not co-referenced) variables. These variables may be unified with variables appearing in other stack items through a `co-reference` operator. For example, the first argument of the `capital/2` structure may be unified with the argument of a previously introduced `state/1` predicate. Finally, a stack item may be embedded into the argument of another stack item to form conjunctive goals inside of meta-predicates; this is performed by a `conjoin` operation.

Figure 15 shows the sequence of states the parser goes through in parsing the sentence, "What is the capital of Texas?" The state of the parser is shown as a term of the form: `ps(Stack,Input)` where `Stack` is a list of constituents comprising the current stack and `Input` are the remaining words of the input buffer. The individual stack items are shown one per line, and the state of the input buffer is on the last line of each parse state. Each stack item contains a portion of the query structure being built and a list of the words that have been shifted from the input while the stack item has been at the top of the stack. This is done so that the actual words used to introduce various structures are available to serve as context for forming control rules. The word list is maintained in reverse order and packaged with the query structure via the `:/2` functor.

The query extracted from state 14 is: `answer(A, (capital(B, A), equal(B, stateid(texas))))`. The parse sequence illustrates all of the basic operation types that are used to construct queries from

```
         Parse State                                                      Operation Type

1.  ps([answer(_,_):[]],
       [what,is,the,capital,of,texas,?])                                  shift
2.  ps([answer(_,_):[what]],
       [is,the,capital,of,texas,?])                                       shift
3.  ps([answer(_,_):[is,what]],
        [the,capital,of,texas,?])                                         shift
4.  ps([answer(_,_):[the,is,what]],
       [capital,of,texas,?])                                              introduce
5.  ps([capital(_,_):[],
        answer(_,_):[the,is,what]],
       [capital,of,texas,?])                                              co-reference
6.  ps([capital(_,A):[],
        answer(A,_):[the,is,what]],
       [capital,of,texas,?])                                              shift
7.  ps([capital(_,A):[capital],
        answer(A,_):[the,is,what]],
       [of,texas,?])                                                      shift
8.  ps([capital(_,A):[of,capital],
        answer(A,_):[the,is,what]],
       [texas,?])                                                         shift/introduce
9.  ps([equal(_,stateid(texas)):[texas],
        capital(_,A):[of,capital],
        answer(A,_):[the,is,what]],
       [?])                                                               co-reference
10. ps([equal(B,stateid(texas)):[texas],
        capital(B,A):[of,capital],
        answer(A,_):[the,is,what]],
       [?])                                                               conjoin
11. ps([equal(B,stateid(texas)):[texas],
        answer(A,capital(B,A)):[the,is,what]],
       [?])                                                               shift
12. ps([equal(B,stateid(texas)):[?,texas],
        answer(A,capital(B,A)):[the,is,what]],
       [])                                                                shift
13. ps(['EndOfInput',
        equal(B,stateid(texas)):[?,texas],
        answer(A,capital(B,A)):[the,is,what]],
       [])                                                                conjoin
14. ps(['EndOfInput',
        answer(A,(capital(B,A),
                  equal(B,stateid(texas)))):[the,is,what]],
       [])
```

Figure 15: Sequence of Parse States for “What is the capital of Texas?”

sentences. The initial state consists of the `answer/2` structure on the stack and the input buffer containing the sentence.

The most common operation is a `shift`, which simply transfers a word from the input buffer to the word list of the top item on the stack. This operation accounts for the results in states 2, 3, 4, 7, 8, 12, and 13 of the example parse. In the case of state 13, an attempt to shift from an empty input buffer puts a special end-of-input marker on the stack. Query structures are introduced at the point where their presence is indicated by lexical items at the front of the input buffer. For example, state 5 shows the result of the word "capital" introducing the `capital/2` structure. State 9 results from a special operator combining a `shift` with an `introduce` to handle the state-name at the front of the buffer. Parsing variables are unified via co-referencing operators, as demonstrated in states 6 and 10. The `conjoin` operator embeds one stack item into the argument of another. States 11 and 14 are the product of `conjoin` operations.

For each class of operator, the overly-general operators required to parse any given example may be easily inferred. The necessary `introduce` operators are determined by examining what structures occur in the given query and which words that can introduce those structures appear in the training sentence. `Co-reference` operators are constructed by finding the shared variables in the training queries; each sharing requires an appropriate operator instance. Finally, `conjoin` operations are indicated by the term-embedding exhibited in the training examples.

Using this simple framework, logical queries can be constructed in a compositional fashion using shift-reduce style parsing. It is important to note that only the operator generation phase of CHILL is modified to work with this representation; the control-rule learning component remains unchanged.

5.3.4 Experimental Results

Sample questions in English were obtained by distributing a questionnaire to 50 uninformed subjects. The questionnaire provided the basic information from the online tutorial supplied for Geobase (the existing system), including a verbatim list of the type of information in the database, and sample questions that the system could answer. The subjects were then asked to write down fifteen questions that they expected the system to be capable of answering. A total of 484 questions were gathered. Of these, 284 were discarded, resulting in a corpus of 250 questions with an average length of 7.8 words. Of the 284 questions discarded, 261 were either unanswerable from the information provided in the database (e.g. "What is the most polluted river?"), or were exact duplicates of included questions. The remaining 23 sentences were potentially answerable, but required queries outside the scope of the Geoquery representation. As mentioned above, Geoquery was originally designed to cover a corpus of 50 sentences. The fact that over 90% of the relevant questions eventually gathered were representable is a metric of the relative generality of Geoquery representations.

For evaluation purposes, the corpus was split into training sets of 225 examples with the remaining 25 held-out for testing. CHILL was run using default values for various parameters. This test differed from previous ones in that some background knowledge was provided to the control-rule learning component. This background information comprised predicates from the database itself for recognizing state, city and river names.

Testing employed the most stringent standard for accuracy, namely whether the application produced the correct answer to a question. Each test sentence was parsed to produce a query. This query was then executed to extract an answer from the database. The extracted answer was then compared to the answer produced by the correct query associated with the test sentence. Identical answers were scored as a correct parsing, any discrepancy resulted in a failure. For queries that resulted in multiple answers, the order of solutions was considered unimportant. Figure 16 shows the accuracy of CHILL's parsers over a 10 trial average. The line labeled "Geobase" shows the average accuracy of the Geobase system on these 10 testing sets of 25 sentences. The curves show that CHILL outperforms the existing system when trained on 175 or more examples. In the best trial, CHILL's induced parser comprising 1100 lines of Prolog code achieved 84% accuracy in answering novel queries.

In this application, it is important to distinguish between two modes of failure. The system could either fail to parse a sentence entirely, or it could produce a query which retrieves an incorrect answer. The former case represents a "softer" failure, since the application can be smart enough to indicate the sentence was unparsable and request a paraphrase. The parsers learned by CHILL for Geoquery produced few spurious parses. Figure 17 shows the probability that a test sentence will produce a spurious parse as a function of training set size. Again, for training sets of 175 examples or more, CHILL outperforms the original Geobase interface.

The training time required to achieve these results was relatively small; CHILL was able to learn parsers

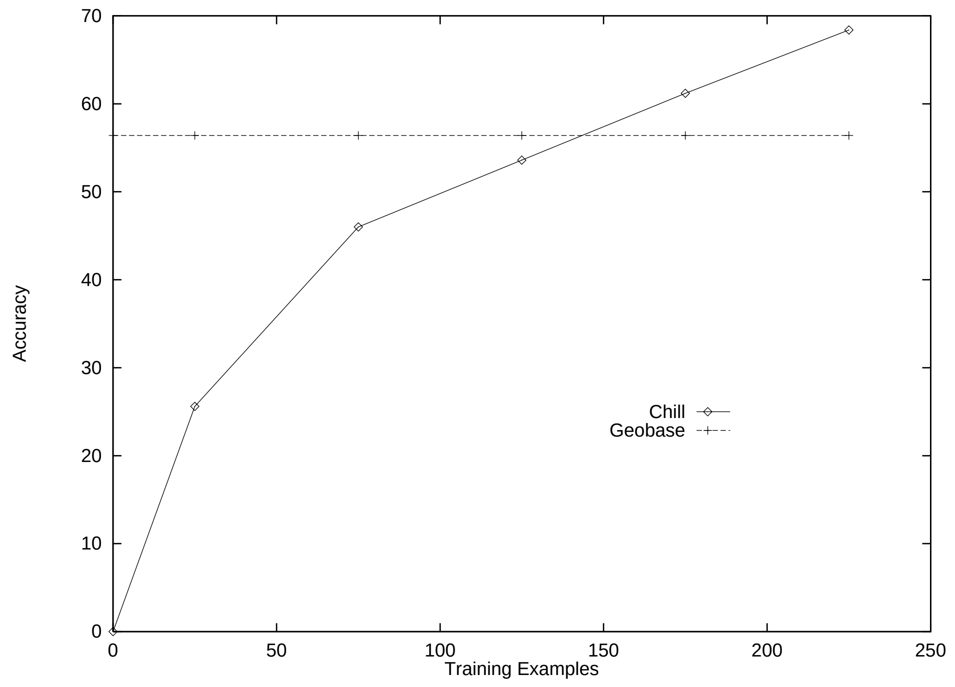

Figure 16: Geoquery: Accuracy

for the largest trials in less than 25 minutes of CPU time on a SPARCstation 5.

5.3.5 Discussion

While the Geobase system probably does not represent a state-of-the-art standard for natural language database query systems, neither is it a "straw-man." Geobase uses a semantics-based parser which scans for words corresponding to the entities and relationships encoded in the database. Rather than relying on extensive syntactic analysis, the system attempts to match sequences of entities and associations in sentences with an entity-association network describing the schemas present in the database. The result is a relatively robust parser, since many words can simply be ignored.

Using CHILL to construct a natural language application certainly does not eliminate the initial need for human expertise. The design of the query language is a nontrivial task and portions of the parsing "shell" must be filled-in with information from the database at hand. Part of this information included a lexicon relating words that appear in potential queries to the logical query structures used to express their

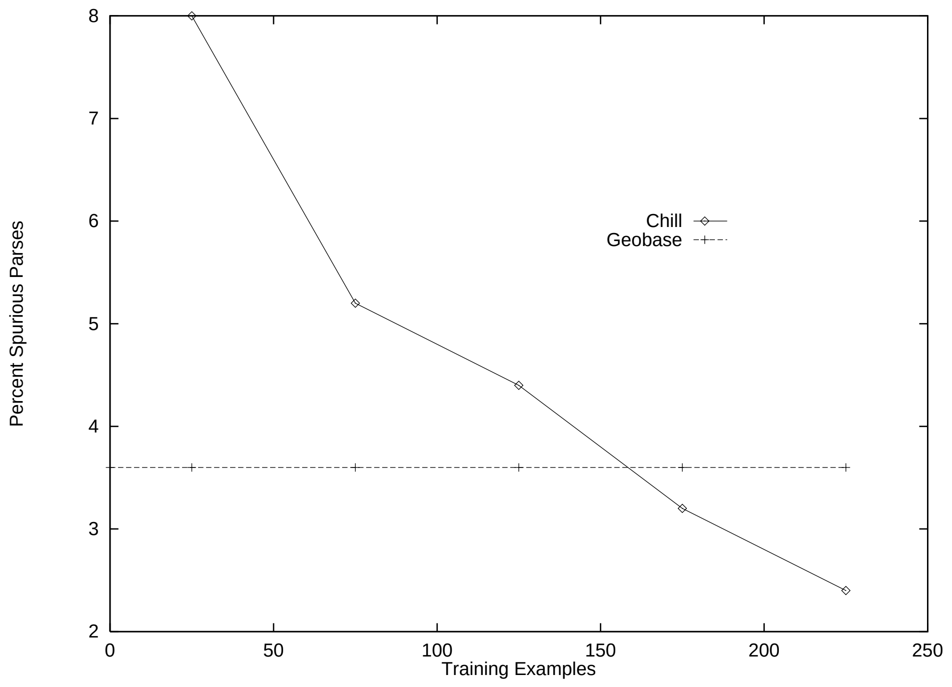

Figure 17: Geoquery: Percentage of Spurious Parses

"meaning." However, the design of these components relies strictly on the local considerations which arise in single examples. The problem of devising rules which are consistent across many examples is placed entirely on the learning component. Thus, the analyst is freed to concentrate on issues such as expressiveness of the representation rather than implementation of a grammar.

The learning curves for CHILL clearly suggest that training on larger corpora will improve CHILL's parsers. Similarly, traditional parser construction techniques involve the collection of examples and "tweaking" of the grammar to account for the new information. Often this is followed by extensive regression testing to insure that the changes did not damage performance on prior examples, since it is very hard to predict all the inter-related effects produced by isolated changes. The real promise of empirical techniques is that they might automate the grammar improvement step allowing construction of parsers that are consistent across a much larger range of natural language inputs than could be achieved with hand-crafted rules. Although the principle has been amply demonstrated for problems such as word-class tagging and syntactic analysis, these results suggest similar advantages at the level of a complete NL application.

6 Related Work

Although there has been a substantial amount of recent research in empirical approaches to parser construction, much of this work addresses somewhat different problems. One major difference is the type of analysis provided. CHILL learns parsers that produce complete, labeled parse trees; other systems have learned to produced simple bracketings of input sentences (Periera & Shabes, 1992; Brill, 1993), or probabilistic language models which assign sentences probabilities (Charniak & Carroll, 1994). Another dimension of variation is the type of input provided to the learning system. While CHILL requires only a suitably annotated corpus, other approaches have utilized an existing, complex, hand-crafted grammar that over-generates (Black et al., 1993; Black, Lafferty, & Roukaos, 1992). CHILL's ability to invent new categories also allows the use of actual words to make parsing decisions, whereas many systems are limited to representing sentences as strings of lexical categories (Brill, 1993; Charniak & Carroll, 1994).

The approach of Magerman (1994, 1995) is more similar to CHILL. His system produces parsers from annotated corpora of sentences paired with syntactic representations. Parsing is treated as a problem of statistical pattern recognition. This involves the coding of parse-tree topography with a finite set of construction features. Associated with each feature is a fixed set of parse-tree context information that is examined to determine the feature's value for a given node. The actual assignment of trees to sentences is performed by heuristic search through the space of possible parse-tree derivations guided by learned probabilistic decision trees. The learned grammars were shown to significantly outperform hand-crafted counterparts on a real-world parsing task involving text from technical manuals. CHILL differs from this approach mainly in its flexibility. Magerman's system is hand-engineered for the particular representation being produced. As an example, the parse-tree encoding scheme includes a feature for `conjunction` which was specifically introduced to improve the performance of the system. The system also includes a set of hand-generated rules for determining what properties a node in the parse tree will inherit from other nodes. Given this hand-crafting of features and rules, it is unclear how easily the approach could be adapted to differing representation schemes.

One approach that learns more semantically oriented representations is the hidden understanding models of Miller et al. (1994). This system learns to parse into tree-structured meaning representations. These representations are similar to syntactic parse trees except that the nodes may be labeled by conceptual categories as in the analyses produced by semantic grammars. The statistical model employs a separate component for determining what is said (the ordering of concepts) and how it is said (the choice of words). Each of these components is modeled with a probabilistic transition network. These networks are trained using extensions of standard statistical estimate-maximize algorithms. With a bootstrapping procedure which utilized the acquisition system to help annotate portions of the ATIS corpus, a single annotator was able to produce 200 annotated sentences a day. Training on 900 of these sentences produced a parser which achieved 61% exact-match accuracy on the remaining 100 sentences. This approach was recently extended to construct a complete interface with separate statistically trained modules for syntactic, semantic and discourse analysis (Miller, Stallard, Bobrow, & Schwartz, 1996). However, the mapping to a final semantic representation employs two separate modules, requiring each training sentence to be labeled with both a parse tree *and* a semantic frame. CHILL maps directly into logical form and does not require annotating sentences with any additional intermediate representations. An experimental comparison with CHILL would require running both systems on identical corpora; however, there are some other general differences worth noting. The hidden understanding model utilizes a propositional approach which forces it to make certain Markov-like assumptions. Thus, it is incapable of modeling phenomena requiring nonlocal references, a situation that does not hold for CHILL, which may examine any aspect of the parse context. A related limitation is that the ordering of concepts in tree-structured representations must match the order of words in the sentence. This makes it awkward to handle some forms of linguistic movement. In theory, CHILL can work with any representation for which suitable parsing operators can be inferred.

Kuhn and De Mori (1995) propose a method for learning semantic mappings using Semantic Classification Trees (SCTs). SCTs are variants of traditional decision trees in which nodes contain restricted regular expression representations that are matched against input queries. Semantic mapping is performed by translating each possible component of the desired output frame into a classification question. A forest of SCTs is then learned to determine the features that should be present in the output. The approach was demonstrated on a portion of the ATIS task, mapping inputs into frames that were then translated into SQL. The approach differs from CHILL on a number of dimensions. First, the input to the semantic component is not raw words, but minimally pre-parsed text, in this respect, it is similar to the multi-step approach of

Miller et al. (1996), although the parsing stage was hand-coded, and somewhat less ambitious. Second, the method requires the output mapping to be cast as a series of classification problems. This propositional restriction is sufficient for simple, flat SQL queries, but it is unclear how the technique would be applied to more sophisticated, nested queries.

Overall, Chill is unique in it's flexibility to employ the same general learning system to construct a variety of parsers without the need to engineer a fixed set of relevant features *a priori*. The same general system can construct parsers for producing syntactic parses, case analyses, and semantic logical forms given little more than a corpus of input/output pairs.

7 Future Work

Improving the accuracy of the parsers learned by Chill requires progress on two fronts. One obvious approach is to use much larger training sets. However, making this practical requires further improvements in the efficiency of the induction algorithm. The most difficult of the trials reported in this paper already required up to 8 hours of CPU time on a SPARCstation 2. Another avenue is to incorporate more language-specific biases into the induction process. The shift-reduce framework of the current system is a rather weak bias compared to restrictions commonly considered in a "principles-and-parameters" approach. A nice feature of ILP methods is that they employ a traditional symbolic representations and can exploit prior background knowledge. Our initial research has focussed on what could be accomplished by placing most of the burden on a very general learning component. Providing the system with background knowledge and tighter constraints could allow more accurate induction from fewer examples.

When parsing into logical form, Chill currently requires a semantic lexicon that maps words into predicates and/or terms that are then composed to produce a final output. We are currently developing techniques that automatically acquire this lexicon from the training corpus using a symbolic induction algorithm (Thompson, 1995). We are also exploring the use of ILP methods in constructing rules for *information extraction* (the task of identifying specific items in a natural language text (Lehnert & Sundheim, 1991)). ILP techniques can potentially induce more complex patterns than previous learning methods applied to this task (Riloff, 1996; Soderland, Fisher, Aseltine, & Lehnert, 1995). By inducing unbounded relational patterns that characterize the context surrounding particular phrases, concise rules for extracting information can be learned from a training corpus of filled templates (Califf & Mooney, 1997).

A significant problem for empirical NLP in general is corpus construction. Annotating large corpora with parse trees, logical forms, or filled templates, is a difficult and time consuming task. The use of *active learning*, specifically *selective sampling* (Cohn, Atlas, & Ladner, 1994; Lewis & Catlett, 1994), could potentially ease this problem. Not all training examples are equally useful to a learning system, and by carefully selecting useful examples, annotation effort can be dramatically reduced. This approach has been successfully employed in training part-of-speech taggers (Engelson & Dagan, 1996) and we intend to explore it's use in parsing and information extraction.

8 Conclusions

This paper has presented CHILL, a system that uses relational learning methods to induce traditional, shift-reduce parsers from corpora, complementing the results of recent statistical methods. We believe the primary strength of corpus-based methods is not the particular approach or type of parser employed (e.g., statistical, connectionist, or symbolic), but the fact that real data are used to automatically construct complex parsers. The presented experimental results suggest that CHILL performs comparably to previous approaches on similar tasks.

We believe that the main advantage of an inductive logic programming approach is the resulting flexibility. CHILL is capable of learning to parse into standard syntactic representations as well as more meaning-oriented case structures and logical forms for database querying. In addition, CHILL is able to learn using highly structured contexts and can automatically create new predicates necessary to support accurate parsing. These abilities reduce the need for the feature-engineering common in propositional approaches. Of course, this does not obviate the need for engineering representations suitable to a particular task, but once representations have been devised based on usefulness to the task, the learning component assumes the burden of identifying and exploiting relevant structural differences. A further potential attraction of the ILP approach is the ease with which it may be integrated with traditional, symbolic parsing methods. While this work has used a shift-reduce framework, other NLP work may also benefit from the direct application of relational learning methods.

Chill represents an alternative to the broad-but-shallow paradigm that dominates current work in empirical NLP. The results from the geography database application presented here suggest that empirical methods may also be useful in the rapid development of smaller, domain-specific parsers that produce useful final representations. For some tasks, it may be feasible to automatically construct virtually the entire linguistic component of a complex natural-language system. One could envision, for example, collecting a sample of database queries that are actually submitted to an information system in the course of normal use, pairing these with the English sentences that express them, and presenting these to a Chill-like system to generate a natural language front-end. Automated generation of natural language interfaces could be an exciting application of empirical techniques, but is only possible when the learning system is able to learn the mapping from strings of words into useful representations for the task at hand. We believe the techniques employed in Chill, namely parser acquisition as control-rule learning and relational learning algorithms are significant step in this direction. Chill is really just a starting point in the investigation of ILP learning techniques in natural language processing.

Acknowledgments

We would like to thank the anonymous reviewers for helpful comments on the initial version of this paper. This research was supported by the National Science Foundation under grant IRI-9310819 and the Texas Advanced Research Program under grant ARP-003658-114.

References

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

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

Black, E., & et. al. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the Fourth DARPA Speech and Natural Language Workshop*, pp. 306–311.

Black, E., Jelineck, F., Lafferty, J., Magerman, D., Mercer, R., & Roukos, S. (1993). Towards history-based grammars: Using richer models for probabilistic parsing. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pp. 31–37 Columbus, Ohio.

Black, E., Lafferty, J., & Roukaos, S. (1992). Development and evaluation of a broad-coverage probabilistic grammar of English-language computer manuals. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pp. 185–192 Newark, Delaware.

Bod, R. (1993). Using an annotated language corpus as a virtual stochastic grammar. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 778–783.

Borland International (1988). *Turbo Prolog 2.0 Reference Guide.* Borland International, Scotts Valley, CA.

Brill, E. (1993). Automatic grammar induction and parsing free text: A transformation-based approach. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pp. 259–265 Columbus, Ohio.

Briscoe, T., & Carroll, J. (1993). Generalized probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, *19*(1), 25–59.

Califf, M. E., & Mooney, R. (1997). Relational learning of pattern-match rules for information extraction. submitted.

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

Charniak, E., & Carroll, G. (1994). Context-sensitive statistics for improved grammatical language models. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* Seattle, WA.

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.

Cohen, W. W. (1995). Text categorization and relational learning. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 124–132 San Francisco, CA. Morgan Kaufman.

Cohn, D., Atlas, L., & Ladner, R. (1994). Improving generalization with active learning. *Machine Learning*, *15*(2), 201–221.

Engelson, S., & Dagan, I. (1996). Minimizing manual annotation cost in supervised training from corpora. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics* Santa Cruz, CA.

Fillmore, C. J. (1968). The case for case. In Bach, E., & Harms, R. T. (Eds.), *Universals in Linguistic Theory*. Holt, Reinhart and Winston, New York.

Goodman, J. (1996a). Efficient algorithms for parsing the DOP model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 143–152 Philadelphia, PA.

Goodman, J. (1996b). Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pp. 177–183 Santa Cruz, CA.

Hindle, D., & Rooth, M. (1993). Structural ambiguity and lexical relations. *Computational Linguistics*, *19*(1), 103–120.

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.

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.

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lehman, J. F. (1994). Toward the essential nature of satistical knowledge in sense resolution. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 734–741 Seattle, WA.

Lehnert, W., & Sundheim, B. (1991). A performance evaluation of text-analysis technologies. *AI Magazine*, *12*(3), 81–94.

Lewis, D. D., & Catlett, J. (1994). Heterogeneous uncertainty sampling for supervised learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 148–156 San Francisco, CA. Morgan Kaufman.

Magerman, D. M. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pp. 276–283 Cambridge, MA.

Magerman, D. M. (1994). *Natural Lagnuage Parsing as Statistical Pattern Recognition*. Ph.D. thesis, Stanford University.

Manning, C. D. (1993). Automatic acquisition of a large subcategorization dictionary from corpora. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pp. 235–242 Columbus, Ohio.

Marcus, M. (1980). *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, MA.

Marcus, M., Santorini, B., & Marcinkiewicz, M. (1993). Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, *19*(2), 313–330.

McClelland, J. L., & Kawamoto, A. H. (1986). Mechanisms of sentence processing: Assigning roles to constituents of sentences. In Rumelhart, D. E., & McClelland, J. L. (Eds.), *Parallel Distributed Processing, Vol. II*, pp. 318–362. MIT Press, Cambridge, MA.

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

Miikkulainen, R., & Dyer, M. G. (1991). Natural language processing with modular PDP networks and distributed lexicon. *Cognitive Science*, *15*, 343–399.

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.

Mooney, R. J., & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, *3*, 1–24.

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., King, R., & Sternberg, M. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, *5*(7), 647–657.

Muggleton, S. H. (Ed.). (1992). *Inductive Logic Programming*. Academic Press, New York, NY.

Periera, 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 generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence (Vol. 5)*. Elsevier North-Holland, New York.

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pp. 3–20 Vienna.

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

Riloff, E. (1996). An empirical study of automated dictionary construction for information extraction in three domains. *Artificial Intelligence*, *85*, 101–134.

Simmons, R. F., & Yu, Y. (1992). The acquisition and use of context dependent grammars for English. *Computational Linguistics*, *18*(4), 391–418.

Soderland, S., Fisher, D., Aseltine, J., & Lehnert, W. (1995). Crystal: Inducing a conceptual dictionary. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1314–1319.

Thompson, C. A. (1995). Acquisition of a lexicon from semantic representations of sentences. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pp. 335–337 Cambridge, MA.

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

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

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

Zelle, J. M., & Mooney, R. J. (1994a). 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. (1994b). Inducing deterministic Prolog parsers from treebanks: A machine learning approach. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 748–753 Seattle, WA.