Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers

by

John M. Zelle, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1995

Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers

Approved by
Dissertation Committee:

____

____

____

____

____

To my parents who started me on the road, and to Lib, without whom I never would have reached my destination.

Acknowledgments

I have been positively influenced by many people during the tenure of this research. I am deeply grateful for the significant contributions of my advisor, Ray Mooney. He has been a constant source of guidance and served as a perfect model of the dedicated researcher and educator. I do not believe I could have had a better mentor. I would also like to thank the other members of my committee: Risto Miikkulainen, Bruce Porter, Benjamin Kuipers and William Cohen. I have also benefited from contributions by a former committee member, the late Robert Simmons.

Many others have provided both ideas and moral support during my years at the University of Texas, especially: Paul Baffes, Jeff Mahoney, Dan Clancy, Tara Estlin, Dave Moriarty, Cindi Thompson, Ulf Hermjakob, Joshua Konvisser, Mary-Elaine Califf, Charles Calloway and James Lester. They have all been good friends and colleagues; I am glad to have had the chance to get to know them.

I am also deeply indebted to my family, who have always been supportive of my educational endeavors. I offer a special thanks to my wife, Elizabeth Bingham, whose love of learning brought me to Texas, and whose love and confidence in me kept me afloat when I wasn't sure I could make it. I am also thankful for my constant computer-side companion during the writing of this dissertation, Grendel—if only he would stay off the keyboard.

Portions of this research were supported by the National Science Foundation under grant IRI-9310819 and the Texas Advanced Research Program under grant ARP-003658-114. The author was also partially supported by a fellowship from the Schlumberger Corporation.

John M. Zelle

The University of Texas at Austin
August 1995

Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers

Publication No. ____________________

John M. Zelle, Ph.D.
The University of Texas at Austin, 1995

Supervisor: Raymond J. Mooney

Designing computer systems to understand natural language input is a difficult task. In recent years there has been considerable interest in corpus-based methods for constructing natural language parsers. These empirical approaches replace hand-crafted grammars with linguistic models acquired through automated training over language corpora. A common thread among such methods to date is the use of propositional or probabilistic representations for the learned knowledge. This dissertation presents an alternative approach based on techniques from a subfield of machine learning known as inductive logic programming (ILP). ILP, which investigates the learning of relational (first-order) rules, provides an empirical method for acquiring knowledge within traditional symbolic parsing frameworks.

This dissertation details the architecture, implementation and evaluation of Chill, a computer system for acquiring natural language parsers by training over corpora of parsed text. Chill treats language acquisition as the learning of search-control rules within a logic program that implements a shift-reduce parser. Control rules are induced using a novel ILP algorithm which handles difficult issues arising in the induction of search-control heuristics. Both the control-rule framework and the induction algorithm are crucial to CHILL's success.

The main advantage of CHILL over propositional counterparts is its flexibility in handling varied representations. CHILL has produced parsers for various analyses including case-role mapping, detailed syntactic parse trees, and a logical form suitable for expressing first-order database queries. All of these tasks are accomplished within the same framework, using a single, general learning method that can acquire new syntactic and semantic categories for resolving ambiguities.

Experimental evidence from both artificial and real-world corpora demonstrates that CHILL learns parsers as well or better than previous artificial neural network or probabilistic approaches on comparable tasks. In the database query domain, which goes beyond the scope of previous empirical approaches, the learned parser outperforms an existing hand-crafted system. These results support the claim that ILP techniques as implemented in CHILL represent a viable alternative with significant potential advantages over neural-network, propositional, and probabilistic approaches to empirical parser construction.

Contents

Chapter 1

Introduction

Computer understanding of natural (human) languages is one of the oldest and most enduring challenges in the field of artificial intelligence. Fundamental to the problem of computer understanding is the ability to translate or *parse* natural language inputs into an internal *meaning representation language* (MRL) that is suitable for computer manipulation.

Traditional work in natural language processing (NLP) has pursued a *rationalist* approach to the parsing problem, searching for perspicuous, rule-based representations of the knowledge required for language processing. Over time, a myriad of frameworks, from augmented transition networks to unification grammars, have been proposed for representing and computing with linguistic knowledge. (Allen, 1995; Gazdar & Mellish, 1989). Certainly, great progress has been made in understanding the fundamental issues involved in natural language understanding. Modern representations allow grammarians to model many aspects of language elegantly and facilitate the construction of grammars for domain-specific language processing applications.

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 "sub-language" applications is difficult and time-consuming, yielding systems which are often inefficient and incomplete. Additionally, successful systems often have an ad hoc quality and porting to new domains requires essentially starting from scratch. In short, NLP, as traditionally practiced, suffers from a "knowledge acquisition bottleneck." Partially in response to these difficulties, there has been increasing recent interest in *empirical* approaches to natural language processing.

1.1 Empirical NLP

The empirical alternative 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 tack will be the main focus of this dissertation, 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. The first task, annotation, is the province of human experts (perhaps with mechanical help). They must devise a training corpus which demonstrates the type of NL analysis that is required. For example, if the desired system is a broad-coverage 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 (not annotated) text for language acquisition, those employing annotations have proved more powerful (Periera & Schabes, 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 desired representation. With such an approach, the human burden is on engineering useful representations, relegating the difficult issue of constructing a parser for such representations to the acquisition system.

Following in the footsteps of speech recognition research, corpus-based natural language processing has concentrated primarily on statistical techniques (Charniak, 1993) applied to such problems as part-of-speech tagging (Merialdo, 1994; Charniak, Hendrickson, Jacobson, & Perkowitz, 1993) and the induction of stochastic context-free grammars (Periera & Schabes, 1992) or transition networks (Miller, Bobrow, Ingria, & Schwartz, 1994). 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, 1994) and transformations (Brill, 1993), statistical methods dominate.

A common thread in all of these approaches is that the acquired knowledge is represented in a *propositional* form (perhaps with associated probabilities). This means for example, 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 exact conditions of the rule(s) are determined by the acquisition algorithm but the context over which the rules are formed, and the exact properties which may 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 pre-chosen features of interest.

One might expect these simple, unstructured models to be effective with speech, an essentially linear phenomenon. However, it is somewhat surprising they they have been relatively successful in areas such as syntactic analysis which traditionally entail the construction and manipulation of indefinitely-large, highly-structured representations. There are probably two contributing factors to this success. First, the designers of such systems have invested considerable effort to simplify these problems by hand-crafting appropriate feature sets, and carefully identifying relevant, finite contexts over which learning occurs. Second, these systems may be trained on huge corpora, achieving accuracy by guaranteeing consistency across a far greater number of sentences than their hand-constructed, rationalist counterparts. The knowledge may be much more complete, even if the representation is not as powerful. Indeed, the chief weakness of traditional NLP has been the difficulty of engineering consistent grammars for large corpora.

This dissertation considers an empirical approach utilizing a *structured* knowledge representation. Relational representations have long been a tool of traditional NLP. Virtually all of this work has utilized hand-crafted grammars, as suitable methods for automating the construction of relational knowledge bases had not yet been developed. Now, however, a growing subfield of machine learning research called Inductive Logic Programming (ILP) addresses the problem of learning first-order logic descriptions (Prolog programs) (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). The research in this dissertation attempts to bridge the gap between the rational and empirical approaches to NLP by applying ILP to the problem of parser acquisition.

1.2 CHILL: An Empirical Parser Acquisition System

Chill (Constructive Heuristics Induction for Language Learning) is a general approach to the problem of inducing natural language parsers. The learning problem

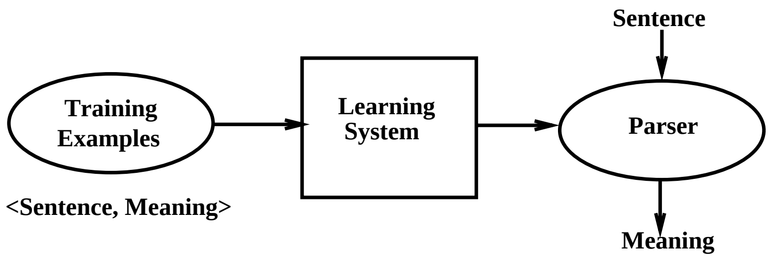

Figure 1.1: The Parser Acquisition Problem

addressed is depicted in Figure 1.1. Given a suitably annotated corpus, CHILL produces a parser for mapping subsequent sentences into representations. CHILL is unique among empirical approaches to date, in that it takes a broad view of the notion of parsing.

Strictly speaking, the term "parsing" is often used to indicate translation into a form which makes explicit the *syntactic* structure of a sentence as a hierarchy of labeled constituents. For example a sentence such as: "The man threw the ball" might be bracketed into its constituent noun- and verb-phrase as $[_S[_{NP}$The man] $[_{VP}$ threw $[_{NP}$ the ball]]]. 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. In practice, natural language systems are usually concerned with deeper, semantically-oriented issues. At a very minimum these systems need the ability to identify important relationships such as who did what to whom. Typically, such systems have employed more semantically oriented MRLs such as case-role representation, which exposes the argument structure of an utterance. For example, the sentence above might be parsed as: [threw, agent:[the man], patient:[the ball]], indicating that there was an act of throwing performed by a man acting on a ball. Other systems, especially those for database applications may parse sentences into logical forms which are manipulated by automated deduction algorithms. A simple first-order predicate calculus representation might be: man(m1) $\wedge$ ball(b1) $\wedge$ threw(m1,b1).

Parsing in Chill is taken in the broader sense of translating input sentences into whatever MRL is convenient for the application at hand. Chill has been used to learn parsers for syntactic parses, case-role parses, and first-order logical forms. This flexibility to learn using different representations is one of the major advantages of the approach.

Chill achieves this flexibility by treating parser induction as the problem of learning rules to control the actions of a shift-reduce parser expressed as a Prolog program. Control rules are induced utilizing a novel ILP algorithm that has been developed to handle the issues arising in the control-rule domain. The induction algorithm itself is a very general concept learning system that has been demonstrated on a wide-range of benchmark ILP problems. Given the power of first-order rules, there is less need to hand-engineer appropriate features and contexts over which Chill learns, as is required in propositional systems. The induction algorithm can automatically extract the relevant portions of structured contexts and construct new predicates to represent novel syntactic and/or semantic word and phrase categories that are necessary to perform accurate parsing. Parsers for alternative representations are learned by simply substituting new parsing frameworks; the learning component itself remains unchanged.

The Chill approach has been evaluated on a number of problems using both artificial and real-world corpora. Chill has been demonstrated by learning case-role parsers for artificial corpora previously used to demonstrate the language processing abilities of artificial neural networks and by"reverse-engineering" a semantic grammar for a much larger corpus represented in a database query system for tourist information. Syntactic parse-tree parsers have been learned from an existing tree-bank concerning air-travel information, a corpus that has been used to demonstrate previous statistical and transformational acquisition methods. These experiments demonstrate that Chill learns parsers as well or better than previous, propositional approaches on comparable tasks. Finally,Chill has been used to engineer a complete demonstration application, producing parsers that map questions concerning United States geography directly into executable database queries. On this problem, which goes beyond the scope of previous empirical systems, the learned parsers significantly outperform an existing hand-crafted system.

1.3 Organization of Dissertation

The rest of this dissertation details and discusses an initial implementation of Chill. The next chapter provides background information situating this work in relation to previous work in machine learning concerning control-rule learning and inductive logic programming. Chapter 3 gives a detailed description of Chill by way of a simple case-role parsing example. Chapter 4 presents the details of the ILP induction algorithm used in Chill. Chapter 5 presents experimental results demonstrating Chill's performance on case-role mapping tasks, while chapter 6 contains experiments concerning parsing the ATIS corpus from the Penn Treebank. Chapter 7 discusses experiments using Chill to develop a prototype database query front-end. Chapter 8 outlines closely related research. Chapter 9 discusses open questions and directions for future research, while Chapter 10 presents conclusions.

Chapter 2

Background

As mentioned in the introduction, Chill is based on ideas from two subfields of machine learning: control-rule learning and inductive logic programming. This chapter presents an introduction to some of the background research from these areas. The prupose of this material is to place Chill in its proper perspective against the background of other work in machine learning, and to lay the technical foundations required to understand the presentation of Chill in the following chapters.

2.1 Control-Rule Learning

Many AI problems may be usefully formalized as some type of search problem. However, weak search methods such as means-ends analysis are usually not sufficient to render complex real-world problems solvable. The control-rule learning subfield of machine learning addresses this shortcoming. The basic idea of control-rule learning is that, through experience, a problem solving system may improve its performance by discovering and using heuristics that suggest which search paths are likely to solve new problem instances. Learned search-control knowledge may improve both the eficiency and accruacy of knowledge-based systems. Efficiency is improved by eliminating search paths that do not lead to solutions. Such control rules act as heuristics to allow more efficient search in future problems. Accuracy can be improved by learning control rules which prune the search along paths leading to incorrect solutions. These control rules may be viewed as preconditions that were previously omitted from the search operators. Adding these preconditions results in a more accurate problem solver.

The learning of search-control knowledge has been investigated primarily in the context of STRIPS-like planners (Minton, 1988) and forward-chaining production systems (Mitchell, 1983; Langley, 1985; Laird, Rosenbloom, & Newell, 1986). The basic inputs to a control-rule learning system are an initial search-based problem solver, and a set of training problems. The output is an improved problem-solver. The learning process may be broken down into three basic phases. In the first phase, the training problems are solved with a version of the initial problem-solver. Since the original solver is weak, it may require guidance to solve the training problems. This guidance might come in the form of providing a solution path to the system, or perhaps providing a correct solution, from which a solution path can be inferred. With this solution path in hand, the learning system analyzes those places where the search process of the problem solver would have been led down erroneous paths. For each decision-point, the context of the decision is saved as a positive *control-example* for the option that lies on the solution path. This same context serves as a negative control-example for options which deviate from the solution path.

Once the system has uncovered positive and negative control examples for a given option, it must construct *control-rules* which characterize the correct option for any given decision point. A control-rule is a kind of heuristic that can be used to select the most useful option given the current context of a particular decision. Some systems learn control rules by using inductive (Simliarity Based Learning, or *SBL*) methods (Mitchell, Utgoff, & Banerji, 1983). An SBL learner would look at the postive and negative control examples for a given option, call it X, and attempt to induce a defintion of the concept such as "contexts in which X is useful." This concept would cover the positive control examples for option *X*, but not the negative ones. An alternative approach is to use analytic (Explanation Based Learning, or *EBL*) methods (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986). An EBL learner would analyze positive control examples in the context of a trace of the problem-solver to extract the specific features which contributed to the example's success. Some systems have employed a combination of SBL and EBL to learn control-rules (Mitchell, 1984; Cohen, 1990; Zelle & Mooney, 1993a).

In the final phase, the learned control rules must be integrated with the initial problem solver to produce an enhanced system. Intuitively, this means adding an evaluation at each decision point to select which option (or options) should be followed-up, based on the learned control-rules.

2.1.1 Learning Search-Control in Logic Programs

Some recent research has investigated the learning of search-control heuristics to improve the efficiency of problem solvers implemented as logic programs (Cohen, 1990; Zelle & Mooney, 1993a). A logic program is expressed using the *definite clause* subset of first-order logic. A definite clause is a disjunction of literals having exactly one unnegated literal, called the *head*. The negated literals comprise the clause *body*. Computation in logic programs is performed using a resolution proof strategy on an existentially quantified goal.

Taking an example from Zelle and Mooney (1993a), a program to sort lists might be expressed as a collection of definite clauses comprising a logical definition of the two-place predicate, `sort`. A goal of the form `sort(X,Y)` is taken to be true exactly when `Y` is a sorted version of the list represented by `X`. One possible defintion for this predicate is shown in Figure 2.1. When using a logic program to do computation, the arguments of the top-level goal are typically partitioned into *input* and *output* argument sets. The program is executed by providing a goal which has its input arguments instantiated. A theorem prover contructively proves the existence of

```
sort(X,Y) :-  permutation(X,Y), ordered(Y).

permutation([],[]).
permutation([X|Xs],Ys) :-  permutation(Xs,Ys1), insert(X,Ys1,Ys).

insert(X,Xs,[X|Xs]).
insert(X,[Y|Ys],[Y|Ys1]) :-  insert(X,Ys,Ys1).

ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Figure 2.1: Sorting Program

a goal meeting these constraints and produces bindings for the output arguments. In the sorting example, the first argument may be considered an input, and the second an output. Given the definition of `sort` shown, proving the goal `sort([9,1,5,3,4],Y)` produces the output binding: `Y = [1,3,4,5,9]`.

Prolog provides a particular implementation of logic programming using a very simple search strategy. Proofs are attempted by trying clauses in a depth-first fashion with simple (chronological) backtracking. Multiple solutions may be found for a given goal by asking the prover to backtrack and find alternative bindings for the output arguments. The notion of search-control in a Prolog program can be viewed as a *clause selection* problem (Cohen, 1990). Clause selection is the process of deciding which of several applicable program clauses should be used to reduce a particular subgoal during the course of a proof. If program clauses are always applied appropriately, the program executes deterministically (without backtracking) and produces only correct solutions.

In the sorting program shown, the default Prolog search strategy results in a very inefficient sorting algorithm. The program generates all possible permutations of the input list until one is found that happens to be in order. Permutations of a list are generated by first permuting the tail of the list and then inserting the head

Table 2.1: Examples of `useful_insert_1` from `sort([9,1,5,3,4],X)`

Positives	Negatives
`insert(9,[],A)`	`insert(9,[5],A)`
`insert(1,[3,4,5],A)`	`insert(9,[4,5],A)`
`insert(5,[],A)`	`insert(9,[3,4,5],A)`
`insert(3,[4],A)`	`insert(9,[1,3,4,5],A)`
`insert(4,[],A)`	`insert(5,[4],A)`
	`insert(5,[3,4],A)`

of the list somewhere in the permuted tail. Notice that the only choice point in this program occurs in the definition of `insert/2`.[1] Any `insert/2` subgoal with a non-empty second argument will match with either clause of the definition. Prolog always applies the first clause and only revises that choice upon backtracking caused by the final list not meeting the `ordered/1` condition.

The DOLPHIN system (Zelle & Mooney, 1993a) can automatically transform this $O(n!)$ sorting algorithm into one which runs in $O(n^2)$ time by learning from a single top-level example. Table 2.1 shows some control examples characterizing the usefulness of the first clause of `insert`; these control examples are generated by analyzing a trace of the program solving the goal `sort([9,1,5,3,4],X)`. The positive control examples are subgoals to which the first clause of `insert/2` was applied in the found proof. The negative examples represent subgoals for which the first clause was initially tried but later backtracked over in favor of the second clause. DOLPHIN, which uses a combination of SBL and EBL methods, learns a two-clause control-rule for this concept:

```
useful_insert_1(insert(A,[],[A]))← true
useful_insert_1(insert(A,[B|C],[A,B|C])← A =< B
```

These rules indicate that the first clause of `insert` should be used to insert into an empty list, or into a list having a first element that is larger than the item to be inserted. Incorporating these control clauses in the original program produces an im-

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

```
insert(A, B, [A|B]) :- useful_insert_1(A,B,[A|B]).
insert(A, [B|C], [B|D]) :- insert(A, C, D).

useful_insert_1(A, [], [A]).
useful_insert_1(A, [B|C], [A,B|C]) :- A =< B.
```

Figure 2.2: Improved Insert Predicate

proved version of `insert` shown in Figure 2.2. Using this version of `insert/2`, causes `permutation/2` to produce only ordered lists. Executing the improved program under the Prolog search strategy produces a version of the insertion sort algorithm, resulting in a dramatic improvement in program efficiency.

The control-rule learning framework of CHILL is very similar to that of DOLPHIN. CHILL learns rules to control the behavior of a shift-reduce parser expressed as a Prolog program. A major difference, however, is that CHILL learns control-rules to improve the accuracy of the parser, rather than efficiency *per se*. However, early pruning of inappropriate paths also leads to very efficient parsers. Another difference is that CHILL uses a purely inductive algorithm to learn control rules, although both systems make extensive use of ILP.

2.2 Inductive Logic Programming

2.2.1 The ILP Problem

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 an example 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`. Some positive instances might be: `member(1, [1,2])`, `member(2, [1,2])`, `member(1,[3,1])`, etc. While instances such as `member(1,[])`, member(2,[1,3]), would serve as negative examples. Additional information is provided in the form 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 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. 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).
```

ILP research has clustered around two basic induction methods, top-down and bottom-up. Each of these approaches has it's strengths and weaknesses; the induction algorithm employed in CHILL attempts to draw on the strengths of both.

2.2.2 Top-Down ILP Algorithms

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 summarized in Figure 2.3. A clause is said to cover 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. FOIL continues to learn clauses until all of the positive examples have been covered.

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

Figure 2.3: The FOIL Covering Algorithm

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 overed after extending the clause with a literal, L. Let T_+ denote the number of positive tuples in the set T and define:

$$I(T) = -\log_2(T_+/|T|). \tag{2.1}$$

The chosen literal is then the one that maximizes:

$$gain(L) = s \cdot (I(T) - I(T')), \tag{2.2}$$

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).

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])`); FOIL will continue 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.

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. The papers referenced above should be consulted for details on these and other features.

2.2.3 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 covering algorithm, except that new clauses are hypothesized by considering *least-general generalizations* (LGGs) of more specific clauses (Plotkin, 1970). The LGG of clauses C_1 and C_2 is the least general clause which 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) :-
  sib(john,ron), sib(john,dave),
  parent(ron,deb), parent(ron,ben),
  male(john), male(dave), female(deb).
```

```
uncle(bill,jay):-
  sib(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):-
  sib(A,C), sib(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):-
  sib(A,C) parent(C,B), male(A).
```

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

Golem produces candidate clauses by considering *Relative* LGGs (RLGGS) of positive examples with respect to the background knowledge. A positive example, `E`, is represented by the clause: `E :-` ⟨`every ground fact`⟩ where ⟨`every ground fact`⟩ is a conjunction of all true ground literals which 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. Of course, interesting background relations will give rise to an infinite number of such facts. Golem uses a finite subset corresponding to the facts which can be derived

```
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 the 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
```

Figure 2.4: The GOLEM Clause Construction Algorithm

through a fixed number of binary resolutions. The RLGG of two examples is simply the LGG of the examples' representative clauses.

The greedy clause construction algorithm of GOLEM is shown in Figure 2.4. 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 includes a number of filtering and clause–pruning heuristics to make the search process more efficient. The referenced paper describes these enhancements.

Chapter 3

Parser Induction with CHILL

The easiest way to understand the parser acquisition process in CHILL is to consider a concrete example. This chapter details the operation of CHILL by way of a simple case-role mapping problem that has been previously used to demonstrate certain language processing abilities of artificial neural networks (McClelland & Kawamoto, 1986; Miikkulainen & Dyer, 1991; Miikkulainen, 1993).

3.1 Overview

As described in the introduction, CHILL treats parser acquisition as a control-rule learning problem within a logic program that implements a suitable parser. The parsers learned by CHILL are based on basic a mechanism known as *deterministic shift-reduce parsing.*

3.1.1 Shift-Reduce Parsing

The idea of shift-reduce parsing originates from the well-known equivalence between the languages described by context-free grammars and languages recognized by push-down automata. A shift-reduce parser uses two data structures: an input buffer to store words of a sentence that have not yet been examined, and a stack which stores information concerning sentence constituents that have been recognized so far. Initially, the stack is empty, and the input buffer contains all of the words of a sentence to be processed. Parsing proceeds by applying a sequence of `shift` and `reduce` operations. A shift operation removes some items from the input buffer and pushes a corresponding item on the top of the stack, while a reduce operation pops one or more elements from the top of the stack and replaces them with a new element.

A "recognizer" for the language of a particular context-free grammar (CFG) is produced by translating each production of the grammar into a suitable parsing operator. For example, a production of the form $S \rightarrow NP\ VP$ is implemented as an operator which pops VP followed by NP from the stack and pushes S in their place. Productions which introduce terminal symbols (words) may be implemented either by unary reduction operators, or by shift operators which remove a word (or phrase) from the input buffer and push the corresponding category onto the stack. For example, $DET \rightarrow$ `the` becomes an operator to remove the word, "the" from the input buffer and push DET on the stack. A sentence is "recognized" by the parser if there is some sequence of operations which takes the parser from its initial state to a final state where the stack contains a single item representing the start-symbol for the CFG and the input buffer is empty.

The notion of a shift-reduce recognizer may be generalized to a shift-reduce parser (in the broad sense described in Section 1.2) by allowing the parsing operators to employ structured representations as stack items. These stack items represent pieces of the evolving structure, and `reduce` operations combine various pieces to form larger structures. The result is translator from input sentences into MRL expressions. In essence, a shift-reduce parser may be regarded as the operationalization of a phrase-structure grammar augmented with operations for building meaning representations.

Notice that the process of parsing a sentence is a search problem. The parser must find a sequence of operators that transforms the initial state into a final representation. A parser that never retracts (backtracks over) an operator once it has been applied is said to be *deterministic* (Marcus, 1980). Of course, natural languages are rife with ambiguities: lexical-class ambiguity, attachment ambiguity, semantic ambiguity, pronominal reference, etc. Given these ambiguities, choosing the correct operator to apply at any given point during parsing requires a great deal of knowledge, from information about syntax, to domain-specific world-knowledge. It is this knowledge that must be encoded into the search-control rules learned by CHILL.

Although CHILL produces deterministic parsers, this does not mean that the resulting parsers are incapable of producing multiple analyses for a single sentence. It may well be that a sentence in its given context is truly ambiguous and admits multiple analyses. In this case, additional analyses may be found by backtracking. The crucial point is that each applied operator leads to a correct analysis of the sentence.

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. Finally, the mechanism provides a principled control-structure that is representation neutral. Different styles of analysis may 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.

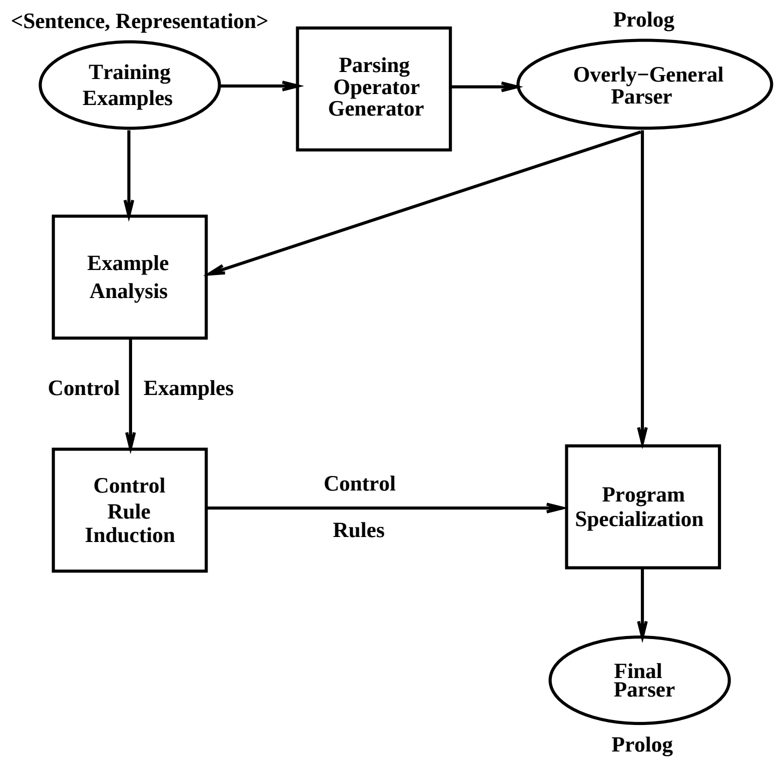

Figure 3.1: The CHILL Architecture

3.1.2 The CHILL Architecture

Figure 3.1 shows the basic components of CHILL. The input is a set of training instances consisting of sentences paired with the desired parses. The output is a deterministic shift-reduce parser in Prolog which maps sentences into parses. 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 training examples are parsed using the overly-general parser 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 architecture shown here makes no commitment regarding the *type* of analysis which the learned parser is expected to produce. One of the strengths of CHILL is the ease with which it may be adapted for use with differing analyses. In theory, CHILL can learn parsers for any representation framework that meets certain technical criteria, namely *operator transparency*, *derivation transparency*, and *training tractability*. Operator transparency means that a set of operators sufficient to parse the training examples is inferable from the structure of the examples themselves. Derivation transparency requires the ability to determine the correct sequence of operator applications required to parse any given training example. Finally, tractability requires that the training examples are parsable in a "reasonable" time-frame. The following sections explain these criteria and the details of CHILL in the context of case-role mapping.

3.2 An Example Framework: Case-Role Mapping

3.2.1 The Mapping Problem

Among the most common meaning-representation languages for natural language systems are various incarnations of case-role analysis. Traditional 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 from 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`). Domain-specific semantic knowledge is required to make the correct assignment. Obviously, the case-role assignment task is a difficult one, requiring many sources of knowledge, both syntactic and semantic.

3.2.2 Shift-Reduce Case-Role Parsing

Chill adopts a simple shift-reduce framework for case-role mapping similar to that used by Simmons and Yu (1992). The parsing process is best illustrated by way of example. Consider the sentence: "The man ate the pasta." Parsing 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 via some case-role to form a new element which is pushed back onto the stack. The sequence of actions and stack states for the example sentence is shown in Figure 3.2. The action notation *(x label)*, indicates that the top two stack items are combined via the role, *label*, with the item from stack position, x, being the head.

It is easy to see how this constrained framework meets the criterion of operator transparency. Any given training example requires a structure-building reduce action for each unique case-role that appears in the analysis. The set of actions required to

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

produce a set of analyses is just the union of the actions required for each individual analysis.

The second criterion, derivation transparency, is not quite as obvious. The notion of *the* "correct" operator sequence is somewhat slippery since there may be many potential sequences of operator applications which produce the correct parse as a final result. For example, the sentence above could have been parsed by first shifting every word in the sentence onto the stack and then performing the necessary reductions. This is analogous to distinctions between leftmost and rightmost derivations in parsing context-free grammars. It is not critical which derivation sequence is chosen, but it is imperative that the derivations be consistent so that similar examples are all parsed in a similar way. This is most easily guaranteed *operationally.* When searching for a successful parse, the operators are ordered to favor a certain type of derivation; the first derivation found is then considered to be the correct sequence for that example. More concretely, if reduce operators are always preferred over a shift, parsing will follow an opportunistic derivation which minimizes stack growth.

The third criterion, training tractability, is assured during Example Analysis, which will be discussed in Section 3.3.2.

3.3 The Phases of CHILL

3.3.1 Parsing Operator Generation

A shift-reduce parser for this representation is easily encoded 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 3.3 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`. 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; it is used to allow for "bare" heads as case fillers. Thus, it handles cases such as `reduce(man,det,the,NewTop)` and `reduce([ate,agt:[man,det:the]],patient,[pasta,det:the],NewTop))`.

Extending the program to parse further examples is accomplished by adding additional clauses to the `op/4` predicate. However, knowing that a role, say "agt" is used in the analysis does not indicate which of the two possible operators, *(1 agt)* and *(2 agt)* is actually needed. During operator generation, both are added to the overly-general parser; thus insuring a sufficient set of operators for parsing the training examples. Any unnecessary operator clauses will be removed from the program during the subsequent specialization process.

Notice that each specific parsing operator has been encoded as a single clause in the overly-general parser. Learning control information to render this parser deterministic requires learning clause-selection heuristics for the `op/4` predicate. Thus, we are faced with exactly the control-rule learning problem which was outlined in Section 2.1.1.

```
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.

reduce([Head|Slots],Role,Filler,[Head,Role:Filler|Slots]) :- !.
reduce(BareHead,Role,Filler,Result) :-
             reduce([BareHead],Role,Filler,Result).
```

Figure 3.3: Overly-General Parser for “The man ate the pasta.”

3.3.2 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. Intuitively, we need to somehow classify the context in which each operator is actually useful. Clearly, the information necessary for choosing an operator must reside in the state of the parser; at any given point, an examination of the contents of the stack and the remaining sentence should determine the appropriate course of action.

CHILL will specialize the parser by including control heuristics that guide the application of operator clauses. 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 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 "snapshot" of the 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. Examples of correct operator applications 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 NLP 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 must 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 being 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)
```

Note that there are other parse states such as `op([],[the,man,ate,the,pasta], A, B)` which do not appear in this list. This is because the (1 agt) clause of `op/4` requires that its first argument be a list containing at least two items. Since the clause cannot match these other subgoals, they will not be included as negative examples.

If only a single parsing of each sentence was desired, the set of negative control examples might be smaller. For example, the operator `op([man,the],[ate, the,` `pasta], A, B)` is a positive control example for the clause implementing the *(1 det)* action. If this clause occurred *before* the *(1 agt)* clause, then this control example would not appear as a negative example for the latter. If the *(1 agt)* clause were the last operator, it would have no negative control examples at all, and it would serve as a "default" operator when none of the prior clauses applied.

Example analysis is actually slightly more complicated than so far described. The overly-general parser as shown above, even when presented with training examples (which have instantiated outputs) will take an exponential amount of time to find the correct parse. Executing this specification as a standard Prolog program results in a search through all operator sequences of length 0, 1, 2, etc., until a sequence is found which produces the given parse. In order to insure training tractability, a modified version of the overly-general parser is used. This modified version employs intermediate checks on the state of the parse to immediately reject those that are obviously inconsistent with the final parse. The required sophistication of the intermediate checks depends on how long one is willing to wait for a parse of the training examples. For straight-forward case analyses, CHILL simply checks that any reduction operation produces a constituent which appears in the final result. While this check does not guarantee that incorrect paths will not be tried, it is sufficient to render the time required for example analysis insignificant compared to that required for subsequent phases of the acquisition process, notably control-rule induction.

Devising a consistency checking scheme to insure training tractability may sometimes require some ingenuity. Indeed, designing an appropriate checking scheme was one of the most difficult tasks in producing the database query analyses which are the subject of Chapter 7. However, it does not seem to be a severe constraint on the applicability of CHILL. The tractability of verifying a parse as opposed to *finding* one is somewhat analogous to the ease with which a proof may be verified, even though constructing one from scratch is often intractable.

3.3.3 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.

A major contribution of this research is CHILL's novel ILP algorithm which combines elements of both top-down and bottom-up methods introduced in Section 2.2. 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 chapter, 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. It 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 and the next word in the sentence is 'the'."

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

3.3.4 Program Specialization

The final step is to "fold" the control information back into the overly-general parser. Each operator clause in the overly-general parser is modified by adding the learned control knowledge so that attempts to use the operator inappropriately fail immediately. In this way, the search space of the overly-general parser is pruned to efficiently produce correct parses, effectively utilizing the learned control information without incurring the overhead of a separate interpreter.

For non-disjunctive (single clause) control rules, the learned conditions are simply placed into the program clause preceding the original conditions (if any), and the clause head is unified with the argument of the control rule. For disjunctive control rules, a single new literal is added at the front of the program clause. This new literal has the same arguments as the clause head. The definition of the new literal comprises the clauses of the learned control rule with the head functors modified to match the name of the new literal. 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]],[the|C],[D],[the|C]) :-
        animate(B), reduce(A,agt,[B,det:the],D).
animate(man). animate(boy). animate(girl) ....
```

The final parser simply consists of the overly-general parser with each operator clause suitably constrained.

3.4 Implementation

Chill was originally implemented in Quintus Prolog, and has since been rewritten in SICStus Prolog, version 2.1.9. The core control-rule learning components including the induction algorithm comprise about 4000 lines of code. The ability to parse with differing representations is achieved by loading different modules implementing the overly-general parser, and parsing operator generation.

As discussed in the example-analysis section, the version of the overly-general parser used in training is more elaborate than that used in the program specialization phase. Using Chill for a given type of analysis involves writing a training module that contains code for operator generation and parsing training examples, and a testing module, which is a simple overly-general parser shell. Chill calls on predicates in the training module to perform operator generation and perform parsing during example analysis. Chill then creates a set of optimized operators which are used in conjunction with the overly-general parser in the testing module to produce the final parser.

So far, training and testing modules have been developed for three basic types of analyses: case-role mapping, syntactic analysis, and logical database queries. In each case, the testing module is trivial, comprising fewer than 30 lines of Prolog code. The size of training modules is very dependent on the difficulty of parsing training examples tractably. The training code for syntactic and case-role analysis is not much more extensive than that required for their respective testing modules, while the training component of the database query system contains about 400 lines of code.

Chapter 4

The CHILL Induction Algorithm

As noted in Section 3.3.3, the learning of parser-control rules is a demanding induction task. At the time CHILL was being developed, no existing ILP system combined all of the necessary features. The control-rule induction component of CHILL employs a new induction algorithm called CHILLIN[1] (Zelle & Mooney, 1994a) for learning relational concept definitions.

While the GOLEM and FOIL systems presented in Chapter 2 have certainly been successful, each has its weaknesses. As discussed in section 2.2.3, GOLEM is based on the construction of relative least-general generalizations, *RLGG*s (Plotkin, 1970) which forces the background knowledge to be expressed *extensionally* as a set of ground facts. This explicit model of background knowledge can be excessively large, and the clauses constructed from such models can grow explosively. A partial answer to the efficiency problem is the restriction of hypotheses to the so-called ij-determinate[2] clauses, which reduces the class of of logic programs which can be learned. A related problem is sensitivity to the distribution of the input examples. If only a random sampling of positive examples is presented, the resulting model of

[1]for CHILL INduction algorithm

[2]A literal is determinate if its output variables have at most one binding, given the bindings of its inputs. A clause is ij-determinate if its body contains no literals with arity greater than j, all its literals are determinate, and every variable appearing in the clause is "connected" to the head by a chain of i or fewer literals.

the predicate to be learned is incomplete, and GOLEM may fail to create sufficiently general hypotheses, resulting in diminished performance.

FOIL also uses extensional background knowledge, but this is done for efficiency reasons; top-down algorithms can easily use intentionally defined background predicates to evaluate various competing hypotheses (as in CHILLIN and others (Lavrač & Džeroski, 1994; Cohen, 1992; Pazzani, Brunk, & Silverstein, 1991)). A more fundamental weakness is that FOIL constructs clauses which are function-free. Any functions (e.g. list structures) must be handled by including explicit constructor predicates as part of the background knowledge. The proliferation of constructor predicates can significantly degrade FOIL's performance. In addition, FOIL suffers it's own version of the incomplete model problem when trying to learn recursive predicates. Recursive hypotheses are evaluated by using positive examples as a model of the predicate being learned. When the examples are incomplete, they provide a "noisy oracle" and FOIL has difficulty learning even simple recursive concepts (Cohen, 1993).

Although specifically designed to deal with issues arising in the parser acquisition problem, CHILLIN is itself a novel ILP system combining elements of both top-down and bottom-up ILP methods. The use of bottom-up techniques allows CHILLIN to deal gracefully with highly structured examples without requiring "constructor" predicates, while top-down specialization allows for efficient induction in the presence of intensionally defined background relations. In addition, CHILLIN includes mechanisms for demand-driven predicate invention, more efficient learning of recursive predicates from random examples, and induction without the use of explicit negative examples. This chapter presents the details of CHILLIN and compares it with the seminal ILP systems FOIL and GOLEM.

4.1 The Algorithm

4.1.1 Top Level

The input to CHILLIN is a set of positive and negative examples of a concept (in the case of CHILL, parser–control examples) expressed as facts, and a set of background predicates expressed as definite clauses. The output of the induction is a definite-clause concept definition which entails the positive examples, but not the negative.

CHILLIN is at its heart a compaction algorithm that tries to construct a small, simple program that covers the positive examples. The algorithm starts with a most specific definition (the set of positive examples) and introduces generalizations which 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.

The metric for calculating program size is similar to the one used in CIGOL (Muggleton & Buntine, 1988). The size of a program is the sum of its clauses. Given a clause, C of the form `H :- B`, the size of C is computed as follows:

$$size(C) = 1 + termsize(H) + termsize(B) \tag{4.1}$$

$$termsize(T) = \begin{cases} 1 & \text{if } T \text{ is a variable} \\ 2 & \text{if } T \text{ is a constant} \\ 2 + \sum_{i=1}^{arity(T)} termsize(arg_i(T)) & \text{otherwise} \end{cases} \tag{4.2}$$

Generalizations in CHILLIN 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

```
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 4.1: CHILLIN Induction Algorithm

$\forall e \in E : [(C - C_i) \cup G \vdash e]$. 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 4.1 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 from these pairs is used to reduce `DEF`. The reduction of the definition is implemented as an efficient approximation of empirical subsumption. `G` is added at the top of the definition and then the standard Prolog proof strategy is used 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. This process guarantees that any clause which is empirically subsumed by G *alone* is removed from the definition; however, it does not guarantee that the resulting definition is minimal. Some clauses may be kept because they are the first to cover an example, even though the example could also be covered with a subsequent clause.

4.1.2 Constructing Generalizations

The `build_gen` algorithm is shown in Figure 4.2. There are three basic processes involved. First is the construction of a simple 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 and illustrated with a simple example from parser acquisition in the following subsections.

Constructing an Initial Generalization

The initial generalization of the input clauses is computed by finding the simple LGG of the clauses. For example, when learning control rules for the agent reduction in a case-role parsing system, 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 valid generalization (it covers no negative examples) and no further processing is required. Of course, such generalizations are not always correct. 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

```
**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 | e $\in$ Pos $\wedge$ REDUCED $\not\vdash$ E }
LITERAL := invent_predicate(CPOS, CNEGS, GEN)
GEN := GEN $\cup$ LITERAL
return GEN
```

Figure 4.2: Build_gen Algorithm

are constructed may contain non-trivial conditions. However, the construction of clause LGGs is still straight-forward. Unlike the RLGGs used by Golem, the simple LGGs in Chill are independent of any background knowledge and efficiently computable from the input clauses. At this point, GEN 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.

Adding Antecedents

As its name implies, `add_antecedents` attempts to specialize `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 which adds literals derivable either from 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 slight 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, S, of the number of clauses in `DEF` which are subsumed by `GEN`. This estimate is obtained by a method analogous to the approximation of empirical subsumption explained in Section 4.1.1. Each positive example is associated 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`. Let S be the estimate of clauses subsumed by `GEN` and T_{-} the count of negative examples covered by `GEN`. Let S' and T'_{-} represent the respective values for `GEN` extended by a literal, L. The gain of L is then:

$$gain(L) = S' \cdot (log_2(S'/(S' + T'_{-})) - log_2(S/(S + T_{-})))$$

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(Y). Hence, animate(Y) 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.

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. Chillin uses an approach similar to that of Champ (Kijsirikul, Numao, & Shimura, 1992). The first step in predicate invention is to identify a subset of clause variables, the instantiations of which are sufficient to discriminate between the positive and negative examples covered by the clause. The tuples produced by the *projection* of selected variables provide positive and negative examples of the new concept. The top-level induction algorithm is then called recursively with these examples to create a definition of the new predicate.

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. Thus, the projection consisting of the single variable, `Y` is sufficient to render the positive and negative tuples disjoint. In general, separating the positive and negative examples may require simultaneously constraining multiple variables. In the table above, The tuples represented by the pair `X` and `Z` taken together do separate the examples even though they do not do so individually.

Champ chooses a projection via a greedy process of elimination. Initially, the projection contains all of the variables. Each variable is tested in turn to see whether its removal from the projection would cause overlap between the positive and negative tuples; variables that are not needed to insure disjointness are dropped from the projection. Champ's algorithm guarantees that the resulting projection is minimal in the sense that no other variable can be dropped and still maintain separation of the examples; however, it does not guarantee a projection containing the minimum number of variables required. Considering variables in a different order may produce a different minimal projection having fewer variables. The greedy selection of some minimal projection is used as an efficient approximation to computing the minimum projection.

The final criterion for acceptance of an invented predicate in CHILLIN is whether the generalization that uses the new predicate produces compaction of the current top-level definition. Obviously, a new predicate is more likely to be useful if it has a small definition itself, since the size of the predicate's definition counts against the compaction of the top-level predicate produced by the generalization. Ideally then, CHILLIN should select a projection of clause variables that separates the positive and negative examples while minimizing the size of the new predicate's definition. Of course, selecting the projection having the smallest definition is even less tractable then selecting a minimum projection. CHILLIN employs a greedy growth algorithm in an effort to find a small projection which guarantees separation and at the same time minimizes the number of unique positive tuples in the projection. This process is based on the assumption that a concept with fewer positive examples will have a simpler definite-clause definition. Although this is not true in general, it seems an intuitive heuristic. In many cases CHILLIN is unable to achieve any compaction of the positive examples for the new predicate and ends up "memorizing" the positive examples, obviously the fewer, the better.

The search for a projection in CHILL 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 number of unique positive tuples. At each step, that variable is added which maximizes the ratio of negative examples eliminated to positive tuples added. The search terminates when the overlap between positive and negative tuples is empty. This process does not guarantee a minimal projection in the CHAMP sense, but does prefer small projections; in practice, the resulting projections are almost always minimal. If a minimality guarantee was desired, the greedy growth algorithm of Chillin could be used to produce an initial projection that is subsequently "minimalized" by the Champ elimination algorithm.

Once a projection has been chosen, 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`, since it alone is sufficient to discriminate the covered positives from the covered negatives. The derived positive examples are `p1(man)`, `p1(boy)` and `p1(girl)`, and the negative examples are `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, something which is not possible in the purely top-down framework of Champ.

Handling Recursion

When introducing clauses with recursive antecedents, care must be taken to avoid unfounded recursion. Foil handles this issue by attempting to establish an ordering on the arguments which may appear in a literal. Chill takes a much simpler approach based on structure reduction: each recursive literal must have an argument that is a proper sub-term of the corresponding argument in the head of the clause. For example, in the clause `member(A, [B|C]) :- member(A,C)`, the second argument of the recursive literal is structure reducing, and any recursive chaining of this clause must eventually "bottom-out." Well-founded recursion among multiple clauses is guaranteed by ensuring that every recursive literal has at least one argument that is structure reducing, and for all other recursive literals in the definition, the same argument is a (possibly improper) sub-term of the corresponding argument in the head of the containing clause. This property is maintained by dropping any unsound recursive literals produced by the LGG operation and only considering addition of recursive antecedents which meet the structure-reducing conditions. These restrictions on recursive definitions are more restrictive than those imposed by FOIL, but this simple approach works well on a large class of problems.

The evaluation of recursive clauses also requires some consideration. Testing the coverage of a non-recursive clause is easily achieved by unifying the head of a clause with an example and then attempting to prove the body of the clause using the background theory. Evaluation of a recursive clause, however, requires a definition of the concept being learned. FOIL and GOLEM both rely on the extensional definition provided by the positive examples, an approach which results in the "noisy oracle" problem discussed in the introduction to this chapter. CHILL, on the other hand, is able to use the current definition of the predicate being learned, which is guaranteed to be at least as general as the extensional definition. Coverage of recursive clauses is tested by temporarily adding the clause to the existing definition and evaluating the antecedents in the context of the background knowledge and the current (extended) definition. In this way, generalization of the original examples (say the discovery of the recursive base-case) can significantly improve the coverage achieved by correct recursive clauses. This approach gives CHILLIN a significant advantage in learning recursive concepts from random examples.

This approach to recursion has proven effective in practice, although it is not without shortcomings. If a recursive clause is introduced and subsequent generalizations expand the coverage of the recursive call, the resulting definition could cover negative training examples; the current implementation does not check to insure that new generalizations maintain global consistency (although this would be easy to do). Such undesirable ordering effects have not arisen in practice because recursive clauses do not generally show high gain until adequate base-cases have been constructed.

4.1.3 Implementation

Efficiency Considerations

The actual implementation of CHILL is somewhat more complicated than the abstract description presented so far. As the above discussion indicates, the process of constructing a generalization involves three steps: form an LGG, add antecedents and invent a new predicate. If this much effort was expended for a reasonable sampling of clause pairs on every iteration of the compaction loop, the algorithm would be intolerably slow. The current implementation provides two remedies for this problem.

First, the outer compaction loop is initially performed using only the LGG construction process to find generalizations. When no more compaction is found using simple LGGs, the more sophisticated refinement mechanisms are tried. Significant compaction is often obtainable in the initial phase, reducing the size of the definition on which the subsequent (more intensive) processing is done. This initial pass can often reduce thousands of control examples to a definition containing only tens of unit clauses.

A second conservation of effort is achieved by interleaving the building of generalizations. A given iteration of the compaction loop begins by gathering a sampling of clause pairs from which LGGs are immediately constructed. These generalizations form a pool of clauses which may need further refinement. CHILLIN proceeds by repeatedly removing the most promising clause and extending it with a single antecedent. The resulting clause is then returned to the pool and the process continues. If the selected clause is unextendable, it is set aside as a candidate for predicate invention. Predicate invention is invoked only if the pool of clauses has been exhausted without finding a consistent generalization.

Parameters

As with all complex learning algorithms, the performance of CHILLIN may be "tuned" by setting a number of parameters. There are five major parameters which affect the performance of CHILLIN. *Sample_size* determines the number of pairs of clauses which are considered for constructing generalizations on each iteration of the main compaction loop. This value defaults to 15. *Effort* determines the level of effort which is used in searching for generalizations. CHILLIN may be restricted to just trying LGGs, using LGGs plus top-down specialization without invention, or using LGGs, specialization and invention. The last case is the default. In a similar spirit, a separate parameter, *recursion* is provided to disable or enable the learning of clauses containing recursive literals. Since the evaluation of recursive clauses requires extra work, turning off recursion may significantly speed up the induction process. *Failures_to_exit* is a parameter which determines how many compaction iterations which result in no further compression are allowed before giving up. By default this value is 3. Finally, there is a limit on the number of arguments that an invented predicate may have; the default is 2, meaning that generalizations which require the invention of predicates having more than 2 arguments are not followed up.

4.2 Experimental Comparison with Other ILP Systems

Obviously CHILLIN has been primarily used within the larger parser-acquisition framework of CHILL. However, a series of experiments was performed to compare the performance of this ILP algorithm to GOLEM and FOIL on some benchmark ILP tasks. These systems were chosen for comparison because they are well-known, and arguably the most mature and efficient ILP platforms developed to date.

4.2.1 Experimental Design

There is, as yet, no standard approach to the evaluation of ILP systems. This evaluation focuses on the ability of systems to perform relational concept-learning tasks. The question of interest is how well a hypothesis learned from some (random) sampling of examples characterizes the entire example space. Therefore, we have adopted an experimental strategy, common in propositional learning, of randomly splitting the example space into disjoint training and testing sets. The systems were trained on progressively larger portions of the training examples and the performance of the learned rules assessed on the independent testing set. This process of splitting, training and testing was repeated and the results averaged over 10 trials to produce learning curves for each of the systems on several benchmark problems. It is important to note that ILP systems are often tested using a set of complete[3] or carefully chosen positive examples. We would not necessarily expect the systems to perform as well under the more difficult conditions of random selection used here.

The number of training examples in successive training sets was chosen experimentally to highlight the interesting parts of the learning curves. Except where indicated, enough training examples were provided so that the system having the best accuracy achieved a perfect score on the majority of the runs. The distribution of positive examples in many relational domains is quite sparse, and a relatively large number of positive examples are required for each of these ILP systems. In order to insure a reasonable number of positive training examples, training sets were always selected to be one-fifth positive and four-fifths negative examples. Testing sets included an equal number of positive and negative examples to test the ability of the resulting rules to recognize instances of the concept and reject non-instances. This change in distribution from training to testing is somewhat non-standard, but it was done in an effort to gauge empirically (by test-set coverage) whether or not

[3]In this case, all possible examples up to a given size are generated from a set of constants and functors. Each example is then classified as either a positive or negative instance.

the systems had learned a completely correct definition.

The experiments were performed using version 5.0 of FOIL and version 1.0 α of GOLEM both of which are written in C. All of the algorithms were run with default settings of the various parameters. No extra mode, type, or bias information was provided besides the examples and background predicates. While all of the algorithms can make use of additional constraints, they do not necessarily do so in consistent ways; therefore, providing no extra information to any algorithm allows for a more direct comparison.

4.2.2 Accuracy Results

Learning Recursive Programs

The first three learning problems tested the ability to learn simple recursive concepts. Three problems widely used in the ILP literature were chosen: the list predicates `member` and `append` and the arithmetic predicate `multiply`.

For the list predicates, the data consisted of all lists of length 0-3 defined over three constants. The background information consisted of definitions of list construction predicates, `null` which holds for an empty list and `components` which decomposes a list into its head and tail. The results for these two problems were approximately the same. The learning curves for `member` and `append` are presented in Figures 4.3 and 4.4. Overall, with random examples, CHILLIN was able to learn accurate definitions with fewer examples than the other systems, and without using the background predicates. FOIL did show a slight advantage on `append` for very small training sets.

The domain for the `multiply` problem consisted of integers in the range from zero to ten. The definition was to be learned in terms of background predicates: `plus`, `decrement`, `zero`, and `one`. We expected FOIL and GOLEM to do well on this problem as it is a standard benchmark which both systems have been shown capable of learning. CHILLIN, in its current form, is not able to formulate the correct

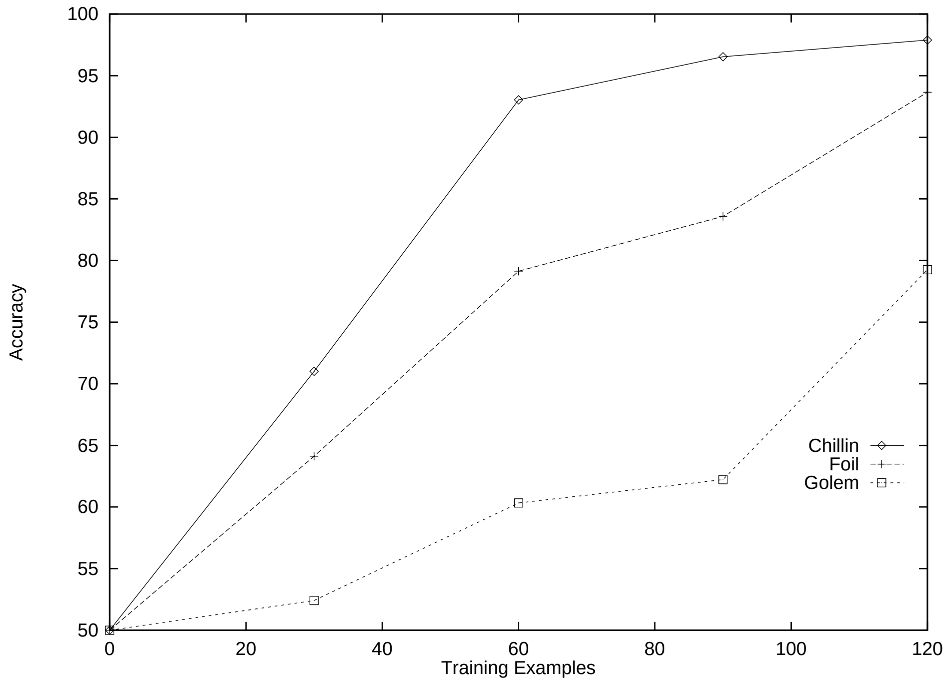

Figure 4.3: Accuracy on `member`

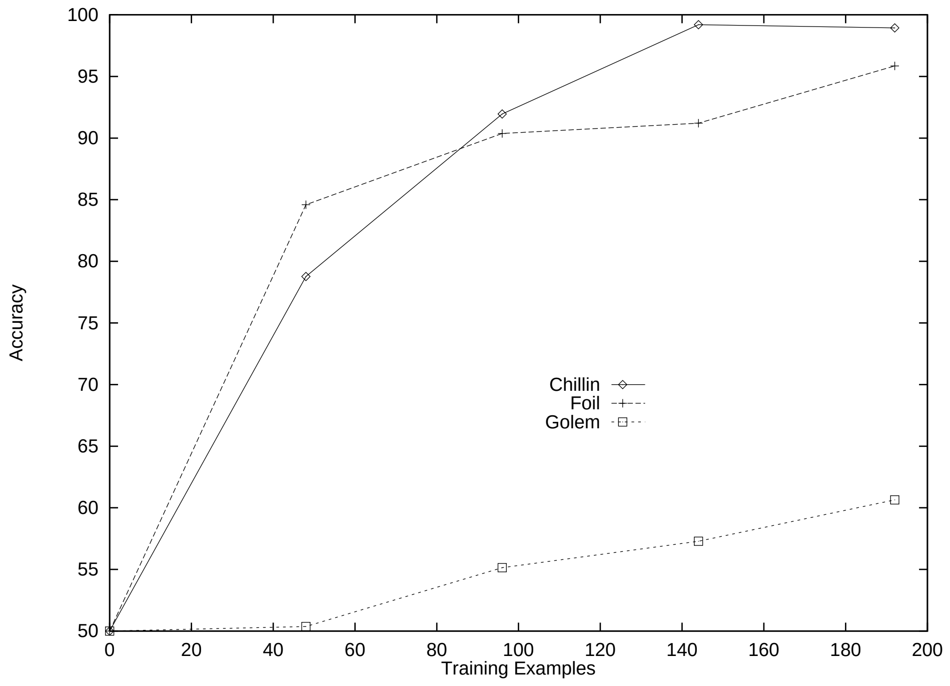

Figure 4.4: Accuracy on `append`

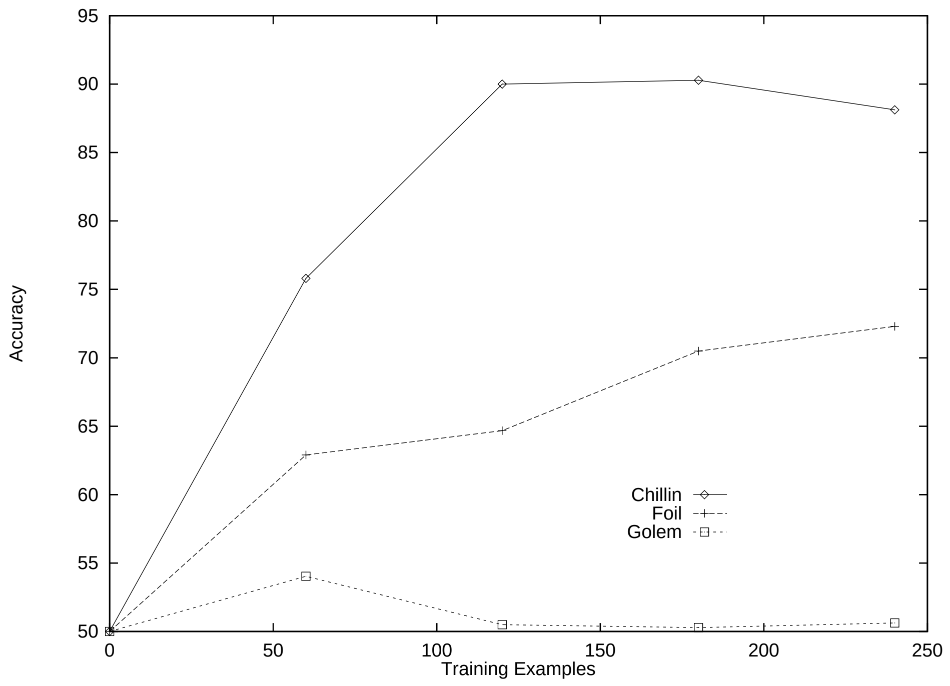

Figure 4.5: Accuracy on `multiply`

recursive definition for this predicate, since the required recursive clause does not meet the structure-reducing conditions.

The learning curves, shown in Figure 4.5, turned out to be quite surprising. None of the systems showed the ability to learn this concept accurately from random examples. Chillin quickly converged to definitions that were 90 percent correct for the limited domain, and was unable to improve. Its inaccurate definitions, however, were much better than those found by either of the other systems. Further experimentation showed that Foil kept improving as the training set grew, but it was only reliable in generating correct definitions with nearly complete training sets. Golem was unable to learn the correct definition without additional guidance such as mode declarations.

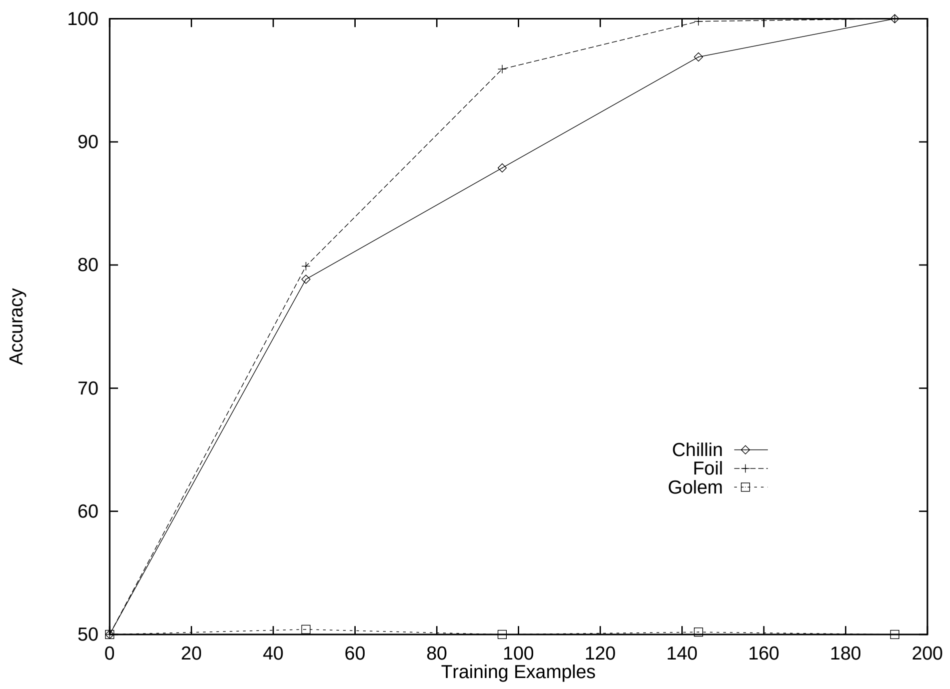

Figure 4.6: Accuracy on `uncle`

Learning with Nondeterminate Literals

Another traditional test-bed for relational learners is the domain of family relationships. We performed experiments with an extended family tree in which the target predicate was either `grandfather` or `uncle` and the background consisted of facts concerning the relations: `parent`, `sibling`, `married`, `male` and `female`. This domain is interesting because it requires the use of literals which violate determinacy conditions used by Golem and other bottom-up ILP systems.

As expected, Chill and Foil do quite well on these problems, and Golem is unable to learn any reasonable definitions. On the `uncle` problem, shown in Figure 4.6, both Foil and Chill learned accurate definitions from 100 training examples, with Foil having a slight edge on smaller training sets. Rather surprisingly, however, Foil seemed to have more trouble on the simpler `grandfather` definition. As can be seen in the learning curves in Figure 4.7, Foil's performance takes a mysterious dip

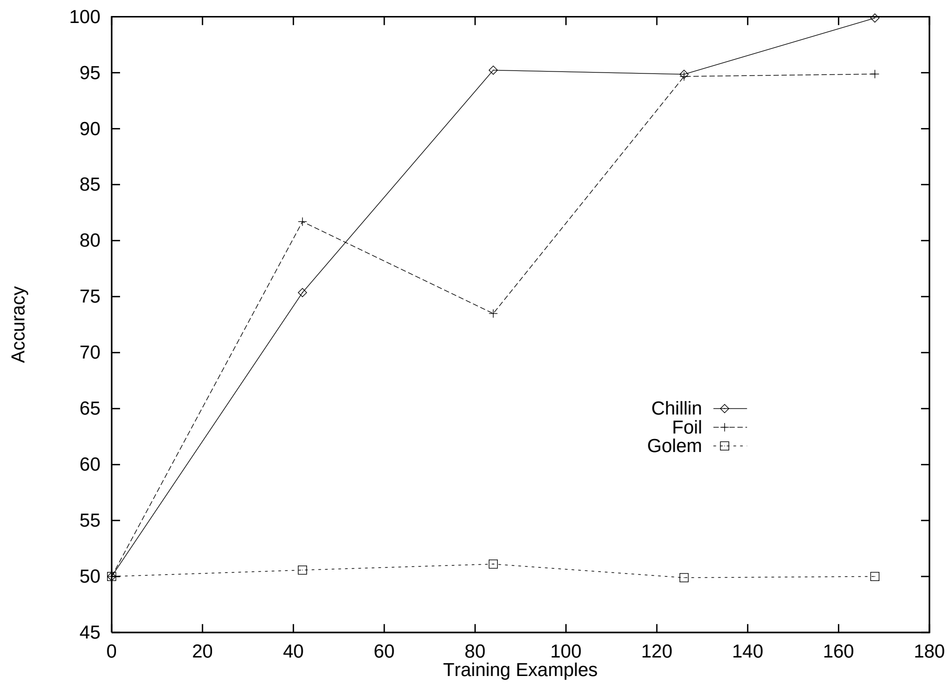

Figure 4.7: Accuracy on `grandfather`

at 75 training examples before catching up with CHILL at 125 examples. Even at 175 examples where CHILL succeeds in finding a correct definition in all 10 trials, FOIL is only learning the correct definition half of the time. These experiments indicate that CHILL, like FOIL is able to learn definitions containing nondeterminate literals.

Control-rule Learning

The previous experiments concerned learning well-defined concepts containing only one or two clauses. The CHILL induction algorithm was originally designed for learning control rules from structured examples where the definition of the correct concept is not necessarily simple, and certainly is not known *a priori*. The last experiment was attempted to compare the performance of these systems on this type of problem. The problem chosen was a relatively simple task of determining when a shift-reduce parser should perform a shift operation in parsing a simple, regular corpus of active

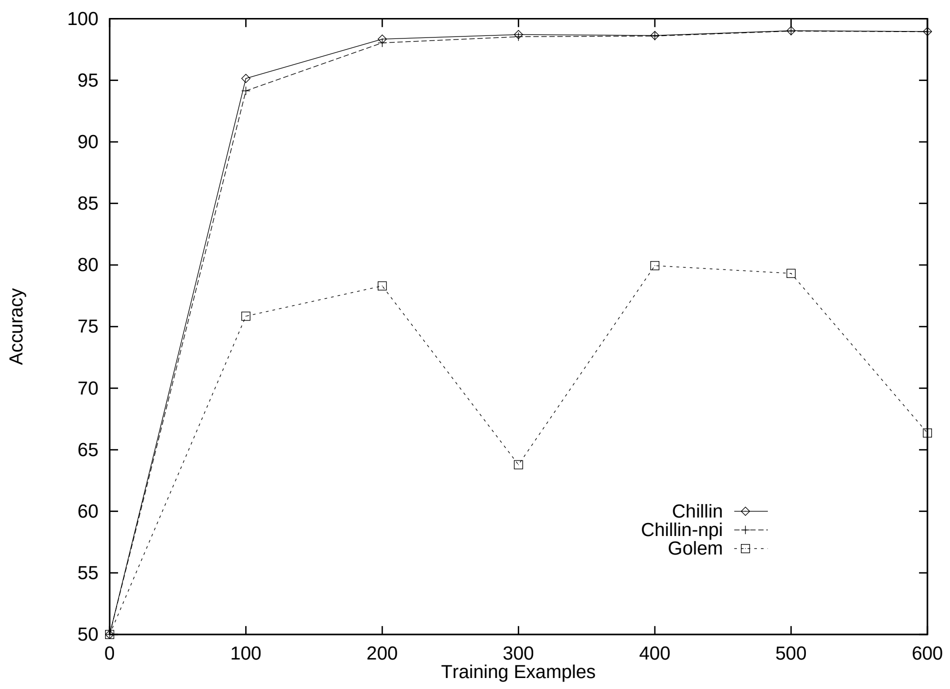

Figure 4.8: Accuracy for control rule (`shift`)

sentences.[4] CHILL typically learns a five or six clause definition for this concept.

The data for this problem was modified slightly so that the only logical functions appearing in the examples are list constructions. GOLEM and CHILLIN can both handle these structures without explicit constructor predicates. Unfortunately, it is not possible to run FOIL on this data. FOIL requires extensionally expressed constructor predicates; the `components` relation over lists of the required size (up to 8) constructible from the set of 34 constants appearing in these examples would require trillions of background facts. This illustrates the difficulties posed by the extensional background requirement.[5]

The graph of Figure 4.8 shows the learning curves. On this problem CHILLIN tends to invent new predicates. For direct comparison, we performed the experiments

[4]This data is derived from the case-role corpus of McClelland and Kawamoto (1986) which is described in detail in Chapter 5.

[5]One possible "solution" to this difficulty is to provide only those `component` facts that can be derived by decomposition of lists appearing in the training examples. This work-around was not tried for these experiments.

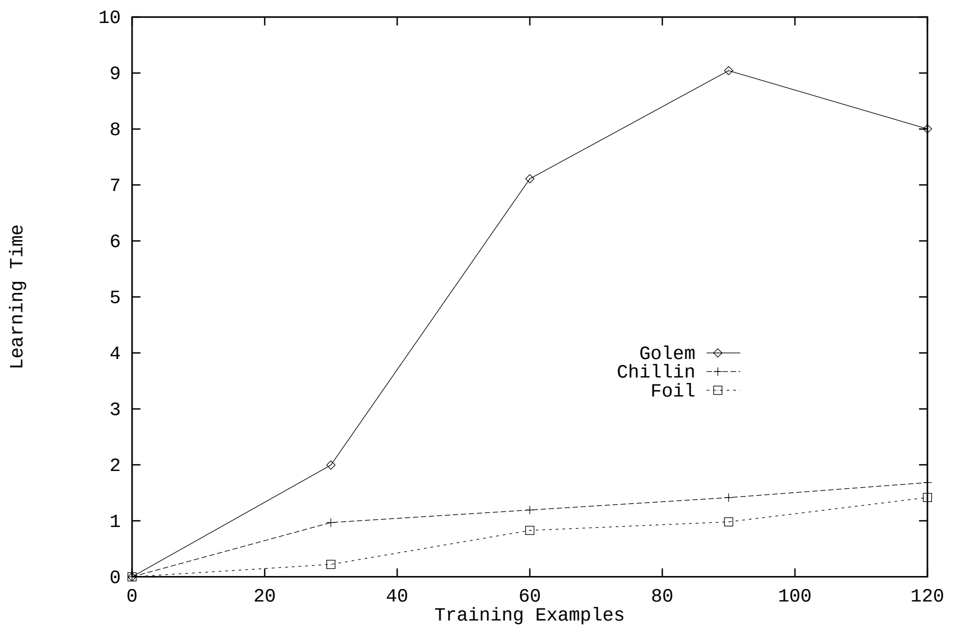

Figure 4.9: Time for `member`

with two versions of CHILLIN; the curve labeled "CHILLIN-npi" is CHILLIN with predicate invention turned off. The learning curves show that CHILLIN rapidly converges to very good definitions. Disabling predicate invention had only a minor impact (1%) in accuracy with smaller training sets, and no difference was detectable for larger sets. GOLEM, on the other hand, never achieves greater than 80% accuracy and displays erratic learning behavior in this domain.

4.2.3 Timing Results

Given the differences in implementation, we expected FOIL and GOLEM to be considerably faster than CHILLIN. However, this was not the case. On all problems where GOLEM was learning useful rules, it was significantly slower than CHILLIN often by a factor of 10 or more. While FOIL tended to be faster than CHILLIN, the learning times for the two systems were generally comparable. The timing curves for the `member` experiments shown in Figure 4.9 are typical. This graph shows the

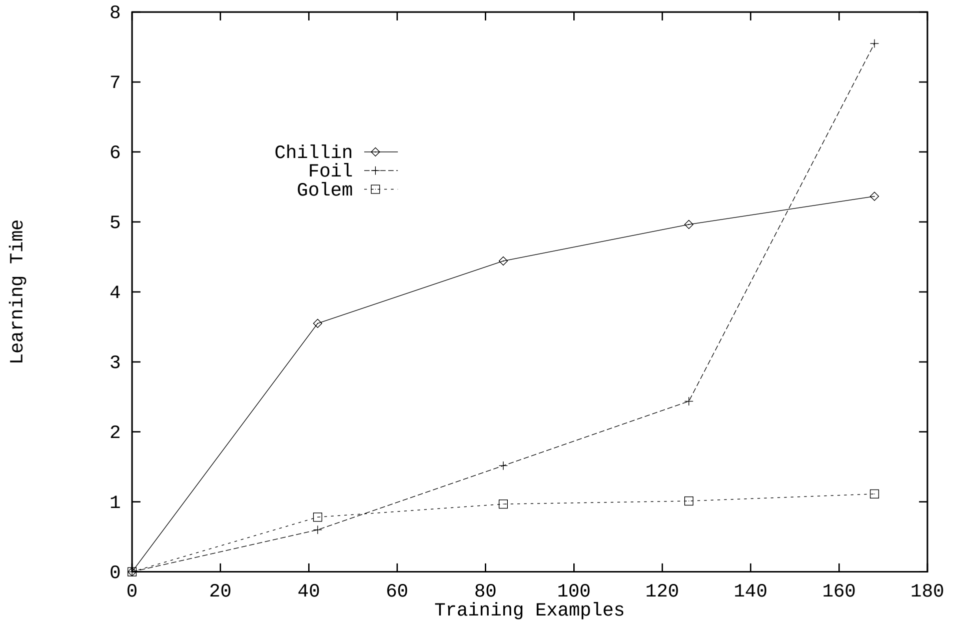

Figure 4.10: Time for `grandfather`

CPU-time in seconds required to learn a set of rules as a function of training set size. In the experiments where FOIL had more difficulty learning accurate rules such as `multiply` and `grandfather`, CHILLIN actually ran faster than FOIL at some data points. The graph in Figure 4.10 shows timing results for `grandfather`. Note that the run-time for GOLEM is lower here because it is not learning a definition, but rather, just memorizing the examples.

4.3 An Extension: Learning Without Negative Examples

4.3.1 Motivation

Most ILP systems require both positive and negative examples of ground instances of a predicate to be learned. However, explicit negative examples of a predicate are not always readily available. A standard solution is to automatically produce a large set of negative examples using a closed-world assumption, i.e. for an n-ary predicate, all n-tuples of terms chosen from a fixed set are generated and the positive examples are removed. However, it is frequently intractable to generate an adequate set of negative examples using this brute-force approach. The domain of parser acquisition presents such an example.

A naive alternative to CHILL's method of parser acquisition would be to simply present a corpus of sentences paired with representations as positive examples to an ILP system. For example, we might try to learn a definition of the concept `parse(Sentence,Rep)`. The induced logic program, might then be used to prove goals having the second argument uninstantiated, effectively producing parses of sentences provided as input. One problem with this approach is the lack of a convenient set of negative examples. Clearly, it is intractable to generate all possible sentences paired with incorrect analyses as a set of negative examples. Even selecting a manageable–sized random subset of these negative examples is unlikely to be sufficient, as such a sample is unlikely to include the many "near–miss" examples which are crucial to learning good generalizations.

A version of CHILLIN has been implemented which exploits the notion of output completeness (introduced in Section 3.3.2) to implicitly determine when a clause is overly-general and to quantify the degree of over–generality by simply estimating the number of negative examples covered. This extension of CHILLIN was used to provide comparisons between CHILL and the "naive" approach to parser acquisition. The results of these comparisons are presented in Section 6.4, the remainder of this section explains the techniques used for induction in the presence of implicit negative examples.

4.3.2 Counting Implicit Negative Examples

Learning without explicit negatives requires an alternate method of evaluating the utility of a clause. A mode declaration and an assumption of output completeness together determine a set of implicit negative examples.

Consider the predicate, `parse(Sentence,Representation)`. Providing the mode declaration `parse(+,-)` indicates that the predicate should provide the correct representations when provided with the sentence. Assuming that the training examples are output complete, determining if a clause is overly-general is straightforward. For each positive example, an *output query* is made to determine all outputs for the given input (e.g. `parse([the,man,ate], X)`). The clause is then used to prove the output query. If any outputs are generated that are not positive examples, the clause still covers negative examples and requires further specialization.

When such specialization is needed, the gain metric employed by the top-down component of CHILLIN must be able to compute the number of negative examples covered. Clearly, each ground, incorrect answer to an output query counts as a single negative example (e.g. `parse([the, man, ate], [ate, pat:[man, det:the]])`). However, output queries will frequently produce answers with universally quantified variables. For example, the clause, `parse([the,A,B],[C, agt:D]) :- true` generates the output, `[C, agt:D]` which may cover a great number of potential negative examples (e.g. `[hit, agt:[man, det:barbecue]])` . Similarly, the clause, `parse(X,Y) :- true` produces entirely uninstantiated output, intuitively, we would think that it covers even more negative examples than the previous result, even though it contains fewer uninstantiated variables. What is needed is some method of estimating the coverage of these (partially) uninstantiated results.

Quantification of coverage employs a parameter u representing the cardinality of the set of all examples of the concept being learned. Generally, relational concepts are quite sparse over the universe of constructible terms, so u may be considered to be some (very) large multiple of the count of positive examples. The count of examples covered by a non-ground answer to an output is estimated by uniformly distributing the probability of matching an example across all components of the example. For example, the term `parse(S,R)` where `S` and `R` are uninstantiated must cover all u possible examples. If we assume there are s possible sentences and r

possible representations, then $u = s \cdot r$. Distributing these possibilities equally across the two arguments suggests $s = r = \sqrt{u}$. A result such as `parse([the, man, ate], Rep)`, where `Rep` is uninstantiated, is estimated to cover $\sqrt{u}$ examples, since that is the number of different values possible for `Rep`.

The case of the result `parse([the, man, ate], [C, agt:D]` is slightly more complicated. The $\sqrt{u}$ possibilities for the second argument are further restricted by the structure appearing there. In this case, the argument starts with a list construction functor, which has arity 2. To account for that particular functor's appearance as well as the possible terms which could appear as its arguments, each argument is computed to account for $\sqrt[3]{\sqrt{u}}$ possibilities. In general, the coverage for each argument of a term with arity n is recursively calculated to be $\sqrt[n+1]{t}$ where t is the number of possibilities for the entire term. Following this strategy, we compute $\sqrt[6]{u}$ possibilities for `C` and $\sqrt[18]{u}$ terms possible for `D`. The entire term is therefore calculated to cover $\sqrt[18]{u^4}$ total examples. Finally, the number of *negative* examples covered is calculated as the total coverage of all examples minus the number of positive examples covered.

While these calculations employ assumptions which do not hold for any but the most artificial of domains, they do embody certain intuitive properties. Most importantly, the more instantiated a term is, the fewer examples it covers. This favors specialization of clauses so that output arguments do not contain free variables; this is essential to the induction of "constructive" programs (i.e. programs that produce fully instantiated outputs when given fully instantiated inputs). This approach also favors examples that are more instantiated at higher levels, since the estimate of coverage decreases exponentially with term depth. This supports successively specializing terms in a top-down fashion which is consistent with the overall approach of a FOIL-like learner.

4.3.3 Predicate Invention with Implicit Negative Examples

Replacing the count of covered explicit negatives with an estimate of negative coverage as described above is a simple revision of CHILLIN's top-down specialization component. However, the predicate invention mechanism also makes use of negative examples and requires modification.

There are two cases to consider when modifying predicate invention to use implicit negative examples. The simpler case occurs when all output variables in the head of the clause for which the predicate is being invented also appear in the body of the current clause. In this situation, output queries always produce ground results for the output arguments. The fact that the clause still covers negative examples means that there are some queries for which the ground outputs are simply incorrect. These incorrect outputs become a set of explicit negative examples for the normal predicate invention algorithm, and the recursive invocation of the top-level algorithm is made using explicit negative examples. The version of CHILLIN used for the experiments reported in later chapters used only this mechanism for predicate invention. Generalizations which still contained free variables were simply discarded.

When some of the output variables appearing in the clause head do not appear in the clause body, it is impossible to generate a set of ground negative examples. However, it is still possible to perform demand-driven predicate invention. Since the remaining free output variables must be bound to prevent coverage of implicit negatives, they must be included as output variables of the new predicate. The input variables can be chosen by finding a small subset of the bound variables that are sufficient to *functionally determine* the values of these output variables when covering the positive examples. Once chosen, the instantiations of these variables when the clause is used to cover positive examples produce a set of derived tuples which can be used as positive examples for the new predicate. The implicit negative version of the induction algorithm can then be invoked using these examples and associated mode declaration. Future versions of CHILLIN will implement this approach.

Chapter 5

Experiments with Case-Role Parsing

The explanation of CHILL in the previous two chapters has made use of simple examples from the domain of case-role parsing, a domain that served as one of the first testing grounds for the ideas in CHILL. This chapter discusses some of the experiments which have been conducted with CHILL for this type of representation, some of which were first reported in (Zelle & Mooney, 1993b).

5.1 Background

Semantic case analyses have proven very useful in the construction of natural language systems(Allen, 1995). However, the construction of parsers to produce such representations is complicated by the need to use domain-specific semantic information to resolve ambiguity and produce accurate analyses. While it is certainly possible to manually construct parsers for limited domains, new systems must be written for each semantic domain and the size of the rule-base required for more general applications can make manual construction infeasible. A natural response to this difficulty is to apply machine learning techniques to help automate acquisition of the required knowledge.

Recent research in learning the case-role mapping task has taken place under the connectionist paradigm (Miikkulainen & Dyer, 1991; St. John & McClelland, 1990; McClelland & Kawamoto, 1986). It is argued that proper case-role assignment is a difficult task requiring many independent sources of knowledge, both syntactic and semantic, and therefore well-suited to connectionist techniques.

The work of Miikkulainen and Dyer (1991), who used the case-role mapping task to demonstrate their FGREP method, is illustrative. Their model employs a recurrent network which allows words of an input sentence to be processed sequentially. Following (McClelland & Kawamoto, 1986), the network output has fixed slots for verb, agent, instrument, patient and modifier (a slot for the modifier of a patient such as "cheese" in "pasta with cheese"). The network is trained using a modification of backpropagation that automatically develops distributed word encodings during the training process. Words are presented to, and read out of, the network using these learned encodings. The model was demonstrated using a set of 1475 sentence/case-structure pairs originally from (McClelland & Kawamoto, 1986) (hereafter referred to as the M & K Corpus).

Connectionist models face a number of difficulties in handling natural language. Since the output structures are flat (non-recursive) it is unclear how the embedded propositions in more sophisticated analyses can be handled. The models are also limited to producing a single output structure for a given input. If an input sentence is truly ambiguous, the system produces a single output that appears as a weighted average of the possible analyses, rather than enumerating the consistent interpretations. The symbolic techniques of CHILL do not suffer from these deficiencies.

The crucial test of any learning system is how well it generalizes to handle previously unseen cases. This is particularly important in the domain of parser acquisition; given the generativity of natural languages, it is unreasonable to assume that that a system will be trained on more than a small fraction of possible inputs.

Empirical results demonstrate that CHILL trains faster and generalizes to novel inputs better than its neural network counterparts.

5.2 Parsing the M & K Corpus

5.2.1 Experimental Design

The M & K Corpus comprises 1475 sentence/case-structure pairs. The pairs were produced from a set of 19 sentence templates generating sentences/case-structure pairs for sentences such as "The HUMAN ate the FOOD with the UTENSIL", where the capitalized items are replaced with words of the given category. A complete listing of the templates and fillers may be found in Appendix A. The sample actually contains 1390 unique sentences, some of which admit two analyses. For example, the sentence "The man hit the boy with the bat" is ambiguous as to whether "bat" is an instrument of hitting or a possession. Since our parser is capable (through backtracking) of generating all legal parses for an input, training was done considering each unique sentence as a single example, and insuring that the training corpus was output complete.

Training and testing followed the standard paradigm of 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 reflect averages over five trials. During testing, the 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 inaccuracy, a metric was introduced to calculate the "average correctness" for a given test sentence as follows: $Accuracy = (\frac{C}{P} + \frac{C}{A})/2$ 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

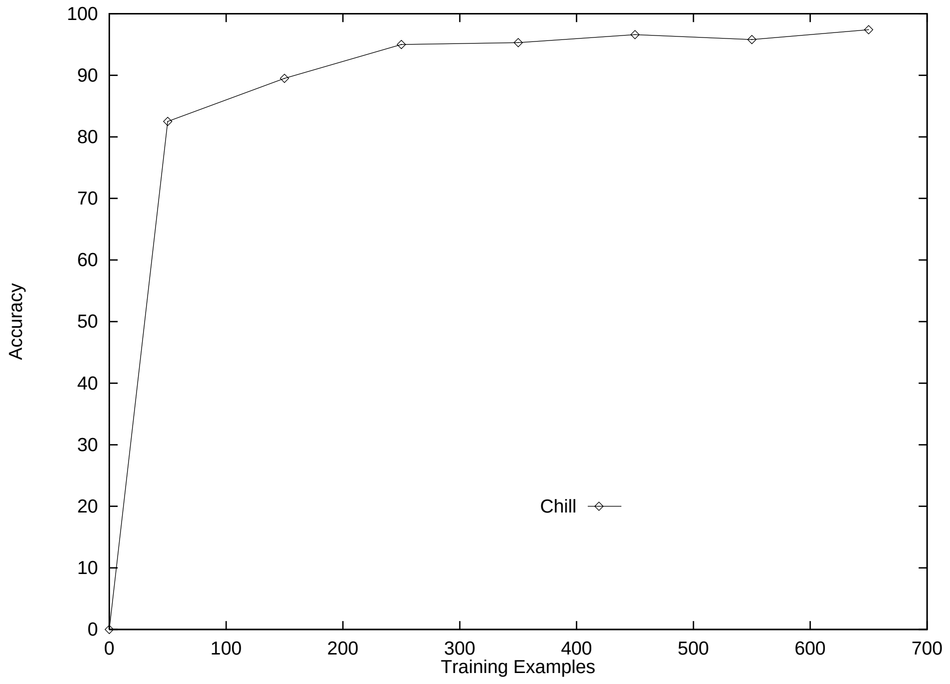

Figure 5.1: M&K Accuracy

sentence.

5.2.2 Results

Chill performs very well on this learning task as demonstrated by the learning curve shown in Figure 5.1. The system achieves 92% accuracy on novel sentences after seeing only 150 training sentences. Training on 650 sentences generated a set of parsing operators comprising about 60 lines of Prolog code that achieved 98% accuracy. The system also exhibits the desirable property that it tends to produce very few inaccurate parses. The graph in Figure 5.2 shows the probability that a produced parse is incorrect as a function of training set size.

This initial experiment, following the example in (Miikkulainen & Dyer, 1991), did not use distinct tokens for different senses of ambiguous words. However, one of the original motivations for connectionist approaches was the ability to handle lexical ambiguity (McClelland & Kawamoto, 1986; St. John & McClelland, 1990). The M

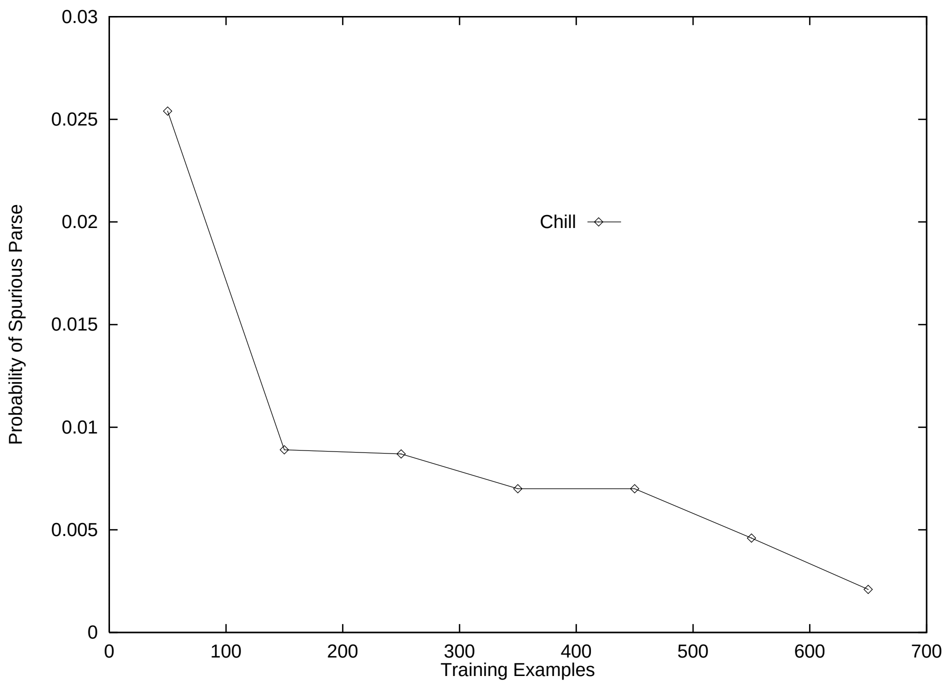

Figure 5.2: M&K Percent Spurious Parses

& K Corpus was explicitly designed with two ambiguous lexical items, “bat” (flying vs. baseball) and “chicken” (live-animal vs. dead-food). A second experiment was carried out using distinct tokens for different word senses in the case representations. The parsing model was extended to include independent shift operators for each sense of ambiguous words. Using this modification, the experiment was repeated. The results are shown in Figure 5.3. The curve is virtually identical to that of the previous experiment showing that Chill can successfully incorporate simple lexical disambiguation within the case-role mapping task.

One of the nice properties of the M & K Corpus is that it contains enough complexity to be interesting, yet remains simple enough that the resulting rules are amenable to inspection and analysis. Such analysis revealed that Chill consistently invented interpretable word classes. For example, the invention of `animate` occurred as illustrated in the algorithm discussion in Chapter 3. This concept is implicit in the analyses presented to the system, since only animate objects are assigned to the

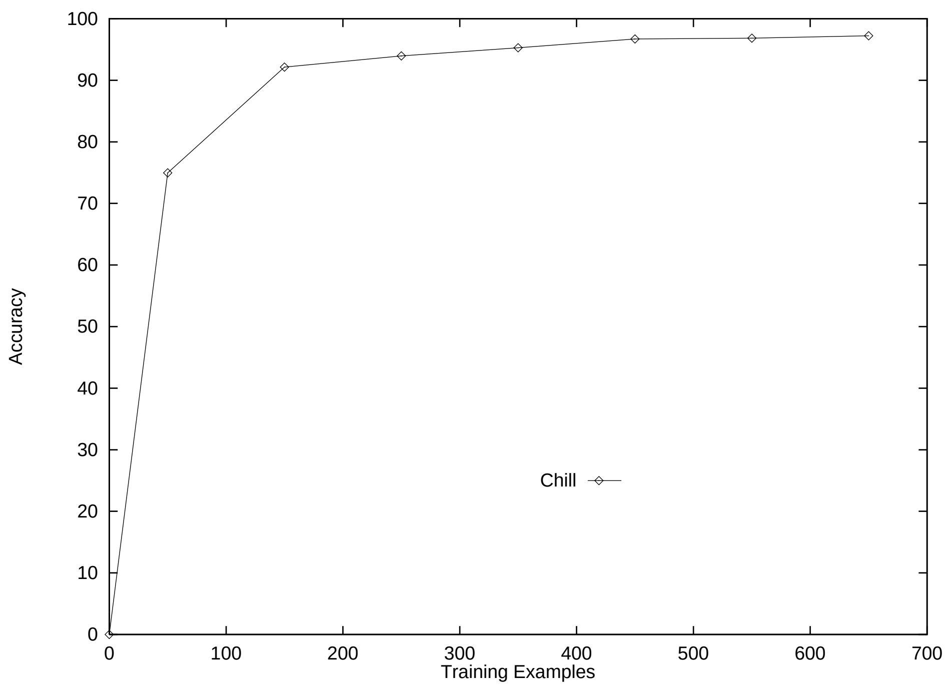

Figure 5.3: M&K Accuracy with Lexical Disambiguation

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 M & K 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 M & K Corpus. Notice that this invented rule itself contains two invented word categories. These observations of the types of rules created by CHILL establish confidence that the basic mechanisms are generating "reasonable" as well as accurate generalizations over the training corpus.

5.3 Comparing to Connectionism

Direct comparison with connectionist results for the M & K Corpus is difficult, as the connectionist systems produced only a single parse for each sentence, and overall sentence accuracy is often not reported. The closest comparison can be made with the results in (Miikkulainen & Dyer, 1991) where an accuracy of 95% was achieved in assigning words to case slots after training with 1439 of the 1475 pairs. In a similar experiment (limiting CHILL to a single parse and calculating case-slot assignment accuracy), CHILL achieved the same level of performance after training on only 650 examples. Of course, one of the advantages of CHILL is the ability to produce multiple parses, as the previous results demonstrate, CHILL achieves better accuracy at the full sentence level with far less training data. With respect to training time, the creation of the parsers for the M & K Corpus required less than 30 minutes of CPU time on a SPARCstation 2. This compares favorably with backpropagation training times usually measured in hours or days.

As a further comparison to connectionist approaches, CHILL was used to duplicate a generalization experiment reported by St. John and McClelland (1990). This experiment used an artificial corpus created from ten actions and ten names with sentences in either active ("John saw Mary") or passive voice ("Mary was seen by John"), resulting in a total of 2000 such sentences. The task of the learned system was to identify the verb, agent and patient of novel sentences. When trained with 1750 of the 2000 sentences, their neural network model processed the remaining 250 sentences with 97% accuracy. An average learning curve over five trials for CHILL is shown in the graph of Figure 5.4. The testing set in this case consisted of 1900 sentences. As can be seen from the graph, CHILL significantly outperforms its neural counterpart, achieving 100% accuracy after training on only 90 examples.

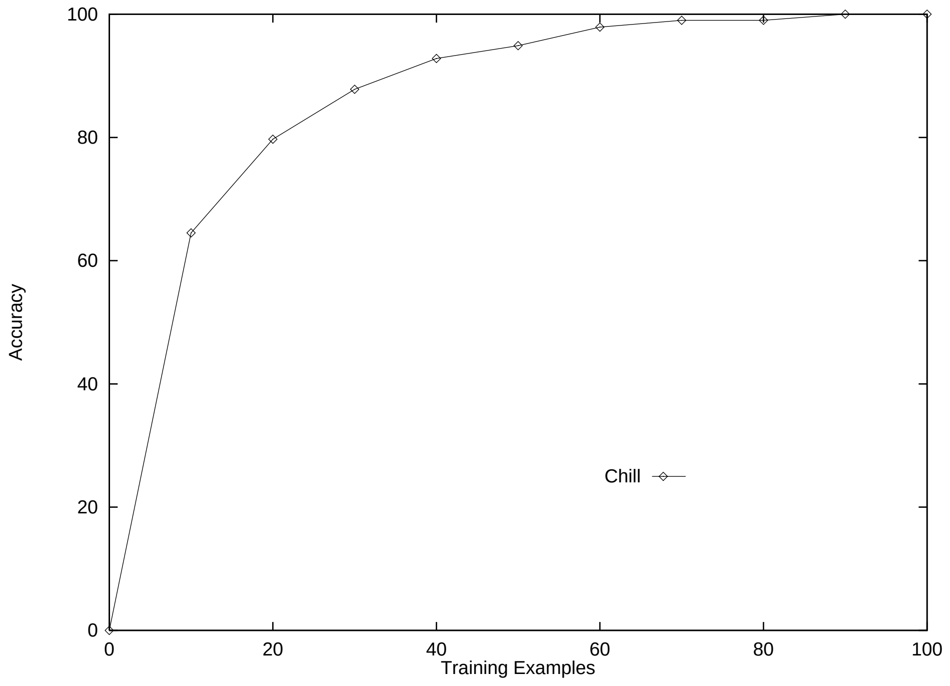

Figure 5.4: Active/Passive Corpus Accuracy

One must be cautious in making such comparisons as the types of inaccuracies exhibited by CHILL and neural approaches differ substantially. Neural networks always produce an output many of which contain minor errors, whereas CHILL tends to produce a correct output or none at all. From an engineering standpoint, it seems advantageous to have a system which "knows" when it fails; connectionists might be more interested in failing "reasonably." Indeed, a major motivation for much work in neural networks is the modeling of human cognitive function. For example, Miikkulainen (1995) has recently proposed a neural architecture called SPEC for processing recursive clause structure. A "feature" of this network is that it has difficulty learning deep center embeddings [1] in the absence of strong semantic constraints: a property

[1]For example, people understand "The girl who the dog bit liked the boy," but have trouble with deeper embeddings as in "The girl who the dog who the cat chased bit liked the boy."

that is well documented for humans. Experiments with the Spec corpus have proven that Chill is a very efficient learner in this limited domain, requiring fewer than 30 random examples to learn parsers with 100% accuracy on novel sentences. However, Chill automatically generalizes to sentences with any level of center embedding, and therefore fails to model this aspect of human language learning/processing mechanisms.

5.4 A Database Query Domain

These initial case-role parsing experiments were run with small, artificial corpora specifically designed to illustrate the case mapping problem. As such, they do not necessarily reflect the true difficulty of acquiring case-role parsers for natural language applications. Further experiments were designed to test Chill on a more extensive task, specifically, performing case-role parsing of natural language database queries. A more sophisticated approach to the database query task will be presented in Chapter 7, where Chill is used to learn parsers which map English questions directly into suitable formal queries. Here we are mainly concerned with further evaluation of the ability to learn case-role parsers.

A corpus of examples was created by "lifting" a portion of a semantic grammar from an extant prototype natural language database designed to support queries concerning tourist information (Ng, 1988). The portion of the grammar used recognized over 150,000 distinct sentences. A simple case grammar, which produced labelings deemed useful for the database query task, was devised to generate case-structure analyses. The example pair shown in Figure 5.5 illustrates the type of sentences and analyses used. A corpus of examples was created automatically by performing a random walk through the database grammar assigning equal likelihood to the possibilities at choice-points.

A learning curve for this corpus is shown in Figure 5.6. The curve depicts a five trial average of generalization results for 500 sentences which differed from any

```
Show me the two star hotels in downtown LA with double rates
below 65 dollars.

[show, theme:[hotels, det:the,
                      type:[star, mod:two],
                      loc:[la, casemark:in, mod:downtown],
                      attr:[rates, casemark:with, mod:double,
                                   less:[nbr(65), casemark:below,
                                                  unit:dollars]]]
       dative:me]
```

Figure 5.5: Case-structure Example from Tourist Domain

used in training. The results are very encouraging. With only 50 training examples, the resulting parser achieved 93% accuracy on novel sentences. With 300 training examples, accuracy is 99%. This suggests that the relative lack of ambiguity in this sample task tends to compensate for the larger example space, resulting in excellent generalization to unseen cases.

These results are certainly encouraging, but not surprising. The corpus is still quite regular and stylized, having been produced by a semantic grammar. It does however show the ability of CHILL to easily "reverse engineer" simple case-role grammars with very high accuracy. The next two chapter investigate the use of CHILL for much more sophisticated language processing tasks.

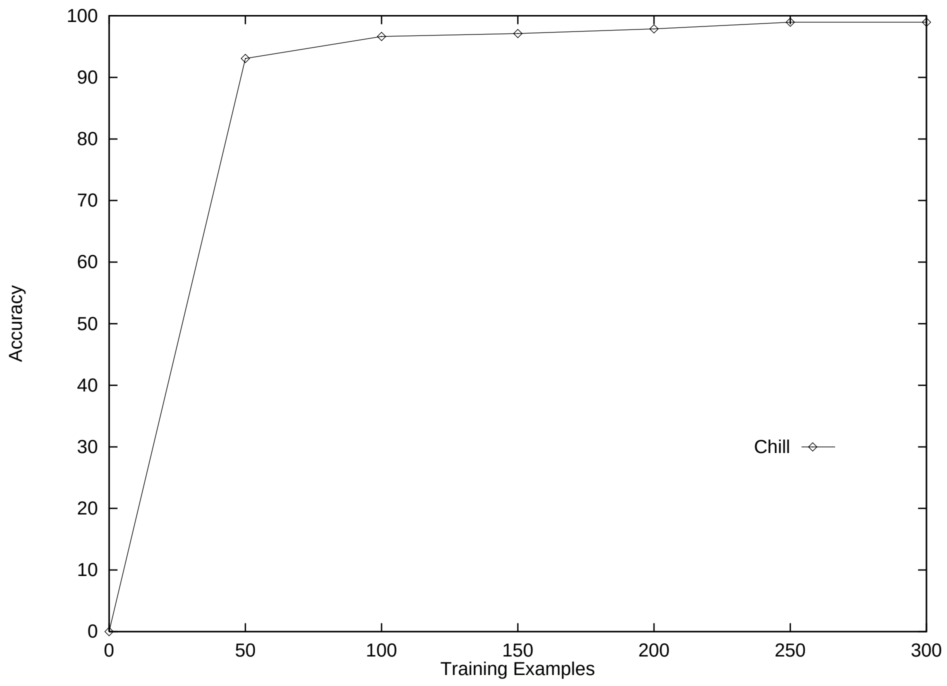

Figure 5.6: Tourist Domain Accuracy

Chapter 6

Experiments with Syntactic Parsing

Most previous work in corpus-based NLP has not dealt with case structures, but rather has concentrated on more surface-oriented issues such as word-class tagging and syntactic parsing. In order to compare with these approaches, a series of experiments was conducted using CHILL to generate syntactic parsers. We selected as a test corpus a portion of the ATIS dataset (specifically the file `ti_tb`) from a preliminary version of the Penn Treebank (Marcus et al., 1993). One of the major motivations for choosing this particular corpus is that it has been used in a number of recent studies in automated parser acquisition (Brill, 1993; Periera & Schabes, 1992) which serve as a convenient benchmark for evaluating the performance of CHILL.

6.1 Producing Syntactic Analyses

6.1.1 The ATIS Corpus

The ATIS corpus contains queries regarding air-travel information. The original questions were obtained using "Wizard of Oz" techniques, so they represent realistic human-computer interaction. Syntactic analyses have been provided by the annotators of the Penn Treebank project. An example analysis of the sentence, "Show me the flights that served lunch departing from San Francisco on April 25th" is shown in Figure 6.1. 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. The full corpus used in these experiments contains contains 729 sentences with an average length of 10.3 words.[1]

```
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 6.1: Example Treebank analysis

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 learning binary-branching structures. This simplifies the parser and allows for a more direct comparison to previous approaches (Brill, 1993; Periera & Schabes, 1992) which use binary bracketings.

Making the treebank analyses compatible with the binary parser required completion of the parses into binary-branching structures. This "binarization" was

[1]The preliminary version of the Treebank is far from perfect. A number of the parses contained unbalanced bracketings or improper constructions. Sentences with questionably formed representations were dropped from the corpus for the purposes of these experiments.

accomplished automatically by introducing special internal nodes in a right-linear fashion. For example, the noun-phrase, `np:[the,big,orange,cat]`, would be binarized to create: `np:[the,int(np):[big, int(np):[orange, cat]]]`. The special labeling (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. All of the experiments reported below were done with automatically binarized training examples; control rules for the artificial internal nodes were learned in exactly the same way as for the original constituents.

6.1.2 Using Chill for Syntactic Analyses

The learning component of Chill remained exactly the same as in the case-role experiments except that the initial Parsing Operator Generator and Example Analysis was modified for the for the syntactic analyses of the Penn Tree-bank. 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$]]]$. The analysis is represented 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).`

Clearly, this representation meets the criterion of operator transparency. An examination of the constituents in a parse directly reveals the reduction operators which are required for its construction. In the case of analyses including empty categories (detectable as lexical tokens that appear in the analyses, but not in the sentence), each empty marker produces an extra shift operator to introduce the marker without consuming a word from the input buffer.

As in the case-role parsing framework, derivation transparency was achieved operationally. Since, the corpus only provides a single parse for each sentence, CHILL was applied in the single-parse mode. As explained above in Section 3.3.2, this means that fewer negative control examples were generated. The operators were placed in order of increasing frequency as indicated by the training set to allow for default effects. The first correct parsing found for a sentence was taken to be the correct derivation for the purpose of generating control examples.

To address the issue of training tractability, the version of the overly-general parser used during training includes checks to insure that that the current state of the parse is consistent with the final representation. The consistency checking process is trivial in the case of these syntactic representations, as the order of elements in the sentence is preserved in the frontier of the parse tree. The exact sequence of states that the parser must follow can be derived by a post-order traversal of the parse tree. Visiting a leaf results in a shift operation, while visiting interior nodes results in a reduce operation for the node. By first computing the correct sequence of parse states, it is then possible to "simulate" the action of the shift-reduce parser deterministically, insuring that each operator application produces the correct next state in the sequence.

6.2 Intial Experiments

Intial experiments (as reported in (Zelle & Mooney, 1994b)) were actually carried out on four different variations of the corpus. A subset of the corpus comprising sentences of length less than 13 words was used to form a more tractable corpus for systematic evaluation and to test the effect of sentence length on performance. The restricted corpus contains 536 sentences averaging 7.9 words in length. A second dimension of variation is the form of the input sentences and analyses. Previous experiments in parser acquisition with this corpus have operated sentences and trees containing lexical tags (parts-of-speech) rather than words. Since CHILL has the ability to create its own categories, it can use either sentences and trees with lexical tags (tagged trees) or those with just words (untagged trees). In order to test the advantage gained by tagging, we also ran experiments using both tagged and untagged trees for both the full and restricted corpus.

6.2.1 Experimental Method

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. Sometimes, however, a parse can be useful even if it is not perfectly accurate; the treebank itself is not entirely consistent in the handling of various 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 operator in the training set, shift serves as a sort of default when no reduction action applies. Therefore, at the time of impasse, all of the words of the sentence will be on the stack, and partial constituents will have been built. The contents of stack reflect the partial progress of the parser in finding constituents.

Partial scoring of trees is computed by determining the extent of 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 for the parse. The accuracy of the parse is then computed as $Accuracy = (\frac{O}{Found} + \frac{O}{Correct})/2$ where O is the overlap score, $Found$ 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 parse that do not cross any constituent boundaries in the correct tree (Black, 1991). Of course, this measure only allows for direct comparison of systems that generate binary-branching parse trees.[2] By binarizing the output of the parser in a manner analogous to that described above, we can compute the number of sentences with parses containing no crossing constituents, as well as the proportion of constituents which are non-crossing over all test sentences. 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 labeled parses, and is not directly optimized for the bracketing task.

6.2.2 Results

The results of these preliminary experiments are summarized in Figures 6.2 through 6.5. The figures for the restricted length corpus in reflect averages of three trials, while the results on the full corpus are averaged over two trials. The curves depict results for each of the four metrics outlined above. *Correct* is the percentage of test

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

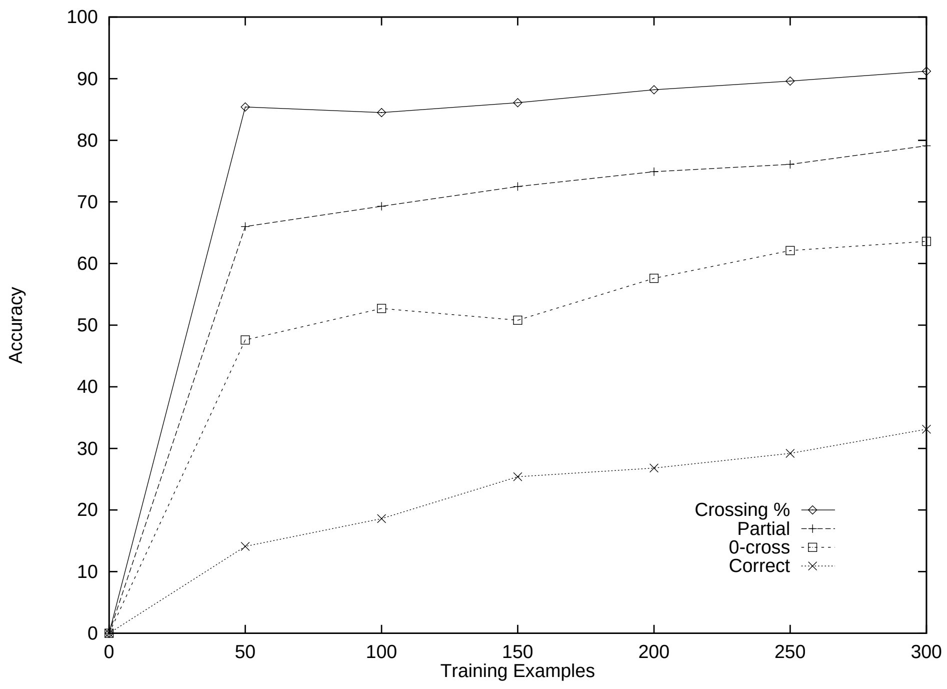

Figure 6.2: Preliminary ATIS: Restricted Corpus with Lexical Tags

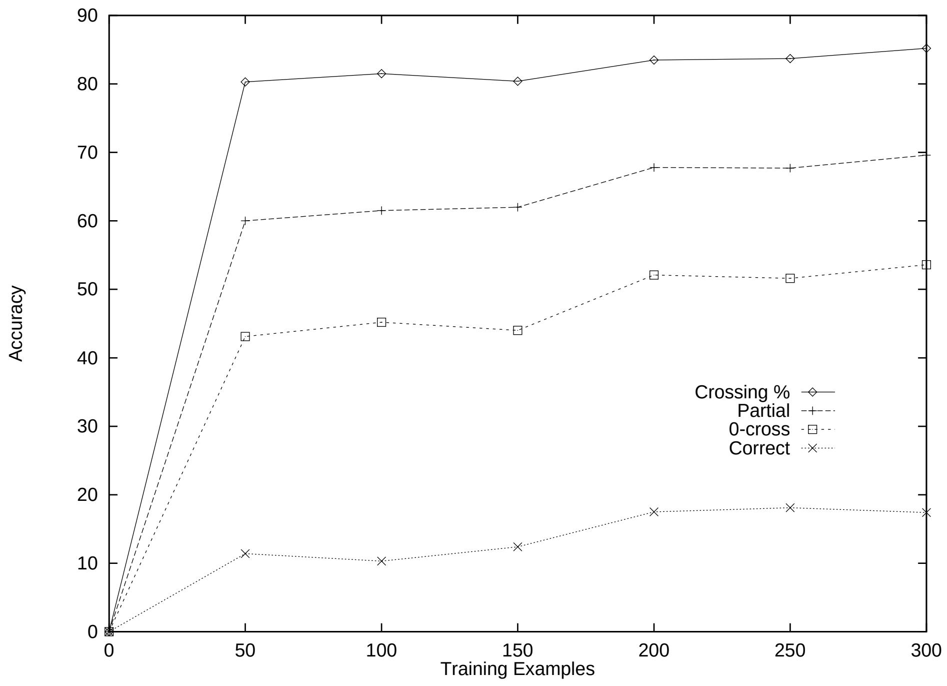

Figure 6.3: Preliminary ATIS: Restricted Corpus with Words

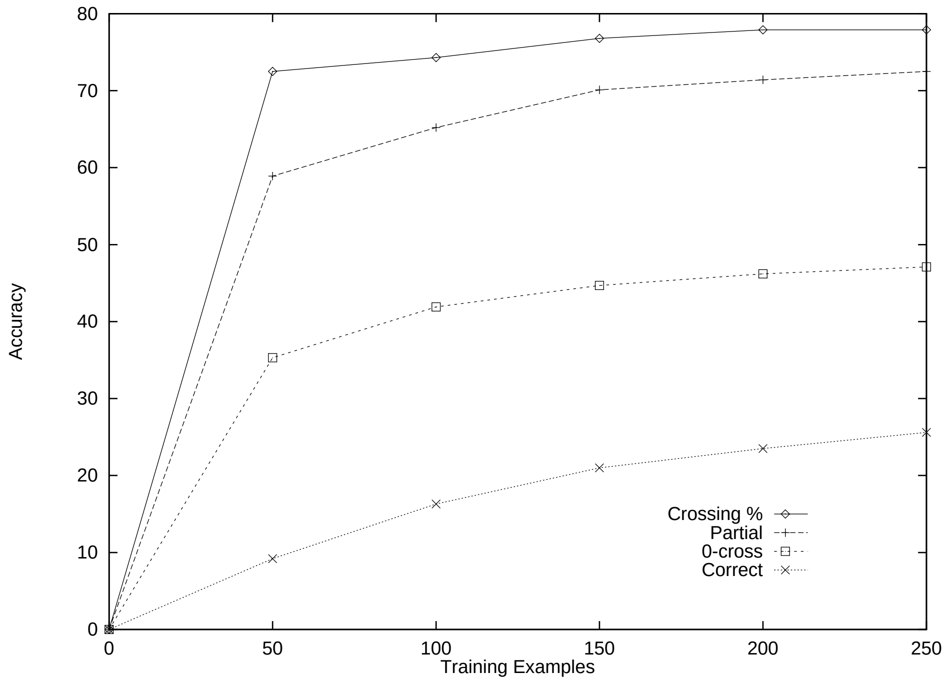

Figure 6.4: Preliminary ATIS: Full Corpus with Lexical Tags

sentences with parses that matched the treebanked parse exactly. *Partial* is partial correctness using the overlap metric. The remaining curves reflect measures based on re-binarizing the parser output. *0-Cross* is the proportion of test sentences having no constituents that cross constituents in the correct parsing. *Crossing%* reports the percentage of (binarized) constituents that are consistent with the treebank (i.e. cross no constituents in the correct parse).

These inital results, representing the simplest approach to construcing operators for this domain were encouraging. While we know of no other results for parsing accuracy of automatically constructed parsers on this corpus, the figures of 33% completely correct using the tagged input and 17% on the untagged text seem quite good for a relatively modest training set of 300 sentences. The figures for 0-cross and crossing% are about the same as those reported in studies of automated bracketing for the unrestricted ATIS corpus (Brill (1993) reports 60% and 91.12%, respectively). However, the CHILL bracketing measures for the unrestricted corpus

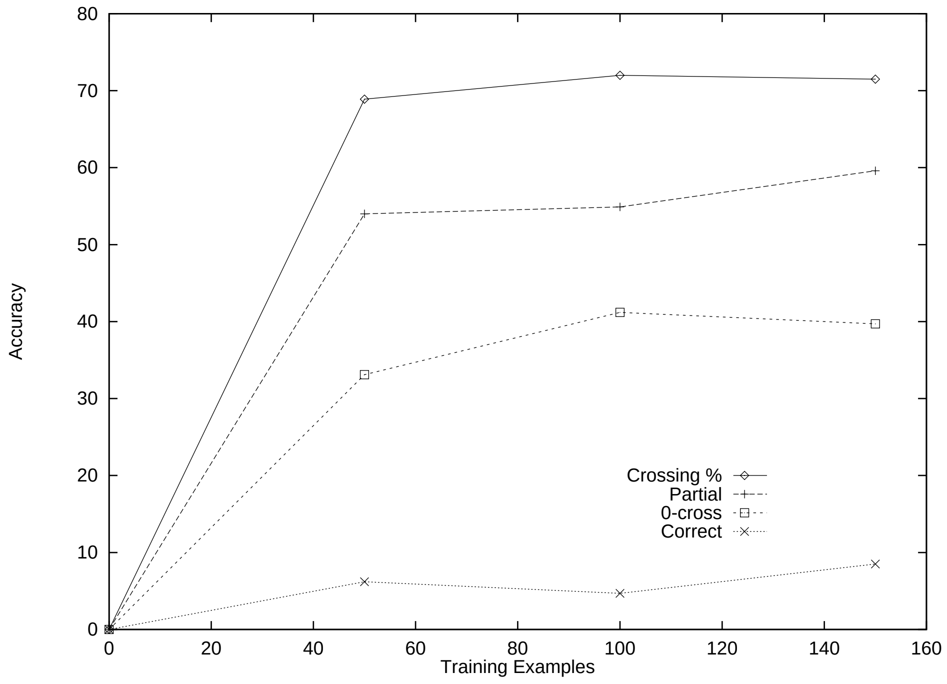

Figure 6.5: Preliminary ATIS: Full Corpus with Words

were not as good.

A comparison of Figures 6.2 and 6.3 shows 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 raw-text use no special mechanism for handling previously unseen words occurring in the testing examples. Achieving 60% (partial) accuracy under these conditions seems promising, if not spectacular. 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. Likewise, the transformational approach of (Brill, 1993) is limited to bracketing strings of lexical classes, not words. A major advantage of the Chill approach is the ability of the learning mechanism to automatically construct and attend to just those features of the input that are most useful in guiding parsing.

Figures 6.4 and 6.5 shows results for the full corpus. As one might expect, the results are not as good as for the restricted set. CHILL did not do as well on measures of bracketing performance as had been reported for previous systems. Even though CHILL was designed to perform a much more difficult task (producing complete labeled parse trees), this result was disappointing.

6.3 Improving the Results

6.3.1 Specializing the Operators

An examination of the rules produced by CHILL suggested that a major problem was that the complexity of the control rules was overwhelming the induction algorithm. CHILLIN was having difficulty extracting generalizations from the control examples, resulting in substantial memorization of specific contexts. Recall that the initial generalizations in CHILLIN are produced by considering a random sampling of example pairs. If the concept definition to be learned has limited disjunction, it can be shown that a relatively small sampling is likely to yield some pairs with examples that are covered by a single clause of the desired definition. As the number of clauses in the target concept increases, the pair sample-size must be increased.

One approach to learning better parsers then, would be to simply increase the sample-size parameter in CHILLIN. Unfortunately, this approach caused CHILLIN to run up against time and memory limits which prevented learning of parsers for the unrestricted corpus. An alternative approach is to somehow reduce the degree of "disjunctiveness" in the control rules being learned. Better results were obtained by making the operators more specific, 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`.

6.3.2 Results

A second set of experiments was run to determine the benefit of the operator indexing scheme in learning with the unrestricted ATIS corpus. These experiments were conducted with three versions of the unrestricted corpus: 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 (duplicating the experiments of previous bracketing systems).

The learning curves for these experiments are shown in Figures 6.6 and 6.7. These curves represent averages over 5 different splits of training and testing examples. The learning curves for the tagged and tags-only versions were not significantly different, so only the latter is shown.

These results show considerable improvement over the initial experiments with the more general operators. After training on 525 sentences, CHILL constructed parsers comprising over 1500 lines of Prolog which averaged completely correct parses for 41% of the novel testing sentences. Using the partial scoring metric, CHILL'S parses garnered an average accuracy of over 84%.

The final figures for 0-cross and crossing% compare very favorably with those reported in studies of automated bracketing for the ATIS corpus. Brill (1993) reports

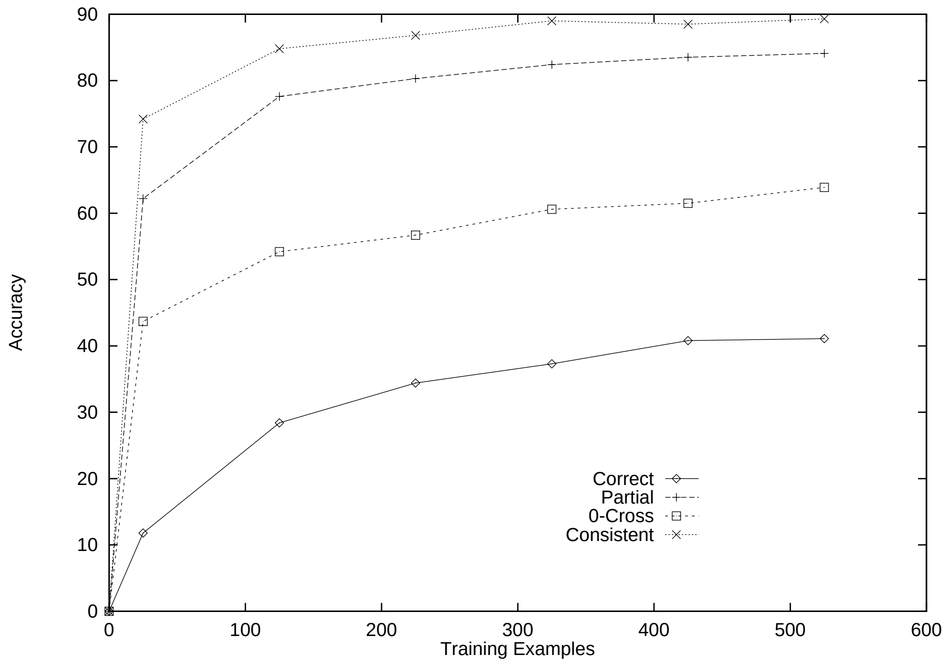

Figure 6.6: Indexed ATIS: Full Corpora with Lexical Tags

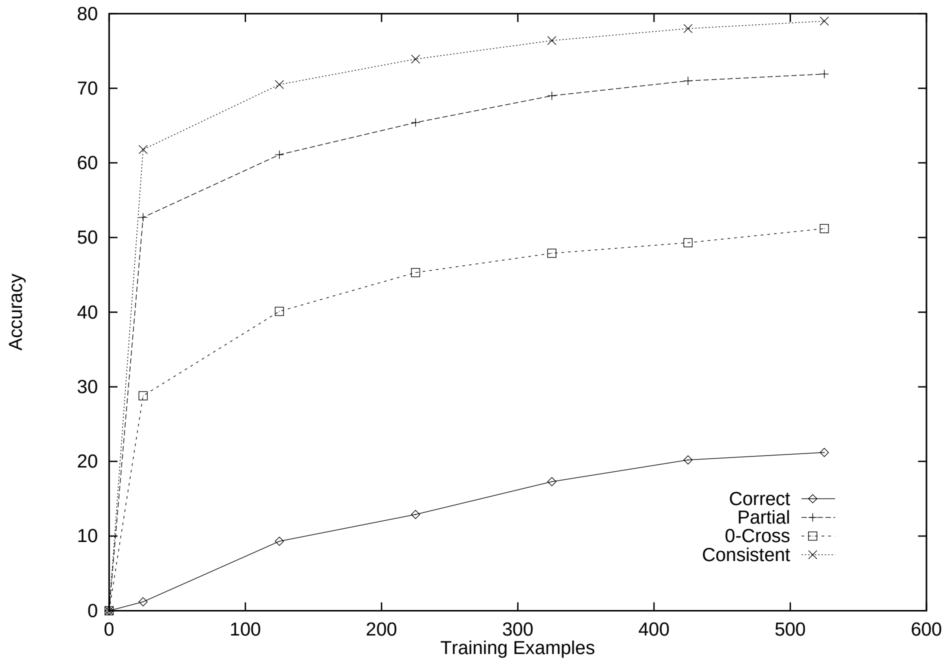

Figure 6.7: Indexed ATIS: Full Corpus without Lexical Tags

60% and 91.12%, respectively, while Periera and Schabes (1992) achieve 90.36% for the crossing% measure. CHILL scores higher on the percentage of sentences with no crossing violations (64%) and slightly lower (90%) on the total percentage of noncrossing constituents. 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 are consistent). It is interesting to note that all three systems approach similar assymptotic levels of performance on the bracketing measures. There are some differences in terms of learning efficiency, however. Brill reaches this level after seeing only 150 training sentences, while Periera and Schabes trained on 700 sentences. CHILL lies in between at 525 training sentences.

The results for the untagged version, as in the initial experiments are significantly lower than for tags. However, the overall (partial) acuracy has increased from 60% to 72% due to operator indexing and absolute accuracy has more than doubled. Basically, the indexing scheme has improved the results for the unrestricted corpus to the level previously achieved on the restricted-length version. There are no comparable results from bracketing studies, as they were not run on untagged versions of the copus.

New word categories were invented in both situations. In the untagged text experiments, CHILL regularly created categories for `preposition, verb, form-of-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 helpful in parsing of new text. These results support the view that a practical acquisition system should be able to create its own categories, as it is unlikely that independently-crafted feature systems will capture all of the nuances necessary to do accurate parsing in a reasonably complex domain.

Obviously, there is still considerable room for improvement in these results, however it is noteworthy that CHILL achieve results as good or better than state-of-the-art empirical systems for bracketing constituents on this corpus. Clearly, the technique of creating more specialized operators can significantly enhance the performance of CHILL for real-world language processing. However, achieving the best possible performance on this task is probably not all that important. It is not clear that syntactic analysis *per se* is particularly useful, except that it may serve as a component of a larger natural language application. Even expert linguists may have disagreements about the proper structure to assign a sentence, the less careful analyses performed for treebanks have a surprisingly high level of inconsistency. On the other hand, even linguistically naive speakers of natural languages seldom have trouble determining the *meaning* of an utterance. Performance of this task is the one which should be optimized. The next chapter investigates the use of CHILL to induce parsers parsers which directly produce semantically-oriented outputs without relying on artificial intermediate analyses such as parse trees.

6.4 The Control-Rule Advantage

One question which has not yet been addressed is the extent to which CHILL's success depends on its general framework of acquisition as control-rule learning. Given the ability of ILP systems to induce useful logic programs, it is reasonable to ask whether CHILLIN's induction mechanism alone might be sufficient to produce the results seen here. The simplest application of ILP to parser acquisition would be to simply present postive examples for the `parse(sentence, rep)` relationship to the induction algorithm and let it create a logic program defining this concept. Subsequent parsing could be performed by presenting the learned program with goals having the second argument uninstantiated, effectively producing parses of sentences provided as input. The advantage gained by the control-rule framework can be assessed by comparing CHILL to the performance achieved by CHILLIN learning the `parse` relation directly.

Creating sets of positive examples for CHILLIN is trivial, we use the same top-

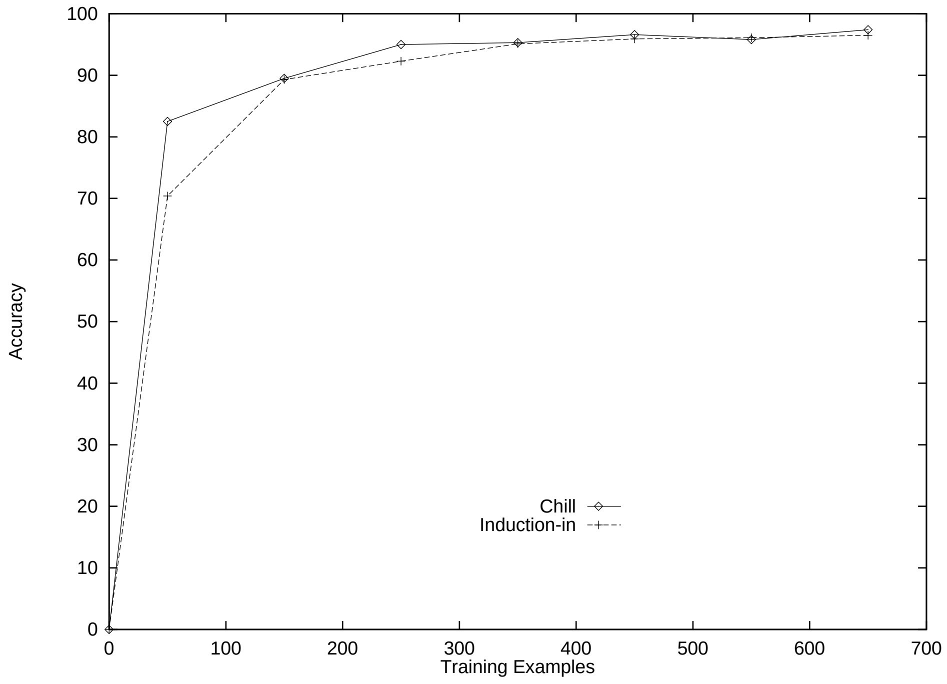

Figure 6.8: Chill vs. Induction on Simple Case-role Task

level examples as were used for training Chill. Creating negative examples is more problematic; it is clearly intractable to generate all possible sentences paired with all possible incorrect representations of those sentences; therefore, we used a mode declaration `parse(+,-)` and Chillin's implicit negatives feature (see Section 4.3).

Using this technique of induction with implicit negatives, the experiments with the simple M&K case-role representations were re-run. Figure 6.8 shows the resulting accuracy compared to that achieved by Chill. The results show that the induction algorithm alone does very well, performing slightly worse than Chill for small examples sets, and becoming indistiguishable as the sample grows. Inspection of the resulting programs showed that the induction algorithm was fairly accurately re-creating the template-and-filler style of program which was used to generate this simple corpus. This provides strong evidence of the power of the ILP induction algoritm for inducing programs from examples exhibiting regular structure.

However, on less-structured, real-world corpora, the advantage of the control-

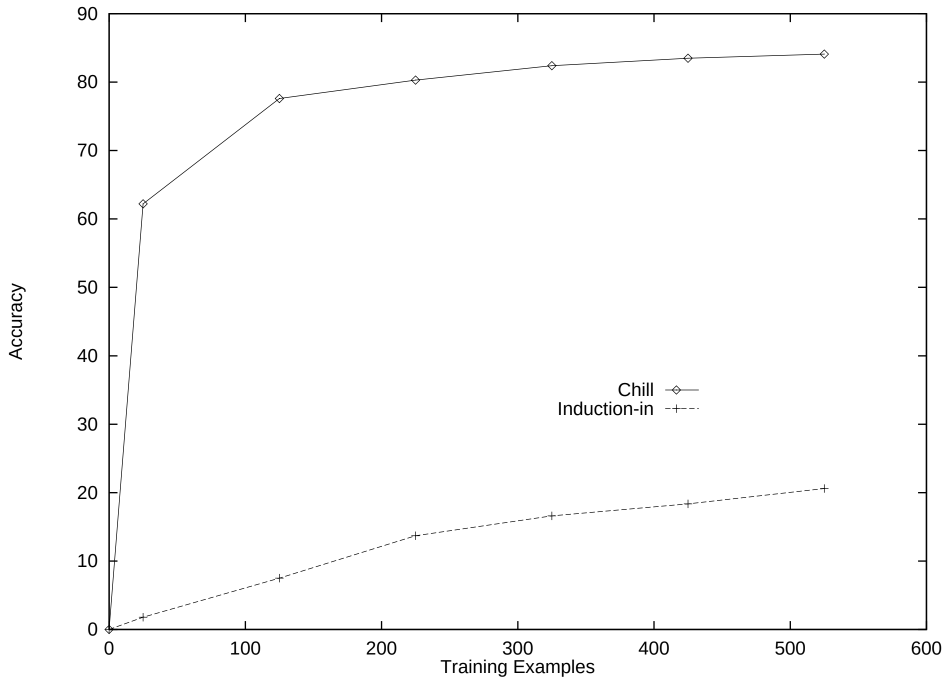

Figure 6.9: Chill vs. Induction on ATIS Tags-only

rule framework becomes readily apparent. Figure 6.9 shows the results for the partial accuracy metric in the ATIS experiment with lexical tags. Here Chill has an overwhelming advantage, achieving 85% accuracy compared to the 20% accuracy of induction with implicit negatives. Clearly, providing the shift-reduce parsing framework significantly eases the task of the inductive component.

Chapter 7

Experiments with Database-Query Parsing

7.1 Motivation

Evaluating empirical parser acquisition systems strictly on the basis of artificial metrics such as those presented in the previous chapters, though certainly valuable, is open to a number of criticisms. While the metrics can provide rough comparisons of relative capabilities, such comparisons must be made 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. Also it is not clear how one should compare systems that are tuned for producing different results (e.g. those that produce bracketings and those that build labeled parse trees).

It is even less clear that these measures will accurately reflect differences in performance on real language-processing tasks. It is not necessarily the case that a system scoring 80% on some parsing metric will actually produce better results than one achieving 70%, unless the desired task is itself parsing, something that even English teachers no longer seem to regard as particularly important. The acid test is whether empirical approaches allow the construction of better natural language systems, or perhaps allow designers to build comparable systems with less time and expertise. Testing the true promise of CHILL requires marrying the learning component with a performance component that uses the learned parsers to actually solve a natural language processing problem. To that end, this chapter reports on the experience of using CHILL to engineer a natural language front-end for a simple database.

The choice of a database-query application was motivated by a number of concerns. First, it is a significant real-world language-processing problem. The database-query task has long been a touchstone in NLP research. The potential of allowing inexperienced users to retrieve useful information from huge computer archives was recognized early-on as an important application of computer understanding. The task has served as the test-bed for important NLP formalisms and approaches, from early work in augmented transition networks (Woods, 1970) and semantic grammars (Brown & Burton, 1975; Hendrix, Sagalowicz, & Slocum, 1978) to modern logic grammars (Warren & Pereira, 1982; Abramson & Dahl, 1989). Current research on the ATIS domain discussed in the previous chapter demonstrates the on-going interest in this problem. Moreover, the need for natural language interfaces is likely to grow as ever more information is made available online to naive users who desire access to that information in familiar ways.

Second, database querying represents a language-processing task of tractable size and scope for this evaluation. Since the understanding problem is limited by the domain of the database and the information retrieval task, it is reasonable to hope that good coverage can be obtained without having to compile a corpus containing tens of thousands of sentences, a task beyond the scope of this work. On the other hand, creating a system that performs well on novel sentences is a nontrivial task, as any student who has done such a project in an AI class can attest.

Finally, and perhaps most importantly for the purposes of this dissertation, a parser for database queries is easily evaluable. The gold standard is whether the system produces the correct output for a given question, a determination which is straight-forward for most database domains. There is no need to fret about partial metrics or engage in philosophical debates over the usefulness of various intermediate representations.

7.2 Overview

The database query parser produces analyses that are substantially different from the types presented so far. The previous representations have been syntactic or shallow semantic structures. In the database-query task, the meaning of a sentence is represented operationally as a query in a suitable database query language. The query language considered here is a logical form similar to the types of meaning representation typically produced from 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 a database.

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 7.1 shows a sampling 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.

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 7.1: Sample Database Queries

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 meeting the technical criteria necessary for CHILL. 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. Typically, a construct of the query language requires a single clause in the interpreter to define how it accesses facts from Geobase.

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

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 7.2: Basic Objects in Geoquery

question, resulting in a corpus of 250 pairs which was used to evaluate the performance of Chill on this task.

As with the previous parsing frameworks, the parsing of database queries is most easily described in terms of specific examples. To that end, the next section describes the particular vocabulary of Geoquery before taking up the description of the operators and algorithms used in learning parsers for this style of representation.

7.3 The Query Language, Geoquery

The query language considered here 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 MRL.

7.3.1 Basic Constructs in Geoquery

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 7.2. 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 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. Analogous convention should be used to differentiate rivers of the same name, but the information in Geobase is insufficient to make the necessary distinctions. Fortunately, there are few duplicate rivers present.

The basic relations are shown in Figure 7.3. The meanings of these relations should be self-evident. One possible exception is the relation *equal/2*. This predicate really amounts to logical equality. 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. The approach of using a single, special predicate to introduce ground terms is also necessary for the uniform treatment of variable bindings in the parsing framework for logical queries, which will be discussed in the next Section 7.4.

7.3.2 Meta-Predicates in Geoquery

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 7.4. These predicates are distinguished in that they

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 7.3: 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 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 7.4: Meta-Predicates in Geoquery

take completely-formed conjunctive goals as one of their arguments. The remaining arguments are variables, as in the case of the basic predicates (except for `equal/2`).

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 next group of meta-predicates are used to select certain extremal elements of sets implicitly defined by goals. For example, executing the query, `largest(P, (state(S), population(S,P)))` binds `S` and `P` to values which maximize the size of `P` over all solutions to the conjunction. In this case, the resultant bindings would be: `S = stateid(california)` and `P = <population of California>`. The goal in these queries is actually executed over independent dummy variables before the resulting bindings are unified with the variables given in the query. This insures that the meta-predicate behaves "logically" in the sense that its position within a conjunct does not affect the meaning of the conjunct. The query `(capital(S,C), largest(P, (state(S), population(S,P))))` produces exactly the same bindings for S and P as before.

The final group of meta-predicates operate on the cardinality of implicit sets. `Count/3` simply counts the number of unique bindings for its dummy variable `D` which occur in solutions to `Goal`. It is used in some questions about number as in "How many states border Texas," which translates to: `answer(C, (count(S,(state(S), next_to(S,S1), equal(S1,stateid(texas))))))`. The final two meta-predicates bind indices for implicit sets with extremal cardinalities. For example, the phrase, "the state with the fewest rivers" translates to `fewest(S, R,(state(S), loc(R, S), river(R)))`. Note that S is the only variable which is bound by executing this goal, all other arguments are treated as internal dummy variables over which quantification is performed.

7.3.3 Discussion

Obviously, Geoquery does not represent a general solution to the problem of representing the meanings of natural language utterances. It does, however, provide a simple language for expressing a great number of interesting queries about geography. A similar approach could be taken for other domains by defining a suitable set of basic predicates. Many of the meta-predicates in Geoquery would be useful beyond the geography domain, but query languages for other domains would almost certainly necessitate extension and modification. Given the lack of agreement on a general MRL for natural language, it seems likely that engineering these sorts of problem-specific representations will remain a *modus operandi* of NLP for some time to come.

It should be emphasized that no part of Geoquery was specifically designed to simplify the learning of parsing control rules. The general framework for parsing into logical representations requires only that the representation be composed of predicates and meta-predicates, and that ground terms are introduced using `equal/2`. The design of similar domain-specific MRLs could easily proceed by analogy to Geoquery, requiring little understanding of the general parsing framework that produces these representations.

One notable deficiency in the current implementation of the Geoquery interpreter is that queries may execute very inefficiently. This is not a problem for the limited database used in these experiments, but would require attention in moving to larger applications. This is not really a natural language issue, but rather a matter of database-query optimization. Once questions have been translated into an unambiguous logical form, they are amenable to optimization or, perhaps, translation into more common query representations such as SQL. Domain independent optimization methods allow the design of the query language to be based on expressiveness and simplicity, rather than worrying about implementation issues. This is the approach taken in CHAT-80 (Warren & Pereira, 1982).

7.4 A Parsing Framework for Logical Queries

Although the logical representations of Geoquery look very different from parse-trees or case-structures, 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.

7.4.1 Overview

As before, we have adopted a simple shift-reduce parsing framework for producing database queries. The parser is again implemented by translating parsing actions into operator clauses. One cosmetic change from the previous examples is that the stack and input components of the of the parse state have been packaged into a single unit using the functor `ps/2`. This was done purely for the convenience of passing complete parse states as a single parameter in the code implementing the training module. Operators are thus expressed as a two-place predicate that transforms an input context into an output context.

The construction of logical queries involves three different types of operators. 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. Initially, such 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.

During the course of parsing, logical variables in queries are treated as ground terms. It is beneficial that variables which have not yet been unified with other variables be automatically identifiable, they are represented with the constant `freevar`. Shared variables take the form `pvar(`n`)` where n is an integer that "tags" the variable. Tags are assigned according to the order in which `freevar`s are co-referenced in the course of a parse, starting with 0. The treatment of parse variables as ground terms is required so that the portions of the parse state over which induction is performed in CHILLIN are initially ground. An added benefit of this scheme is that it provides a single correct naming of query variables, so that deducing the equivalence of structures reduces to a simple equality check.

Figure 7.5 shows the sequence of states the parser goes through in parsing the sentence, "What is the capital of Texas?" 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(pvar(0), (capital(pvar(1), pvar(0)), equal(pvar(1), stateid(texas))))`. This is turned into a valid Geoquery representation by replacing the parse variables with "live" Prolog variables, producing the final result: `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 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

```
        Parse State                                                    Operation Type

1.  ps([answer(freevar,freevar):[]],
       [what,is,the,capital,of,texas,?])                               shift
2.  ps([answer(freevar,freevar):[what]],
       [is,the,capital,of,texas,?])                                    shift
3.  ps([answer(freevar,freevar):[is,what]],
        [the,capital,of,texas,?])                                      shift
4.  ps([answer(freevar,freevar):[the,is,what]],
       [capital,of,texas,?])                                           introduce
5.  ps([capital(freevar,feevar):[],
        answer(freevar,freevar):[the,is,what]],
       [capital,of,texas,?])                                           co-reference
6.  ps([capital(freevar,pvar(0)):[],
        answer(pvar(0),freevar):[the,is,what]],
       [capital,of,texas,?])                                           shift
7.  ps([capital(freevar,pvar(0)):[capital],
        answer(pvar(0),freevar):[the,is,what]],
       [of,texas,?])                                                   shift
8.  ps([capital(freevar,pvar(0)):[of,capital],
        answer(pvar(0),freevar):[the,is,what]],
       [texas,?])                                                      shift/introduce
9.  ps([equal(freevar,stateid(texas)):[texas],
        capital(freevar,pvar(0)):[of,capital],
        answer(pvar(0),freevar):[the,is,what]],
       [?])                                                            co-reference
10. ps([equal(pvar(1),stateid(texas)):[texas],
        capital(pvar(1),pvar(0)):[of,capital],
        answer(pvar(0),freevar):[the,is,what]],
       [?])                                                            conjoin
11. ps([equal(pvar(1),stateid(texas)):[texas],
        answer(pvar(0),capital(pvar(1),pvar(0))):[the,is,what]],
       [?])                                                            shift
12. ps([equal(pvar(1),stateid(texas)):[?,texas],
        answer(pvar(0),capital(pvar(1),pvar(0))):[the,is,what]],
       [])                                                             shift
13. ps([´$$EOI´,
        equal(pvar(1),stateid(texas)):[texas],
        answer(pvar(0),capital(pvar(1),pvar(0))):[the,is,what]],
       [])                                                             conjoin
14. ps([´$$EOI´,
        answer(pvar(0),(capital(pvar(1),pvar(0)),
                        equal(pvar(1),stateid(texas)))):[the,is,what]],
       [])
```

Figure 7.5: Sequence of Parse States for “What is the capital of Texas?”

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, `'$$EOI'`, 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. If the argument is not a `freevar`, the new structure is conjoined with the existing argument. States 11 and 14 are the product of `conjoin` operations.

7.4.2 Parsing Operators and Transparency

Using CHILL to produce a parser for logical query representations requires suitable algorithms for operator generation and example analysis, which are described in this section. Recall that operator transparency requires that the overly-general Geoquery operators be automatically deducible from training examples. As in the previous parsers, each required operator will produce a single overly-general clause for the `op` predicate. Building on the lessons learned from the experiments with the ATIS corpus, the operators for parsing task were designed to be relatively specific. Each class of operation, `introduce`, `co-reference` and `conjoin`, will give rise to multiple specific `op` clauses as dictated by the representations appearing in a training corpus.

Introduce: Generating Pieces of Structure

An `introduce` operation generates a stack item corresponding the some substructure of the complete parse. The introduced substructures are at the level of simple terms, with arguments filled initially by `freevar`. In general, an `introduce` operator is required for each functor appearing the the final query (except for answer/2 which is already in the initial state). The overly-general clause which implements a specific introduce operator takes the form, op(S0,S1) :- introduce(Term, Phrase, S0, S1) which states that Phrase introduces the structure Term into state S0 yielding state S1. More concretely, the example parse above requires an operator of the form op(S0,S1) :- introduce(capital(freevar,freevar),[capital],S0,S1) to introduce the capital/2 structure appearing in the query.

The Phrase argument of introduce/4 may be interpreted as a kind of precondition for the applicability of the operation. The words specified in the Phrase list must be at the front of the input buffer. It is easy to give a logical specification of introduce as follows:

```
introduce(Term, Phrase, S0, S1) :-
     ps_input(S0, Words),
     append(Phrase,_,Words),
     ps_stack(S0, Stack0),
     ps_stack(S1, [Term:[]|Stack0]),
     ps_input(S1,Words).
```

The ps_input/2 and ps_stack/2 predicates select the input buffer and stack portions, respectively, of a parse state. Append/3 is the "standard" predicate for list concatenation, here it is used to insure that Phrase is indeed a prefix of the words in the input buffer.

Obviously the Term argument for an introduce operation is determined by the appearance of a particular term in the query of a training example. Determining a proper Phrase argument requires an extra source of information in the form of a *lexicon.* The lexicon indicates which words or phrases correspond to various semantic structures appearing in queries. It is implemented as a simple table of lexical entries having the form: lex_entry(Phrase, Term). Currently the lexicon is constructed by an analyst with some automated help. The issue of lexicon construction is discussed in more detail in Section 7.4.4.

A closely related issue concerns the introduction of database objects, which are handled specially. Following an approach similar to that used for other terms would suggest having a separate operator for each object that appears in the training examples. Such an approach would have obvious generalization problems. It would only be able to answer questions about objects that had actually been seen in training examples, even though the existence of comparable objects is a direct consequence of the database query task. For example, if the training examples did not contain any questions about Kalamazoo, no subsequent question about Kalamazoo could be answered for lack of an operator to introduce the relevant object. Obviously, it is unreasonable to assume that every object in the database will be mentioned in the training examples. As an alternative, special operators are included for introducing each class of objects in the database.

As an example of a special operator, consider the operator which was applied to produce state 9 in the example parse. This operator applies any time the phrase at the front of the input buffer corresponds to a known state name from the database. A logical specification of this operator is as follows:

```
op(S0,S1) :-
    ps_input(S0,Input0),
    state_name(Input0,Input1,Name),
    ps_stack(S0,Stack0),
    ps_input(S1,Input1),
    ps_stack(S1,[equal(freevar,stateid(Name)):[Name]|Stack0])).
```

Here `state_name/3` holds when `Name` is an atom comprising the words of a state's name and `Input1` is the tail of `Input0` after the words corresponding to the state's name has been stripped. For example, one clause of the relation is `state_name([new, mexico| Tail], Tail, 'new mexico')`. Those familiar with definite-clause grammars will recognize this standard form for "parsing" a constituent from a sentence represented as a difference list. The definitions for predicates such as `state_name` used in special operators are generated directly from information contained in Geobase.

Similar special operators are included in the overly-general parser to introduce `equal` terms for rivers, cities, places, and countries (to handle various ways of expressing the concept "United States"). Cities are a particularly interesting case. Recall that city objects are represented using a term of the form `cityid(CityName, StateAbbrev)`. Sometimes cities are referred to only by name. The special operator for this case introduces the structure with the `StateAbbrev` argument left uninstantiated. A separate operator comes into play if the city name is immediately followed by a state name. In this case, both names are parsed and a completely instantiated structure is introduced.

These special operators generalize the notion of lexicon by directly using information in the database to generate appropriate overly-general rules for introducing various database objects. Although they are engineered specifically from the database at hand, their construction is straightforward and they provide a very general mechanism for handling large classes of similar objects; a condition which nearly defines the notion of a database.

Co-reference: Binding Variables

Co-reference operations cause two parse variables to be "unified." If both variables are `freevar` they are both replaced with a parse variable having the next available tag. If only one of the variables is `freevar`, it is replaced with the other variable. In the case that both variables are previously co-referenced, they must have the same numeric tag for the operation to succeed.

Recognition of the need for a particular co-reference operator is straightforward. The need for a co-reference operator is signalled by multiple appearances of a variable in a query. A variable which appears n times in a query of a training example gives rise to $n-1$ co-reference operators. A co-reference operator has the form: `op(S0, S1) :- coref(F1, N1, I1, F2, N2, I2, S0, S1).` The `coref` relation holds when `S0` is a parse state with the structure on the top of the stack (call it *top*) having the functor `F1/N1` and a subsequent stack item (call it *item*) having the functor `F2/N2`, and `S1` is `S0` with the `I1`th argument of *top* co-referenced with the `I2`th argument of *item*. In the example parse, the first argument of `answer/2` was co-referenced with the second argument of `capital/2`. The clause for this this general action would be: `op(S0, S1) :- coref(capital, 2, 2, answer, 2, 1, S0, S1).`

A couple of technicalities are worth noting. A formal specification of `coref/8` is somewhat messy in that this it is not a strictly logical operator; the tagging of new parse variables has the side-effect of updating the tag counter. The specification is also complicated by allowing the second functor specified in `coref` to be at any stack location beneath *top*. A sequential search is employed, and the operation is performed on the *first* matching stack item only. Finally, knowing that two variables must be co-referenced does not, by itself, determine the exact operator which is required, since it it unknown which of the two variable-containing structures will be on the top of the stack when the co-referencing is performed. This issue is handled analogously to the head identification problem in case-role parsing by generating both potential operators and determining the correct derivation operationally during example analysis.

Conjoin: Building Goals

`Conjoin` operations cause a stack item to be "dropped" into another stack item representing a meta-predicate. The specific `conjoin` operations required for parsing a training example are inferred by locating query structures (other than `equal/2`) that contain arguments that are not variables. Each conjunct of such an argument must have been inserted by an appropriate conjoin operation.

Once an item has been conjoined to another, it is deleted from its former stack position and is no longer accessible to other items for co-reference; therefore, an item may not be conjoined until any necessary co-references have been established. Obviously, this dictates that an item may not be conjoined as long as it contains freevar. More subtly, an item may not be embedded if it contains the only available (not embedded) reference to a variable to which some freevar must eventually be co-referenced. This latter case occurs commonly as "chains" of references. In the example parse above, capital/2 cannot be dropped into answer/2 (state 11) until its first argument has been co-referenced with the first argument of the equal for Texas (state 10).

In order to facilitate this chaining process, conjoin actions operate on the *second* item of the stack rather than the top. The second item is either dropped into the item below, or the item below is lifted into the second item. Any pending co-references for the item on the top of the stack will be completed before attempting conjoin operations with the second item. In a sense, this process implements a one constituent look-ahead, insuring that the top item is complete with respect to references to variables in previous items before those previous items become unavailable. This look-ahead allows the construction of complex queries without requiring operators that look inside arbitrarily nested terms to establish co-references. The case of conjoin operations concerning the final constituents of a sentence is handled by the '$$EOI' marker which occupies the top stack position, placing the final constituents in the second stack position where any necessary conjoin operations can be performed.

As this discussion indicates, the conjoin operation comes in two flavors, depending on whether the second stack item is the one being embedded or the one being embedded into. The former case is implemented by an operator of the form: op(S0, S1) :- drop_conj(F1, N1, F2, N2, I2, S0, S1) where drop_conj/7 holds when S0's stack's second item (call it *item*) has the functor F1/N1 and F2/N2 is the functor of the next item (call it *meta*) and S1 is the state that results from right-conjoining *item* with the I2th argument of *meta*. If the I2th argument of *meta* is freevar, the right-conjoin operation simply replaces it, otherwise *item* is conjoined on the right of the existing argument. Conjoining on the right preserves the relative ordering of constituents as they were introduced from the sentence. More formally, the drop_conj operation may be specified as follows:

```
drop_conj(F1, N1, F2, N2, I2, S0, S1) :-
  ps_stack(S0, [Top,Item:_,Meta:Ws|PostMeta]),
  functor(Item, F1, N1),
  functor(Meta, F2, N2),
  arg(I2, Meta, Conjunction),
  conjoin_right(Item, Conjunction, NewArg),
  change_arg(Meta, I2, NewArg, NewMeta),
  ps_stack(S1, [Top,NewMeta:Ws|PostMeta]),
  ps_input(S0, Input),
  ps_input(S1, Input).
```

Here functor/3 and arg/3 are used in the standard Prolog sense. The former checks the name and arity of the principle functor of a term, and the latter accesses a specified argument of a term. The change_arg/4 predicate produces a copy of a term with a specified argument replaced by another term.

The alternative conjoin operator, lift_conj/7 applies when a meta-predicate is the second stack item, and the item below is "lifted" into it. The operator clause is analogous to that for drop_conj/7. The operator implementation is specified as follows:

```
lift_conj(F1, N1, F2, N2, I2, S0, S1) :-
   ps_stack(S0, [Top,Meta:Ws,Item:_|PostItem]),
   functor(Item,F1,N1),
   functor(Meta, F2, N2),
   arg(I2, Meta, Conjunction),
        conjoin_left(Item, Conjunction, NewArg),
        change_arg(Meta, I2, NewArg, NewMeta),
        ps_stack(S1, [Top, NewMeta:Ws| PostItem]),
        ps_input(S0, Input),
        ps_input(S1, Input).
```

Notice that the stack item is left-conjoined in this case. Since it is lower on the stack, it must have occurred earlier in the sentence.

As with co-reference operations, identifying the need for a conjoin does not strictly determine what operator is required. That is, the conjoin required by a given query may have been produced by either a lift or a drop. Again, the solution to this indeterminacy is simple to generate both operators and allow the derivation found during example analysis to determine which is correct.

7.4.3 Training and Testing

As for the previous parsing frameworks, the Geoquery parser is incorporated into Chill by providing appropriate training and testing modules. The implementation of these modules is significantly more complicated than their syntactic counterparts due to the richness of the operators employed. To begin with, the operators themselves make use of predicates defined in the database; therefore, portions of the database itself must be included in these modules. Additionally, as the mapping between sentences and representations becomes less obvious, more processing is required to insure the tractability of both training and testing.

The Training Module

As for previous frameworks, the main concern of the training module is in implementing a version of the overly-general parser that finds correct derivations of the training examples in tractable time. This task is considerably more difficult for the Geoquery parser than for the case-role or syntactic parsing considered in previous chapters. As before, the main addition to the general parsing framework is an algorithm for consistency checking. After each operator application, the current parse state is analyzed to insure that it remains consistent with the required eventual output.

The difficulty in checking the consistency of the current parse state is that there is no strict ordering relation between the structures that appear in a query and the order in which those structures are introduced by the sentence. The corpus for Geoquery follows the convention that basic predicates appear in the same order in which the words that introduce them appear in the sentence. However, meta-predicates often play the role of quantifiers and may "float" according to the scoping requirements of the sentence. For example, the phrase "state with the largest population, " generates a query fragment having the meta-predicate `largest/2` as its principle functor; however, the only reasonable introducer for this structure is the word "largest," which appears near the end of the phrase.

In general, a query may be viewed as a tree rooted in an `answer/2` node. Internal nodes represent meta-predicates and have a child for each conjunct appearing in its embedded goal. The leaves of this tree are the basic predicates used in the query and will appear in order according to the appearance of words or phrases which introduce them in the sentence. Meta-predicates may be introduced at any point within the frontier that they dominate, including immediately before or immediately after. In the previous example, `largest/2` dominates `state/1` and `population/2` and is introduced between them. In the case of the phrase "the largest state by population" `largest/2` again dominates `state/1` and `population/2`, but is introduced before either. These observations motivate the basic consistency checking algorithm used in the Geoquery parser. The sequence of items on the stack is checked to insure it represents a prefix of a valid traversal of the query tree. A valid traversal is simply one that visits the leaves in order and visits each internal node exactly once at some point during the traversal of the frontier dominated by the node.

The Geoquery training module also incorporates a slight modification of the

Chill architecture to improve training efficiency. Recall that the initial phases of Chill are Operator Generation and Example Analysis. In Operator Generation, Chill analyzes all of the training examples to construct a complete set of operators for an initial overly-general parser. This overly-general parser is then used to perform Example Analysis which builds control examples for each operator. The Geoquery module interleaves these two phases of Chill. A single example is considered at a time: a set of operators is generated, and the example is parsed and control examples extracted from the parse using just the operators for that example. As each training example is analyzed, its control examples are merged into a global pool of control examples. Using this technique, the search for a correct parse derivation is limited to applications of just those operators which might be relevant to the example at hand, resulting in significant speed-up.

The Testing Module

The testing module for the Geoquery parser is relatively straight-forward. The basic parsing shell is the same as in the previous parsing frameworks. The parse proceeds by applying operators until a final state is reached or the parser comes to a dead-end. The former occurs when the stack contains exactly two items, with the top item being '$$EOI'. The latter occurs when a state is reached for which none of the learned operators applies. This simple parsing loop along with the support predicates for the operators comprise the basic test parser.

One complication of the Geoquery parsing operators is that the `introduce` operations increase the amount of structure on the parse stack without decreasing the size of the input buffer. It is possible (in fact likely) that an `introduce` operator may be learned with an overly-general control-rule. During the course of a subsequent parse, this may lead to a situation in which the same operator is applied over and over in an infinite regress leading to a stack overflow. In order to prevent this, the implementation of the operators in the test module is slightly altered. Cycling is prevented by keeping track of what structures have been introduced since the last shift operation. A given structure not allowed to be introduced more than once without some input being consumed. This prevents pathological growth of the stack while insuring that correct parse sequences are unaffected.

7.4.4 Discussion

The operators presented here are sufficient for producing virtually any Geoquery expression. However, there are certain types of constructions, notably those involving duplicated structures which are not handled. For example, the conjunction in "What states border Texas and Oklahoma" requires two instances of `next_to/2` which are both introduced by the word "border." In general, satisfactory handling of conjunctions often requires special considerations in NLP systems (Dahl & McCord, 1983), and no effort has been made to solve these problems in the context of Geoquery.

The framework outlined here also does not allow the use of disjunction within queries to handle a sentence such as "What states border Iowa or Missouri." Disjunction can be handled analogously to conjunction; however, this was not done as the sample queries on which Geoquery was based did not contain disjunctive queries. Similarly, explicit mention of numbers are not handled, as for example in "How many cities have more than 1,000,000 people?" Again, these sorts of queries did not appear in the initial corpus. Clearly, handling explicit numbers in a general way would involve the definition of special operators such as those which introduce database objects, but they do not seem to present any particular difficulty for the general approach. A similar situation exists for ordinal references as in "What is the second largest city in Texas?" which are also not supported in the current version.

Finally, the integration between the parsing framework for logical queries and the actual constructs of Geoquery deserves examination. As indicated above, most of the operators required to generate appropriate parses are directly inferable from the training examples. All of the specific `co-reference` and `conjoin` operators required to parse the training examples may be inferred without further knowledge. However, operators which initially map words into structures require additional information. In the case of `introduce` operators, this extra information is supplied by the lexicon which is used in operator generation. While the construction of the lexicon does require human expertise, it turns out to be a relatively minor burden, at least for the types of queries for which Geoquery has been used. Lexicon development was facilitated through use of an automated tool. Given a training corpus, this tool processes each example to insure that all of the structures present in the query could be introduced by some word or phrase in the corresponding sentence according to the relations given in the lexicon. The program then interactively prompts for trigger words or phrases for any as yet unmotivated structure. The provided phrase is then recorded in a new lexical entry. In this fashion, it is easy to quickly build a lexicon sufficient for parsing a corpus.

The other domain-specific portion of the framework concerns the special operators for introducing database objects in `equal` predicates. These operators were produced by hand-written code which retrieved the relevant lexical items from the database and produced the relevant operators. These operators were then included with those produced by the automated operator generation algorithm for use in parsing the training examples.

Given the need for some domain-specific information in the parsing framework, it seems appropriate to view the overall framework as a "shell" for the creation of database query applications. Once relevant lexical information is provided, the operator generation algorithm automatically creates most of the necessary clauses of `op/2`. While the current research has concentrated on the application of machine learning techniques for acquiring the essential grammar required for NLP applications, the use of machine learning techniques to completely automate the construction of the lexicon-related operators is a fruitful area for future research.

7.5 Experimental Results

7.5.1 Building a Corpus

Sample questions in English were obtained by distributing a questionnaire to students in undergraduate German classes. The questionnaire provided the basic information from the online tutorial supplied for Geobase, including a verbatim list of the type of information in the database, and sample questions that the system could answer. The students were then asked to write down fifteen questions that they expected the system to be capable of answering. A sample questionnaire is shown in Appendix D.

A total of 50 subjects were involved in the corpus collection process. From these subjects, a total of 484 questions were gathered. Of these, 284 were discarded, resulting in 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. By far the most common of these were questions including specific numbers, which as explained above, are not handled by the current version of Geoquery.

An automated tool was developed to help in developing analyses for the corpus. As queries were entered, they were executed by the Geoquery interpreter so that the analyst could confirm the accuracy of the result. The example pair was then passed to the lexicon development tool to insure that any necessary entries were added to the lexicon. Finally, the example pair was parsed by the training module to insure that it contained a valid ordering of constituents. Using this automated system, the first sentences were annotated at a rate of about 10 sentences per hour. As the analyst became more familiar with the representations and the lexicon neared completion this rate increased to around 30 sentences an hour. The complete lexicon for the 250 sentence corpus contains a total of 72 entries. The creation of the annotated corpus and lexicon was completed over a period of five days, although no detailed time-accounting was performed, the annotation effort probably required no more than 20 person-hours of effort.

7.5.2 Experiments

For these experiments, the corpus was split into training sets of 225 examples with the remaining 25 held-out for testing. The default parameters for CHILLIN were used as in the previous experiments. However, unlike the previous experiments, some background predicates were provided to the induction component. This background consisted of the predicates which were used in CHILL's special operators 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. Figure 7.6 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. In general, it is desirable that the rate of spurious parsings be kept to a minimum. Happily, as in the previous experiments, the parsers learned

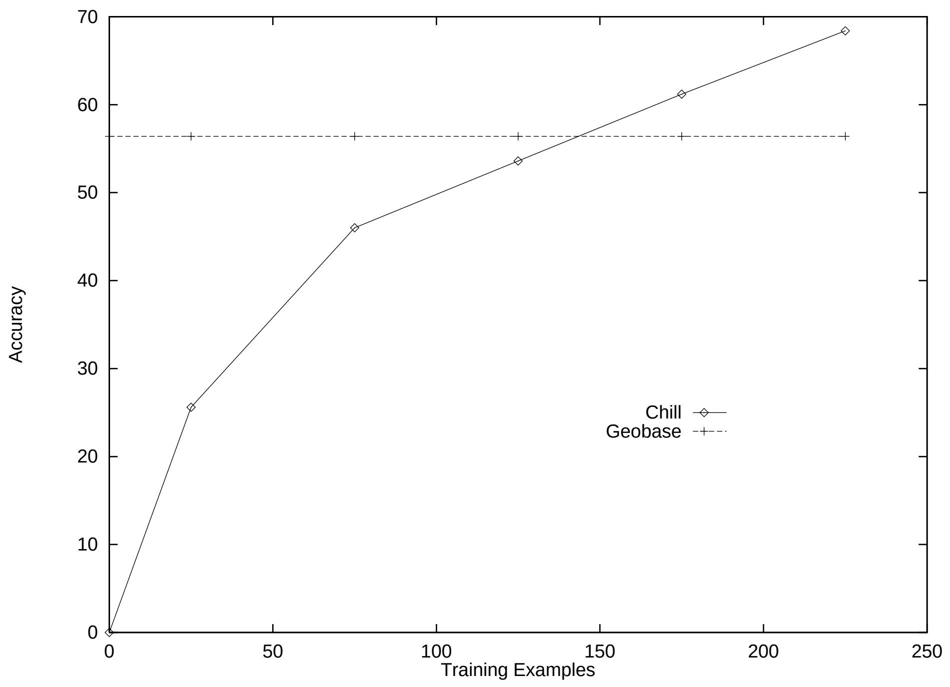

Figure 7.6: Geoquery: Accuracy

by Chill for Geoquery also produced few spurious parses. Figure 7.7 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. The specificity of the operators kept the control-rule induction problems small (the largest having only hundreds of examples) and Chill was able to learn parsers for the largest trials in less than 25 minutes of CPU time on a SPARCstation 5. Interestingly, the testing time was also minimal. While no hard statistics were recorded, the parsers produced by Chill generated queries virtually instantaneously, due to the deterministic framework.

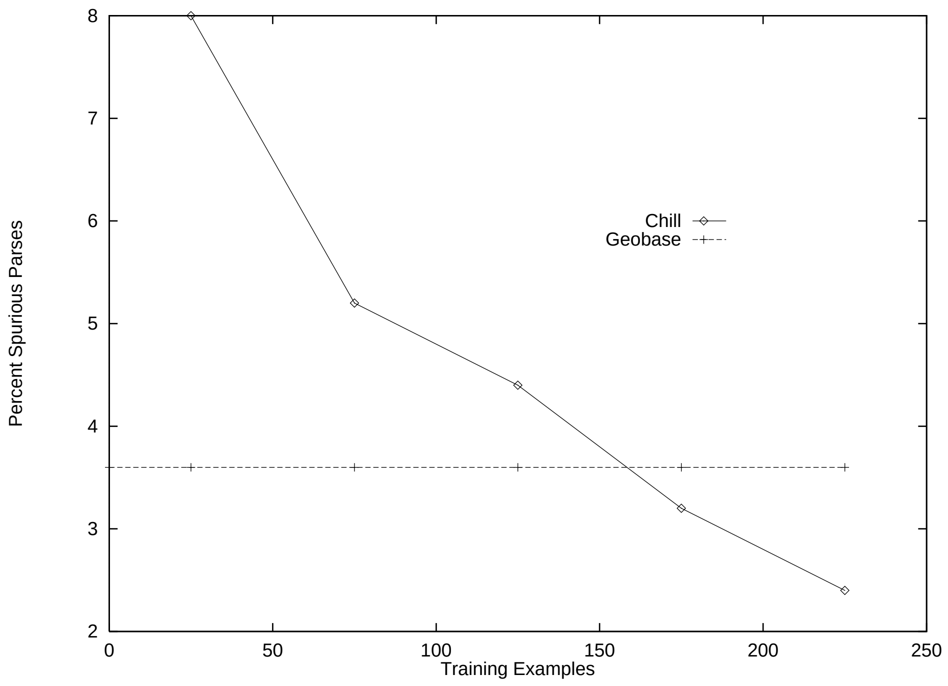

Figure 7.7: Geoquery: Percentage of Spurious Parses

7.5.3 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.

Clearly the performance of Geobase could be improved by investing time in improving its recognition grammar. The usual methodology for improving such systems is to collect a sample of sentences for which the current system fails to generate correct answers, and then attempt to modify the system to rectify the failures. Of course, changes which allow correct results for new sentences might have deleterious effects on sentences which previously parsed correctly. This leads to a sort of grammar-tweaking/regression-testing cycle attempting to insure that overall progress is being made. In a sense, the end effect is that of an empirical approach where new knowledge is induced and entered by a human analyst.

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. However, the design of these components relies largely on the local considerations which arise in parsing analyzing 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. This initial effort is probably significantly less than that invested in constructing an initial grammar by traditional methods.

As for expanding the coverage of the parser, the learning curves for Chill clearly suggest that training on larger corpora will improve Chill's parsers. Improving a parser using Chill technology requires the same investment of effort in collecting examples as traditional techniques, but automates the step of improving the parser to account for the new examples. The time and effort that would otherwise be put into debugging grammars can instead be invested in the collection and analysis of even more examples. The real promise of empirical techniques is that they allow for the construction of parsers that are consistent across a much larger range of natural language than could be achieved with hand-crafted rules. Although the principle has been demonstrated before for "artificial" problems such as word-class tagging and syntactic analysis, this is the first demonstration of the same result at the level of a complete NL application.

Chapter 8

Related Work

The work reported in this dissertation touches on numerous subfields of artificial intelligence, particularly work in machine learning and natural language processing. Much of this related research has been discussed extensively in previous chapters to motivate and explain the choices made in the development of CHILL. Although it would be impossible to discuss in detail all of the other research that has links to CHILL, certain closely related work deserves mention.

8.1 Early Research on Language Acquisition

There is a long tradition of machine learning research on the language acquisition problem. Langley and Carbonell (1985) present a nice discussion of this work and the relationship between machine learning and language acquisition research in general. Early work in this tradition concentrated on *grammar induction*, the problem of learning a recognition or generation procedure for the strings of a language (Solomonoff, 1959; Knowlton, 1962). More recent approaches to the grammar induction problem are described by Wolff (1982), VanLehn and Ball (1987) and Berwick and Pilato (1987).

The formulation of language acquisition as the problem of learning mappings from sentences into meaning representations (or vice-versa) was first investigated by Siklossy (1972) and Klein and Kuppin (1970), and has since become the "standard" paradigm for acquisition research within the AI community (Reeker, 1976; Anderson, 1977; Selfridge, 1981; Sembugamoorthy, 1981; Langley, 1982). Much of this work is similar to CHILL in that it combines inductive techniques with language-specific constraints on grammars or architectures in an effort to make the language-learning task tractable. For example, the LAS system of Anderson (1977) learned an augmented transition network (ATN) to map simple sentences into propositional meaning representations. In addition to imposing a simple ATN architecture, meaning representations were constrained by the so-called "graph deformation condition." This condition enforced a certain form of compositionality, dictating that a tree that maps meaning representations to the order in which words occurred in the original sentence could not have crossing edges. Such restrictions are similar in spirit to CHILL's shift-reduce parsing architecture and the requirements of operator transparency, derivation transparency and derivation tractability.

These early approaches differ from CHILL primarily in scope. They have employed simple propositional and, in many cases, language-specific learning algorithms; whereas, CHILL adopts a very general, first-order induction algorithm. These systems were also not systematically evaluated on realistic, large-scale corpora. Instead, the experiments had more of a "demonstration of idea" flavor. The emphasis in this dissertation on mechanisms robust enough to be of use in real natural language applications is more in line with current work under the rubric of empirical (corpus-based) NLP.

8.2 Language Acquisition as Control-Rule Learning

CHILL is not the first system to treat the problem of language acquisition as the learning of search-control heuristics. Langley (1982) and Anderson (1983) have independently posited acquisition mechanisms based on learning search-control in production systems. However, both of these systems focused on cognitive issues, and they were demonstrated on the problem of language generation rather than parser induction. Neither was demonstrated on corpora of any significant size or coverage.

The work of Berwick (1985) may be viewed as an instance of control-rule learning for parser acquisition. His system learns 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 then creating a rule using certain properties of the current parser state as trigger conditions for its application. As additional rules having the same action were created, the preconditions of the rules were generalized using a simple propositional most-specific-generalization algorithm. The motivation for this work was primarily to provide a learning mechanism to support certain approaches to generative linguistics. The system was not evaluated for generalization using any extensive corpus, but justified on the grounds that it learned a large percentage of the rules from a target grammar of a core subset of English.

In a similar vein, Simmons and Yu (1992) 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. The shift-reduce parsing framework adopted by CHILL was directly inspired by this work. However, this system depends on an analyst to provide appropriate word classifications and requires detailed interaction to guide the parsing of training examples. Although the approach was evaluated on a relatively extensive corpus, the generalization results were quite weak, and it is unclear how it compares with currently popular statistical approaches.

It should be emphasized that both of these approaches differ from CHILL in that they employ propositional representations. The context over which control rules were learned were fixed, pre-determined, and integrally connected with the particular style of syntactic analysis performed by the parsers. Neither system has the ability to create new word or phrase categories or examine more context in order to resolve difficult ambiguities. Similarly, they are limited to producing surface-oriented analyses for which their pre-defined features are most useful.

8.3 Statistical Corpus-Based NLP

Although there has been a substantial amount of research in empirical approaches to language acquisition, much of this work is not directly comparable to CHILL. 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 & Schabes, 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) is more similar to CHILL. His SPATTER 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 models 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, for example the more meaning-oriented case-role and database-query representations on which CHILL has been demonstrated.

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. While a direct comparison with CHILL is impossible without running both systems on identical corpora, there are some 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 (essentially the graph-deformation condition proposed by Anderson (1977)). This makes it awkward to handle some forms of linguistic movement. In theory, CHILL can work with any representation which meet operator transparency and tractability criteria. Finally, it should be noted that CHILL produces knowledge structures more similar to those of traditional parsers which may be advantageous in some situations.

8.4 Related ILP Work

Obviously, the ILP induction algorithm, CHILLIN, draws heavily on the insights of FOIL, GOLEM and CHAMP discussed in Chapter 4. Research in the area ILP has been expanding rapidly, and many other ILP systems have addressed issues of concern in CHILLIN.

Like CHILLIN, SERIES (Wirth & O'Rorke, 1991) and, later INDICO (Stahl, Tausend, & Wirth, 1993) make use of LGGs of examples to construct clause heads containing functions. However, both of these systems pre-compute a set of clause heads for which bodies are subsequently induced. The approach taken by CHILLIN interleaves the bottom-up and top-down mechanisms, handling a larger class of concepts.

A number of recent investigations have considered the noisy-oracle problem in the induction of recursive definitions (Cohen, 1993; Lapointe & Matwin, 1992; Muggleton, 1992). However, the proposed mechanisms either severely limit the class of learnable programs (e.g. to single clause, linearly recursive) or rely on computationally expensive matching of sub-terms, or both. None has yet been implemented and tested in a system for large-scale induction over hundreds or thousands of examples.

Predicate invention is also an area of considerable interest. Like CHILLIN and CHAMP, SERIES and INDICO employ demand-driven predicate invention. These systems differ significantly in the heuristics used to select arguments for the new predicate. Another approach to invention is the use of the intra–construction operator of inverse-resolution (Muggleton & Buntine, 1988; Wirth, 1988; Rouveirol, 1992; Banerji, 1992). In this approach, new predicates are invented through a restructuring of an existing definition, usually to make it more compact. Many of these systems require intervention in the form of an "oracle" to approve and name new concepts. Unfortunately, we are not aware of any work that has systematically evaluated the competing approaches or the practical utility of predicate invention in general.

Finally, it should be noted that Wirth (1989) has done some experiments using ILP techniques for the grammar induction problem. His system learns recognizers expressed as definite clause grammars. However, the presented results were rather preliminary, and no generalization experiments on larger-scale corpora have been reported. Given the results in Section 6.4, it is hard to imagine that purely inductive techniques can learn parsers which produce useful representations as efficiently as the control-rule learning framework of CHILL.

Chapter 9

Directions for Future Research

Obviously, there is still much room for improvement in the results reported in this dissertation, as well as interesting possibilities for further applications of the basic learning techniques.

9.1 Enhancing the Induction Algorithm

Enhancements to CHILLIN could make significant progress in several ways. Improving the accuracy of learned control rules may significantly improve the performance of the resulting parsers. Another way of achieving greater accuracy is through the use of larger training sets which necessitates improvements in the *efficiency* of the induction algorithm as well. Both of these are on-going research efforts within the ILP community.

The top-down component of CHILLIN is a very basic version of FOIL, and many of FOIL's advanced features both for extending the search horizon and for pruning fruitless search have not been implemented. Clearly, CHILLIN could make use of any of the various tweaks and optimizations which have proven useful in FOIL.

Similarly, CHILLIN in its present form does not include mechanisms for handling noisy-examples.[1] Noise in control-examples can arise from corpora that contain inconsistent annotations or that do not obey the output-completeness criterion. It might also be the case that some parsing frameworks may allow for only approximate derivation-transparency. A few incorrect parse derivations during training could introduce noise into the control-example sets even with a flawless training corpus. A simple technique for allowing learned definitions to cover a small number of negative examples was implemented in CHILLIN, but did not seem to improve the results. Incorporating more sophisticated techniques for handling noisy examples might prove a worthwhile extension of the basic induction algorithm (Quinlan, 1986, 1990; Lavrač & Džeroski, 1994).

One particular weakness of the current control-rule learning framework is that each control rule is learned in isolation from the others. It is often the case that concepts which are useful in making control decisions for one operator are also useful in making decisions for others. For example, the concept of `animate` is potentially useful in making a number of decisions during natural language parsing. The current system is forced to re-invent this concept in all the places where it might be useful. This results not only in duplicated effort, but may affect how well the resulting parser generalizes to new inputs. Different control rules for `animate` may "see" slightly different sets of examples. For instance, if the word, "dog" never appears as the agent of a sentence, then the resulting `animate` concept for the agent rule will not include "dog." If a novel sentence uses "dog" as an agent, it may not parse correctly. This is despite the fact that "dog" could have appeared as animate in another control rule such as for the patient of certain verbs like "kill." A single concept for `animate` would include all of the words that were used in either place, thus creating a more general parser. In general, pooling examples to learn a shared concept results in more accurate rules. Implementing a mechanism for the invention of such shared predicates might significantly improve the resulting parsers.

[1] A training set is said to contain *noise* if some of its examples are mis-classified.

Finally, making empirical techniques practical in real applications probably requires the development of techniques for incrementally updating learned knowledge. The induction algorithm in CHILL shares the weakness of other empirical approaches in that it uses batch processing, where grammars are acquired and refined using huge numbers of training examples. In practice, it would be preferable to have an incremental system wherein the parser could be extended and updated as more examples became available, without requiring induction from "scratch". The development of incremental algorithms for parser acquisition and maintenance seems a fruitful avenue for future research.

9.2 Corpus Engineering

The learning curves from experiments with the Geoquery corpus clearly indicate that more training data is likely to produce more accurate parsers. Enlarging the corpus would be an interesting experiment. Further data could be collected through questionnaires, but a more novel approach would be to actually collect the data from a prototype system. As new sentences were encountered, they could be stored and annotated so that CHILL could be periodically re-run to generate new parsers. One could envision making such a system available to a wide audience via an interactive demonstration over the Internet on the world-wide web. As part of the data collection project, statistics could be maintained to get a firm indication of the coverage provided by the current version of Geoquery and what extensions of the parsing framework might be most useful.

Another interesting question is whether the parsers created by CHILL could be improved by corpus "manufacturing." The case-role parsing experiments demonstrate that CHILL performs very well in situations where a corpus contains many similar sentences from which generalizations are easily drawn. "Real" corpora of the size amenable to learning in CHILL are unlikely to exhibit such high levels of regularity. Since an initial corpus must be annotated by hand, one method of increasing regularity would be to allow the annotator to introduce similar sentences. A process of introducing several paraphrases of each example sentence, or of providing sentences with similar structure but different referents might allow the induction of more robust parsers from less "real" data. Although this approach would require some extra effort from the annotator, it would be far easier than annotating an equal number of random sentences, and there is reason to hope that it would actually produce better results.

9.3 Language-Oriented Biases

Beyond improvements to the induction algorithm itself, another way to enhance the induction of parsers in CHILL is by incorporating stronger natural-language learning biases into the surrounding system. The shift-reduce framework of the current system is a rather weak bias compared to the types of restrictions which might be found in more of a "principles-and-parameters" based approach. A distinct advantage of ILP approaches is that they allow learning within a traditional NLP representation framework. This initial research has focused on how much can be accomplished by placing most of the burden on a very general learning component. A tighter integration of linguistic insights with ILP methods could probably create more efficient learning systems for language tasks.

Another avenue for improving the linguistic "savvy" of the learning system would be to provide language-oriented background knowledge as additional context for CHILLIN. Background information such as syntactic knowledge could be provided by predicates which classify words by possible lexical category. More specific lexical information could be directly inserted into the input by using a statistical tagger in a pre-processing step. Likewise, control rules might be able to use existing semantic information such as that provided by an ontological hierarchy (e.g. WordNet (Beckwith, Fellbaum, Gross, & Miller, 1991)). In principle, any relevant background knowledge should improve the performance of the system. Unfortunately, additional knowledge has the side-effect of significantly slowing the search for specializations in the top-down induction component. Making use of significant background knowledge would require modification of the search techniques used in adding literals to clauses. It is possible that providing background information in the form of tree-structured hierarchies might allow for branch-and-bound type pruning of the literal-space based on some variant of the information-gain metric used in Foil.

9.4 Soft Failure

Experiments with Chill have shown that the learned parsers tend to be quite accurate on test sentences for which parses are produced. Increasing the overall accuracy becomes a problem of providing enough training examples to get significant coverage (in terms of the percentage of test sentences which are actually parsed). If a sentence is not parsed, it may indicate that some learned control knowledge is overly-specific. A possible extension to Chill parsers might be a mechanism to search for parses which succeed by ignoring a small number of control constraints. One can conceive of a parsing process which searches for the parse that requires the violation of the fewest constraints. This would be very similar to the least-deviant-first parsing proposed by Lehman (1992) for constructing adaptive parsers.

One problem with this scheme is that there are likely to be many possible parses requiring the lifting of only a few constraints. Some general mechanism for assessing the probability of various derivations would be needed to pick the most likely parsing. Such an approach would require the collection of statistics tracking the likelihood of various derivation sequences. The result might be a marriage of the Chill approach with techniques from statistical NLP. While it is unclear exactly what form such a hybrid might take, it could potentially offer a framework utilizing the representational power of ILP while still offering the preference selecting behavior exhibited by statistical approaches.

9.5 Extending Learning to Other Problems

Finally, there are many interesting directions for extending the methods of CHILL to deal with a broader range of NLP issues. One possibility is the extension of ILP techniques to the learning of word morphology. The parsers learned by the present version of CHILL treat words as unanalyzed atomic units. Being able to recognize the similarities between words having similar roots or resulting from similar derivations might lead to better generalization. Some initial work along these lines has applied ILP to the problem of learning to form the past tense of English verbs (Mooney & Califf, 1995).

At the lexical level, automated techniques for lexicon construction could broaden the applicability of CHILL. Thompson (1995) has demonstrated an initial approach to corpus-based acquisition of lexical mapping rules suitable for use with CHILL-style parser acquisition systems. The basic idea is to find a small set of mappings from words to fragments of meaning structure such that all of the semantic structure appearing in any given training example is motivated by the words or phrases in the example sentence. Although the technique has only been used with case-role type representations, variations might also be useful for the type of lexicon required by the database-query task.

ILP techniques might also be usefully applied in learning larger discourse structures (Litman, 1994) and in information extraction tasks (Soderland & Lehnert, 1994). Larger discourse units might be described in terms of scripts in a suitable logic-oriented MRL. ILP could then be used to learn rules for script-selection and role-binding using techniques similar to those used in the database-query framework in CHILL. Given such a representation, one could also imagine a system to generate natural language outputs from scripts. A generation algorithm with relevant choices encoded as clause-selection decisions would be amenable to acquisition in a manner analogous to parser acquisition in CHILL. Combining parsing and generation components might make possible the induction of complete translation systems from a dual-language corpora annotated with a logic-based inter-lingua.

While is is difficult to envision the details of how such approaches may be implemented, it it clear that relational learning holds great promise in the domain of natural language understanding. CHILL should be viewed as a mere starting point in the investigation of the usefulness of relational learning techniques for NLP in general.

Chapter 10

Conclusion

The research presented in this dissertation may be viewed as complementing recent results in statistical NLP. The primary strength of corpus-based methods does not lie in the particular approach or type of parser employed (e.g. statistical, connectionist or symbolic), but rather with the fact that large amounts of real data may be used to automatically construct complex parsers. The experimental results presented here suggest that CHILL performs as well or better than previous approaches on comparable tasks.

The primary advantage of an approach based on control-rule learning and inductive logic programming is the resulting flexibility. CHILL may be used to learn parsers for any representation scheme that meets the criteria of operator transparency, derivation transparency and derivation tractability. The generality of these criteria has been empirically demonstrated by using CHILL to learn parsers for case-role representations, sophisticated syntactic parse-trees and logic-oriented database queries. No other empirical parser acquisition system has been evaluated on such a wide-variety of tasks or corpora. 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 feature-engineering required in propositional approaches. A further attraction of CHILL's ILP approach is the ease with which it may be integrated with traditional, symbolic parsing methods. Indeed, experimental results demonstrate that the traditional shift-reduce framework employed by CHILL is fundamental to CHILL's success in learning realistic language processing tasks.

In experiments with case-role mapping, CHILL's parsers were shown to outperform those learned by previous techniques based on artificial neural networks on identical artificial corpora. Further experiments with a much larger corpus derived from a database query task showed that CHILL could reverse-engineer a hand-constructed semantic grammar for case-role parsing to a high degree of accuracy with relatively little training data.

Experiments with learning syntactic parsers from the ATIS corpus of the Penn Treebank showed that CHILL compares favorably with state-of-the-art systems for learning unlabeled bracketings. However, CHILL is able to go beyond these previous approaches, producing completely labeled parse-trees for sophisticated syntactic representations including markers for empty constituents. CHILL is also able to learn parsers from corpora without associated lexical tags: a task requiring CHILL's ability to selectively attend to important features of structured contexts and invent new word and phrase categories.

Finally, CHILL was used to learn a parser that maps sentences directly into useful database queries without requiring intermediate syntactic representations. No other empirical approach has been demonstrated to directly learn such deep semantic representations. CHILL's parsers were integrated into a complete a natural language database interface for answering questions about U.S. geography. The resulting system was shown to outperform an existing hand-crafted program in parsing novel sentences and producing correct answers from the database. To our knowledge, this is the first demonstration of the utility of empirical techniques over hand-crafted counterparts at the level of a complete natural language application.

CHILL stands as an existence proof of the utility of modern machine learning techniques in corpus-based parser construction. It must be emphasized, however, that CHILL represents only a starting point. Statistical techniques already have a relatively long history of success in the arena of speech processing. Not surprisingly, the field of empirical parser construction has developed with a bias toward similar techniques. It is important to realize that there are alternatives to learning strategies which "simply gather statistics." Relational learning algorithms can offer significant advantages in some domains; NLP systems requiring deep semantic representations appear to be a likely candidate. Hopefully, these preliminary investigations in will stimulate further research in this direction.

Appendix A

A Generator for the M&K Corpus

Examples for the M&K corpus are generated from the `pattern/3` predciate which has the form: `pattern(PatternNumber, Sentence, Rep)`.

```
pattern(1, [the, Human, ate],
           [ate, agt:[Human, det:the]]) :-
     human(Human).

pattern(2, [the, H, ate, the, F],
           [ate, agt:[H, det:the],
                 pat:[F, det:the]] ) :-
     human(H), food(F).

pattern(3, [the, H, ate, the, F1, with, the, F2],
           [ate, agt:[H, det:the],
                 pat:[F1, det:the, accomp:[F2, prep:with, det:the]]]) :-
     human(H), food(F1), food(F2).

pattern(4, [the, H, ate, the, F, with, the, U],
           [ate, agt:[H, det:the],
                 pat:[F, det:the],
                 inst:[U, prep:with, det:the]]) :-
     human(H), food(F), utensil(U).

pattern(5, [the, A, ate],
           [ate, agt:[A, det:the]]) :-
     animal(A).

pattern(6, [the, Pred, ate, the, Prey],
           [ate, agt:[Pred, det:the],
                 pat:[Prey, det:the]]) :-
     predator(Pred), prey(Prey).

pattern(7, [the,H,broke,the,FO],
           [broke, agt:[H, det:the],
                   pat:[FO, det:the]]) :-
     human(H), fragileobj(FO).

pattern(8, [the,H,broke,the,FO,with,the,BR],
           [broke, agt:[H, det:the],
                   pat:[FO, det:the],
                   inst:[BR, prep:with, det:the]]) :-
     human(H), fragileobj(FO), breaker(BR).

pattern(9, [the,BR,broke,the,FO],
           [broke, inst:[BR, det:the],
                   pat:[FO, det:the]]):-
     fragileobj(FO), breaker(BR).

pattern(10, [the,A,broke,the,FO],
            [broke, agt:[A, det:the],
                    pat:[FO, det:the]]) :-
     animal(A), fragileobj(FO).
```

```
pattern(11, [the,FO, broke],
            [broke, pat:[FO, det:the]]) :-
     fragileobj(FO).

pattern(12, [the, H, hit, the, T],
            [hit, agt:[H, det:the],
                  pat:[T, det:the]]) :-
     human(H), thing(T).

pattern(13, [the, H1, hit, the, H2, with, the, P],
            [hit, agt:[H1, det:the],
                  pat:[H2, det:the, accomp:[P, prep:with, det:the]]]) :-
     human(H1), human(H2), posession(P).

pattern(14, [the, H, hit, the, T, with, the, Htr],
            [hit, agt:[H, det:the],
                  pat:[T, det:the],
                  inst:[Htr, prep:with, det:the]]):-
     human(H), thing(T), hitter(Htr).

pattern(15, [the, Htr, hit, the, T],
            [hit, inst:[Htr, det:the],
                  pat:[T, det:the]]):-
     hitter(Htr), thing(T).

pattern(16, [the, H, moved],
            [moved, agt:[H, det:the],
                    pat:[H, det:the]]) :-
     human(H).

pattern(17, [the, H, moved, the, O],
            [moved, agt:[H, det:the],
                    pat:[O, det:the]]) :-
     human(H), object(O).

pattern(18, [the, H, moved],
            [moved, agt:[H, det:the],
                    pat:[H, det:the]]) :-
     animal(H).

pattern(19, [the, O, moved],
            [moved, pat:[O, det:the]]) :-
     object(O).

human(H) :- member(H, [man, woman, boy, girl]).

animal(X) :- member(X, [bat, chicken, dog,
                        sheep, wolf, lion]).

predator(X) :- member(X, [wolf, lion]).

prey(X) :- member(X, [chicken, sheep]).

food(X) :- member(X, [chicken, cheese, pasta, carrot]).

utensil(X) :- member(X, [fork, spoon]).

fragileobj(X) :- member(X, [plate, window, vase]).

hitter(X) :- member(X, [bat, ball, hammer, hatchet,
                        vase, paperweight, rock]).

breaker(X) :- member(X, [bat, ball, hatchet, hammer, paperweight, rock]).
```

```
posession(X) :- member(X, [ball, bat, hatchet, hammer, vase, dog, doll]).

object(X) :- member(X,[bat,ball,hatchet, hammer,vase,plate,window,
                       fork, spoon, pasta, cheese, chicken, carrot,
                       desk, doll, curtain, paperweight,rock]).

thing(X) :- member(X, [man,woman,boy,girl,
                       bat,chicken,dog,sheep,wolf,lion,
                       ball,hatchet,hammer,vase,plate,window,
                       fork, spoon, pasta, cheese, carrot,
                       desk, doll, curtain, paperweight,rock]).
```

Appendix B

Example CHILL Output

The following set of specialized operators was produced from a run using 550 training examples from the M&K corpus.

```
%-------------------------------------------------------------------
op([A,[B,det:the]],C,[D],C) :-
     pred788(B), pred790(B,A), reduce(A,agt,[B,det:the],D).

pred788(bat).  pred788(boy).   pred788(dog).   pred788(girl).
pred788(lion). pred788(man).   pred788(sheep). pred788(woman).

pred790(A,broke).   pred790(bat,[moved,obj:[bat,det:the]]).
pred790(boy,moved). pred790(bat,ate).

%-------------------------------------------------------------------
op([A,[B,det:the]],[the,C],[D],[the,C]) :-
     pred793(B), reduce(A,inst,[B,det:the],D).

pred793(ball).    pred793(bat).  pred793(paperweight).
pred793(hatchet). pred793(rock). pred793(vase). pred793(hammer).

%-------------------------------------------------------------------
```

```
op([A,[B,det:the]],[],[C],[]) :-
     pred799(B), reduce(A,obj,[B,det:the],C).

pred799(ball). pred799(doll).  pred799(hatchet). pred799(plate).
pred799(rock). pred799(spoon). pred799(vase).    pred799(window).

%----------------------------------------------------------------------
op([A,the|B],C,[D|B],C) :- reduce(A,det,the,D).

%----------------------------------------------------------------------
op([[A,B:C|D],[E,F:[G,det:the]]],H,[I],H) :-
     pred801(H), reduce([E,F:[G,det:the]],obj,[A,B:C|D],I).

pred801([]).
pred801([with,the,A]) :- pred802(A).

pred802(ball).    pred802(bat).  pred802(fork).  pred802(hammer).
pred802(hatchet). pred802(rock). pred802(spoon). pred802(vase).
pred802(paperweight).

%----------------------------------------------------------------------
op([[A,det:the],B,[C,D:E|F]|G],[],[H|G],[]) :-
     reduce([C,D:E|F],B,[A,det:the],H).

%----------------------------------------------------------------------
op([moved,[A,det:the]],[],[B,[A,det:the]],[]) :-
     pred809(A), reduce(moved,obj,[A,det:the],B).

pred809(dog). pred809(sheep).

%----------------------------------------------------------------------
op([[A,det:the],[B,agt:[C,det:the]]],[with,the,D],[accomp,[A,det:the],
   [B,agt:[C,det:the]]],[the,D]) :- pred861(A,D).

pred861(A,B) :- pred862(B), pred873(A).

pred862(ball).   pred862(bat).   pred862(carrot). pred862(vase).
pred862(cheese). pred862(pasta). pred862(dog).    pred862(doll).
pred862(hammer). pred862(hatchet). pred862(chicken).

pred873(boy). pred873(girl). pred873(man). pred873(woman).

%-------------------------------------------------------------------
op([[A,obj:[B,det:the],agt:[C,det:the]]],[with,the,D],
   [inst,[A,obj:[B,det:the],agt:[C,det:the]]],[the,D]).

%-------------------------------------------------------------------
op(A,[B|C],[B|A],C) :- pred897(A).

pred897([A|B]) :- pred899(A).
pred897([]).
pred897([[A,B:C]]).

pred899(accomp). pred899(inst). pred899(the).
```

Appendix C

Example Trace of CHILLIN

What follows is two traces of output from CHILLIN learning a definition of the concept `uncle/2`. The first trace shows how the definition is learned when provided with the necesary background knowledge: `wed/2`, `sibling/2`, `male/1` and `female/1`. In the second trace, `male/1` is not provided and must be invented. The traces have been augmented with comments (bracketed with "**") to help explain what is happening.

```
** Begin trace with all background predicates  **


| ?- top(uncle).
** First compile the background predicates **
% compiling file /tmp_mnt/v/sally/v6/zelle/ilp/induce/uncle.i
% uncle.i compiled in module thetheory, 1.416 sec 14,756 bytes

Inducing concept: uncle/2 from 50 positives and 100 negatives

** Initial compaction via LGGs of examples, no consistent gen found **
** so the initial definition is just the 50 examples               **
Unit Clause Count: 50
Unit Definition:
```

```
uncle(art,m11).
uncle(calvin,f23).
uncle(calvin,f26).
uncle(calvin,f28).
uncle(calvin,m24).
uncle(calvin,m27).
uncle(carlos,f23).
uncle(carlos,f25).
uncle(carlos,m24).
uncle(david,art).
uncle(david,umo).
uncle(david,wendy).
uncle(eric,frederick).
uncle(eric,jonas).
uncle(eric,melvin).
uncle(eric,prissie).
uncle(fred,frederick).
uncle(fred,jane).
uncle(fred,prissie).
uncle(fred,umo).
uncle(frederick,m24).
uncle(george,art).
uncle(george,cornelia).
uncle(george,frederick).
uncle(george,nancy).
uncle(george,wendy).
uncle(harry,melvin).
uncle(harry,umo).
uncle(jack,f26).
uncle(jack,f28).
uncle(jack,m27).
uncle(james,angela).
uncle(karl,rachel).

uncle(karl,susan).

uncle(leon,janet).

uncle(leon,nero).

uncle(leon,susan).

uncle(mark,janet).

uncle(mark,paul).

uncle(melvin,f20).

uncle(neil,m17).

uncle(nero,christy).

uncle(paul,f2).

uncle(paul,m1).

uncle(peter,f23).

uncle(umo,f14).

uncle(umo,m11).

uncle(umo,m13).

uncle(walt,f14).

uncle(walt,m13).


New Compaction loop

Current Size: 350


Abstract Foil

FOIL Initial Clauses:   ** All pair Lggs sampled gave same result **

uncle(A,B).


** This single clause goes on the heap **


** Each group here shows removing a single clause and searching for **

** the best single literal extension.                               **


Current Clause: uncle(_78383,_78384):-true
```

```
Trying Predicate: male/1
Trying Predicate: female/1
Trying Predicate: sibling/2
Trying Predicate: wed/2
Found Better: wed(_83573,_78383)   Gain = 13.44
Trying Predicate: parent/2

Current Clause: uncle(_78383,_78384):-wed(_83573,_78383)
Trying Predicate: male/1
Trying Predicate: female/1
Trying Predicate: sibling/2
Found Better: sibling(_89289,_83573)   Gain = 9.94
Trying Predicate: wed/2
Trying Predicate: parent/2

Current Clause: uncle(_78383,_78384):-wed(_83573,_78383),
                                      sibling(_89289,_83573)
Trying Predicate: male/1
Trying Predicate: female/1
Found Better: female(_78384)   Gain = 3.16
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2
Found Better: parent(_89289,_78384)   Gain = 18.67

** Adding the parent literal results in a consistent clause.      **
** There are no negative examples forcing A to be male.           **

Foil Done: [0]      ** Bracketed number is count of partial clauses **
 Clause:
uncle(A,B) :-
        wed(C,A),
        sibling(D,C),
        parent(D,B).
 Partials:          ** These are the clauses which were given up on as  **
                    ** unextendable. (There aren´t any in this case)    **

** The FOIL component found a consistent clause, it is returned as the **
** only modification to try                                            **

Trying 1 Mods
uncle(A,B) :-
        wed(C,A),
        sibling(D,C),
        parent(D,B).

** This clause does compress the definition, so it is implemented. **
** Roughly half of the unit clauses are eliminated by this mod.    **

Modification Implemented
 Size: 220

New Compaction loop
Current Size: 220
Abstract Foil
FOIL Initial Clauses:  ** Same single seed clause as last time **
uncle(A,B).

Current Clause: uncle(_117732,_117733):-true
Trying Predicate: male/1
Trying Predicate: female/1
Trying Predicate: sibling/2
Found Better: sibling(_119618,_117732)   Gain = 10.83
Trying Predicate: wed/2
Trying Predicate: parent/2

** wed/2 does not give gain this time, since it covers mostly xs  **
** that were handled by the first clause.  The gain metric gives  **
** preference to covering lots of _differing_ clauses.            **

Current Clause: uncle(_117732,_117733):-sibling(_119618,_117732)
Trying Predicate: male/1
Found Better: male(_119618)   Gain = 2.35
Trying Predicate: female/1
Found Better: female(_119618)   Gain = 7.03
Trying Predicate: sibling/2
Trying Predicate: wed/2
Found Better: wed(_129303,_119618)   Gain = 20.80
Trying Predicate: parent/2
Found Better: parent(_119618,_117733)   Gain = 46.40

Current Clause: uncle(_117732,_117733):-sibling(_119618,_117732),
                                        parent(_119618,_117733)
Trying Predicate: male/1
Found Better: male(_117732)   Gain = 4.12

** Search is aborted ar this point, as a "perfect" literal was found **

Foil Done: [0]
 Clause:
uncle(A,B) :-
        sibling(C,A),
        parent(C,B),
        male(A).
 Partials:
```

```
** The second clause has been found, and is implemented.  **

Trying 1 Mods
uncle(A,B) :-
        sibling(C,A),
        parent(C,B),
        male(A).



Modification Implemented
 Size: 33



New Compaction loop
Current Size: 33
Abstract Foil
FOIL Initial Clauses:    ** This is the clause LGG of the **
uncle(A,B) :-            ** current definition.           **
        parent(C,B).



Current Clause: uncle(_139239,_139240):-parent(_139242,_139240)
Trying Predicate: male/1
Trying Predicate: female/1
Trying Predicate: sibling/2
Found Better: sibling(_139239,_139242)   Gain = 3.67
Trying Predicate: wed/2
Trying Predicate: parent/2

Current Clause: uncle(_139239,_139240):-parent(_139242,_139240),
                                        sibling(_139239,_139242)
Trying Predicate: male/1
```

```
Found Better: male(_139239)   Gain = 2.00
Foil Done: [0]           ** An equivalent clause is found. **
 Clause:
uncle(A,B) :-
        parent(C,B),
        sibling(A,C),
        male(A).
 Partials:



Trying 1 Mods            ** no compaction (since it´s already there) **
uncle(A,B) :-
        parent(C,B),
        sibling(A,C),
        male(A).



Induction Time: 12.483000  ** CPU seconds  on a SPARC 2 **

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

%-----------------------------------------------------------------------
** Background edited to remove male/1 **

| ?- top(uncle).
```

```
% compiling file /tmp_mnt/v/sally/v6/zelle/ilp/induce/uncle.i
% uncle.i compiled in module thetheory, 1.417 sec 14,036 bytes

Inducing concept: uncle/2 from 50 positives and 100 negatives


Unit Clause Count: 50
Unit Definition:
uncle(art,m11).
uncle(calvin,f23).
uncle(calvin,f26).
uncle(calvin,f28).
uncle(calvin,m24).
uncle(calvin,m27).
uncle(carlos,f23).
uncle(carlos,f25).
uncle(carlos,m24).
uncle(david,art).
uncle(david,umo).
uncle(david,wendy).
uncle(eric,frederick).
uncle(eric,jonas).
uncle(eric,melvin).
uncle(eric,prissie).
uncle(fred,frederick).
uncle(fred,jane).
uncle(fred,prissie).
uncle(fred,umo).
uncle(frederick,m24).
uncle(george,art).
uncle(george,cornelia).
uncle(george,frederick).
uncle(george,nancy).
uncle(george,wendy).
uncle(harry,melvin).
uncle(harry,umo).
uncle(jack,f26).
uncle(jack,f28).
uncle(jack,m27).
uncle(james,angela).
uncle(karl,rachel).
uncle(karl,susan).
uncle(leon,janet).
uncle(leon,nero).
uncle(leon,susan).
uncle(mark,janet).
uncle(mark,paul).
uncle(melvin,f20).
uncle(neil,m17).
uncle(nero,christy).
uncle(paul,f2).
uncle(paul,m1).
uncle(peter,f23).
uncle(umo,f14).
uncle(umo,m11).
uncle(umo,m13).
uncle(walt,f14).
uncle(walt,m13).

New Compaction loop
Current Size: 350
Abstract Foil
FOIL Initial Clauses:
uncle(A,B).
uncle(A,art).
```

```
Current Clause: uncle(_94768,_94769):-true

Trying Predicate: female/1

Trying Predicate: sibling/2

Trying Predicate: wed/2

Found Better: wed(_99231,_94768)   Gain = 13.44

Trying Predicate: parent/2


Current Clause: uncle(_94768,_94769):-wed(_99231,_94768)

Trying Predicate: female/1

Trying Predicate: sibling/2

Found Better: sibling(_104290,_99231)   Gain = 9.94

Trying Predicate: wed/2

Trying Predicate: parent/2


Current Clause: uncle(_94768,_94769):-wed(_99231,_94768),

                                      sibling(_104290,_99231)

Trying Predicate: female/1

Found Better: female(_94769)   Gain = 3.16

Trying Predicate: sibling/2

Trying Predicate: wed/2

Trying Predicate: parent/2

Found Better: parent(_104290,_94769)   Gain = 18.67


** First Clause is learned exactly as in the previous trace.  **


Foil Done: [0]

 Clause:

uncle(A,B) :-

        wed(C,A),

        sibling(D,C),

        parent(D,B).
 Partials:


Trying 1 Mods
uncle(A,B) :-
        wed(C,A),
        sibling(D,C),
        parent(D,B).

Modification Implemented
 Size: 220

New Compaction loop
Current Size: 220
Abstract Foil
FOIL Initial Clauses:
uncle(A,B).
uncle(eric,A).

Current Clause: uncle(_132128,_132129):-true
Trying Predicate: female/1
Trying Predicate: sibling/2
Found Better: sibling(_133367,_132128)   Gain = 10.83
Trying Predicate: wed/2
Trying Predicate: parent/2

Current Clause: uncle(_132128,_132129):-sibling(_133367,_132128)
Trying Predicate: female/1
Found Better: female(_133367)   Gain = 7.03
Trying Predicate: sibling/2
Trying Predicate: wed/2
Found Better: wed(_142261,_133367)   Gain = 20.80
```

```
Trying Predicate: parent/2
Found Better: parent(_133367,_132129)   Gain = 46.40

Current Clause: uncle(_132128,_132129):-sibling(_133367,_132128),
                                        parent(_133367,_132129)
Trying Predicate: female/1
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2

** The developing clause: uncle :- sibling, parent, was not extendable **
** However, the next clause retrieved from the heap was consistent.    **

Foil Done: [1]
 Clause:
uncle(eric,A).
 Partials:            ** Ignored, because a complete clause found **
uncle(A,B) :-
        sibling(C,A),
        parent(C,B).



Trying 1 Mods
uncle(eric,A).

** It does give a bit of compaction, but a great number of **
** unit clauses must still be in this definition.          **

Modification Implemented
 Size: 205

New Compaction loop
```

```
Current Size: 205
Abstract Foil
FOIL Initial Clauses:
uncle(A,B).

Current Clause: uncle(_160033,_160034):-true
Trying Predicate: female/1
Trying Predicate: sibling/2
Found Better: sibling(_161221,_160033)   Gain = 10.20
Trying Predicate: wed/2
Trying Predicate: parent/2

Current Clause: uncle(_160033,_160034):-sibling(_161221,_160033)
Trying Predicate: female/1
Found Better: female(_161221)   Gain = 6.69
Trying Predicate: sibling/2
Trying Predicate: wed/2
Found Better: wed(_171009,_161221)   Gain = 19.90
Trying Predicate: parent/2
Found Better: parent(_161221,_160034)   Gain = 43.09

Current Clause: uncle(_160033,_160034):-sibling(_161221,_160033),
                                        parent(_161221,_160034)
Trying Predicate: female/1
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2

** The FOIL component comes up dry, so CHILLIN turns to the  **
** partial clauses to see if invention can help.             **

Foil Done: [1]
```

```
 Clause:
empty.
 Partials:
uncle(A,B) :-
        sibling(C,A),
        parent(C,B).

 Inventing for:
uncle(A,B) :-
        sibling(C,A),
        parent(C,B).

** The trace does not show the loop for selecting variables. **
** A alone is chosen.                                         **

New Predicate Arity: 1
New Predicate Arity OK  ** Only limited arity preds are pursued **

** Now the inductive component is called recursively. **

Inducing concept: ipred1/1 from 13 positives and 3 negatives

Unit Clause Count: 13
Unit Definition:
ipred1(art).
ipred1(calvin).
ipred1(david).
ipred1(eric).
ipred1(frederick).
ipred1(harry).
ipred1(jack).
ipred1(karl).
```

```
ipred1(mark).
ipred1(melvin).
ipred1(nero).
ipred1(paul).
ipred1(umo).

New Compaction loop
Current Size: 65
Abstract Foil
FOIL Initial Clauses:
ipred1(A).

Current Clause: ipred1(_188051):-true
Trying Predicate: female/1
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2
Foil Done: [1]
 Clause:
empty.
 Partials:
ipred1(A).

** Obviously, this invention is vacuous, but this is dicovered **
** within the invention routines, so we get this trace info.   **
 Inventing for:
ipred1(A).

New Predicate Arity: 1
New Predicate Arity OK

** Discovered cirularity of this invention and failed out without **
** proposing any potential generalization. The positive examples  **
** are memorized.                                                 **

Trying 0 Mods

Successfully Invented: ipred1(_160033)

** The newly invented predicate completes the partial clause. **
Trying 1 Mods
uncle(A,B) :-
        sibling(C,A),
        parent(C,B),
        ipred1(A).

Modification Implemented
 Size: 104

New Compaction loop
Current Size: 104
Abstract Foil
FOIL Initial Clauses:
uncle(A,B).
uncle(A,B) :-
        parent(C,B).



Current Clause: uncle(_197230,_197231):-true
Trying Predicate: female/1
Trying Predicate: ipred1/1
Found Better: ipred1(_197230)   Gain = 2.49
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2

** Now the other clause looks better, it is extended. **

Current Clause: uncle(_197204,_197205):-parent(_197207,_197205)
Trying Predicate: female/1
Trying Predicate: ipred1/1
Found Better: ipred1(_197204)   Gain = 2.49
Trying Predicate: sibling/2
Found Better: sibling(_197204,_197207)   Gain = 3.10
Trying Predicate: wed/2
Trying Predicate: parent/2

** And back to the first clause... **

Current Clause: uncle(_197230,_197231):-ipred1(_197230)
Trying Predicate: female/1
Trying Predicate: ipred1/1
Trying Predicate: sibling/2
Trying Predicate: wed/2
Trying Predicate: parent/2

** Couldn´t extend it, back to the other one. **

Current Clause: uncle(_197204,_197205):-parent(_197207,_197205),
                                        sibling(_197204,_197207)
Trying Predicate: female/1
Trying Predicate: ipred1/1
Found Better: ipred1(_197204)   Gain = 2.00

** Just as in the first trace, a previously found clause re-discovered **
Foil Done: [1]
```

```
 Clause:
uncle(A,B) :-
        parent(C,B),
        sibling(A,C),
        ipred1(A).
 Partials:
uncle(A,B) :-
        ipred1(A).



Trying 1 Mods
uncle(A,B) :-
        parent(C,B),
        sibling(A,C),
        ipred1(A).

** No compaction **

** Post Pruning of the clause set eliminates the uneeded clause.   **
** This is done by dropping each clause in turn and seeing if all  **
** the examples can still be proved.  The trace doesn´t show this. **

Induction Time: 18.100000 ** CPU seconds on SPARC 2 **

ipred1(art).
ipred1(calvin).
ipred1(david).
ipred1(eric).
ipred1(frederick).
ipred1(harry).
ipred1(jack).
ipred1(karl).
```

```
ipred1(mark).
ipred1(melvin).
ipred1(nero).
ipred1(paul).
ipred1(umo).
uncle(A,B) :-
        wed(C,A),
        sibling(D,C),
        parent(D,B).
uncle(A,B) :-
        sibling(C,A),
        parent(C,B),
        ipred1(A).

yes
| ?-
```

Appendix D

Geoquery Corpus Questionnaire

This questionnaire was distributed to undergraduates to generate questions for the Geoquery corpus discussed in Chapter 7.

Asking a Computer about US Geography

You are invited to participate in generating data for a research project involving human-computer interaction. We wish to construct a list of questions that people might ask of a computer system that knows about US geography. The system is called Geobase and contains the following information:

Information about states:

- Area of the state in square kilometers
- Population of the state in citizens
- Capital of the state
- Which states border a given state
- Major rivers in the state
- Major cities in the state

- Highest and lowest point in the state in meters

Information about rivers:

- Length of river in kilometers

Information about cities:

- Population of the city in citizens

This information is accessible by asking questions in normal English. Here are some sample inquiries:

Give me the cities in California.

What is the biggest city in California?

What is the longest river in the USA?

Which rivers are longer than 1000 kilometers?

What is the name of the state with the lowest point?

Which states border Alabama?

Which rivers do not run through Texas?

Which rivers run through states that border the state with the capital Austin?

On the following pages, please write down 15 questions that you would expect this system to be able to answer. If you can't think of 15, just write down as many as you can.

Appendix E

The Geoquery Corpus

The 250 sentences and associated queries of the Geoquery corpus are listed here. The sentences are printed out as unquoted Prolog literals. Thus, there is no capitalization, and final punctuation is separated from the last word of the sentence.

---

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

    answer(C, (capital(B,C),largest(A,(state(B),population(B,A))))).

what are the major cities in kansas ?

    answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(kansas)))).

what is the population of the major cities in wisconsin ?

    answer(C, (population(B,C),major(B),city(B),loc(B,A),equal(A,stateid(wisconsin)))).

what is the combined area of all 50 states ?

    answer(C, sum(B,(area(A,B),state(A)),C)).

what is the capital of the state with the highest point ?

    answer(C, (capital(B,C),highest(A,(state(B),high_point(B,A))))).

what states border ohio ?

    answer(B, (state(B),next_to(B,A),equal(A,stateid(ohio)))).

what is the highest point of the state with the largest area ?

    answer(C, (high_point(B,C),largest(A,(state(B),area(B,A))))).

what is the lowest point of the state with the largest area ?

    answer(C, (low_point(B,C),largest(A,(state(B),area(B,A))))).

what is the combined population of all 50 states ?

answer(C, sum(B,(population(A,B),state(A)),C)).

what is the population density of texas ?

answer(B, (density(A,B),equal(A,stateid(texas)))).

how many people live in california ?

answer(B, (population(A,B),equal(A,stateid(california)))).

how many people live in new york ?

answer(B, (population(A,B),equal(A,stateid('new york')))).

how long is the rio grande river ?

answer(B, (len(A,B),equal(A,riverid('rio grande')))).

how many states does the colorado river run through ?

answer(C, count(B,(state(B),equal(A,riverid(colorado)),traverse(A,B)),C)).

how many major cities are in florida ?

answer(C, count(B,(major(B),city(B),loc(B,A),equal(A,stateid(florida))),C)).

what is the biggest city in texas ?

answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(texas))))).

which states border colorado ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(colorado)))).

what is the lowest point in the state of texas ?

answer(B, (low_point(A,B),state(A),equal(A,stateid(texas)))).

what is the lowest point in the state of california ?

answer(B, (low_point(A,B),state(A),equal(A,stateid(california)))).

what is the longest river in the united states ?

answer(A, longest(A,river(A))).

what is the population of arizona ?

answer(B, (population(A,B),equal(A,stateid(arizona)))).

what is the population of idaho ?

answer(B, (population(A,B),equal(A,stateid(idaho)))).

what are the major cities in ohio ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(ohio)))).

which states border new york ?

answer(B, (state(B),next_to(B,A),equal(A,stateid('new york')))).

what is the capital of maine ?

answer(B, (capital(A,B),equal(A,stateid(maine)))).

which states border kentucky ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(kentucky)))).

what rivers run through the states that border the state with the capital atlanta ?

answer(D, (river(D),traverse(D,C),state(C),next_to(C,B),state(B),capital(B,A), equal(A,cityid(atlanta,_)))).

what are the major cities in california ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(california)))).

how many people live in kalamazoo ?

answer(B, (population(A,B),equal(A,cityid(kalamazoo,_)))).

which state is kalamazoo in ?

answer(B, (state(B),equal(A,cityid(kalamazoo,_)),loc(A,B))).

what states have cities named dallas ?

answer(B, (state(B),loc(A,B),city(A),equal(A,cityid(dallas,_)))).

what state is des moines located in ?

answer(B, (state(B),equal(A,cityid('des moines',_)),loc(A,B))).

how long is the shortest river in the usa ?

answer(B, (len(A,B),shortest(A,river(A)))).

what are the rivers in alaska ?

answer(B, (river(B),loc(B,A),equal(A,stateid(alaska)))).

how large is the largest city in alaska ?

answer(C, (size(B,C),largest(B,(city(B),loc(B,A),equal(A,stateid(alaska)))))).

what states border florida ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(florida)))).

what is the smallest state in the usa ?

answer(A, smallest(A,state(A))).

what states have cities named plano ?

answer(B, (state(B),loc(A,B),city(A),equal(A,cityid(plano,_)))).

how many people live in the capital of texas ?

answer(C, (population(B,C),capital(A,B),equal(A,stateid(texas)))).

how many rivers does colorado have ?

answer(C, count(B,(river(B),equal(A,stateid(colorado)),loc(B,A)),C)).

how many rivers does alaska have ?

answer(C, count(B,(river(B),equal(A,stateid(alaska)),loc(B,A)),C)).

what is the biggest city in the usa ?

answer(A, largest(A,city(A))).

what is the population density of the smallest state ?

answer(B, (density(A,B),smallest(A,state(A)))).

what is the total population of the states that border texas ?

answer(D, sum(C,(population(B,C),state(B),next_to(A,B),equal(A,stateid(texas))),D)).

what rivers run through colorado ?

answer(B, (river(B),traverse(B,A),equal(A,stateid(colorado)))).

what is the largest city in states that border california ?

answer(C, largest(C,(city(C),loc(C,B),state(B),next_to(B,A),equal(A,stateid(california))))).

what is the population of illinois ?

answer(B, (population(A,B),equal(A,stateid(illinois)))).

what rivers do not run through tennessee ?

answer(B, (river(B), not(traverse(B,A),equal(A,stateid(tennessee))))).

what is the biggest city in louisiana ?

answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(louisiana))))).

what states border indiana ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(indiana)))).

what is the population of boston massachusetts ?

answer(B, (population(A,B),equal(A,cityid(boston,ma)))).

what is the longest river in mississippi ?

answer(B, longest(B,(river(B),loc(B,A),equal(A,stateid(mississippi))))).

how many citizens in alabama ?

answer(B, (population(A,B),equal(A,stateid(alabama)))).

what is the area of maine ?

answer(B, (area(A,B),equal(A,stateid(maine)))).

how many rivers in washington ?

answer(C, count(B,(river(B),loc(B,A),equal(A,stateid(washington))),C)).

what is the largest city in wisconsin ?

answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(wisconsin))))).

what is the capital of georgia ?

answer(B, (capital(A,B),equal(A,stateid(georgia)))).

rivers in new york ?

answer(B, (river(B),loc(B,A),equal(A,stateid('new york')))).

what states border rhode island ?

answer(B, (state(B),next_to(B,A),equal(A,stateid('rhode island')))).

what is the population of montana ?

answer(B, (population(A,B),equal(A,stateid(montana)))).

what is the total area of the usa ?

answer(B, (area(A,B),equal(A,countryid(usa)))).

where is the lowest spot in iowa ?

answer(B, (low_point(A,B),equal(A,stateid(iowa)))).

how long is the north platte river ?

answer(B, (len(A,B),equal(A,riverid('north platte')))).

what is the highest point of the usa ?

answer(B, (high_point(A,B),equal(A,countryid(usa)))).

what states border new jersey ?

answer(B, (state(B),next_to(B,A),equal(A,stateid('new jersey')))).

what is the longest river ?

answer(A, longest(A,river(A))).

what is the highest point in the state with the capital des moines ?

answer(C, (high_point(B,C),loc(C,B),capital(B,A),equal(A,cityid('des moines',_)))).

what is the most populated state bordering oklahoma ?

answer(C, largest(B,(population(C,B),state(C),next_to(C,A),equal(A,stateid(oklahoma))))).

which state has the smallest population density ?

answer(B, smallest(A,(state(B),density(B,A)))).

what state has the largest population density ?

answer(B, largest(A,(state(B),density(B,A)))).

what states capital is dover ?

answer(B, (state(B),capital(B,A),equal(A,cityid(dover,_)))).

what capital is the largest in the us ?

answer(A, largest(A,capital(A))).

how large is alaska ?

answer(B, (size(A,B),equal(A,stateid(alaska)))).

how many people live in hawaii ?

answer(B, (population(A,B),equal(A,stateid(hawaii)))).

where is the lowest point in the us ?

answer(B, (low_point(A,B),equal(A,countryid(usa)))).

how many cities are in montana ?

answer(C, count(B,(city(B),loc(B,A),equal(A,stateid(montana))),C)).

which states have points higher than the highest point in colorado ?

answer(D, (state(D),high_point(D,C),higher(C,B),high_point(A,B),equal(A,stateid(colorado)))).

how many people live in rhode island ?

answer(B, (population(A,B),equal(A,stateid('rhode island')))).

what city has the most people ?

answer(B, largest(A,(city(B),population(B,A)))).

what is the population of springfield missouri ?

answer(B, (population(A,B),equal(A,cityid(springfield,mo)))).

what is the length of the colorado river ?

answer(B, (len(A,B),equal(A,riverid(colorado)))).

what states does the missouri run through ?

answer(B, (state(B),equal(A,riverid(missouri)),traverse(A,B))).

what are the major cities in wyoming ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(wyoming)))).

what is the lowest point in oregon ?

answer(B, (low_point(A,B),equal(A,stateid(oregon)))).

what is the area of alaska ?

answer(B, (area(A,B),equal(A,stateid(alaska)))).

what is the population of texas ?

answer(B, (population(A,B),equal(A,stateid(texas)))).

what is the population of san antonio ?

answer(B, (population(A,B),equal(A,cityid('san antonio',_)))).

what is the highest point in kansas ?

answer(B, (high_point(A,B),equal(A,stateid(kansas)))).

what is the longest river in the us ?

answer(A, longest(A,river(A))).

what length is the mississippi ?

answer(B, (len(A,B),equal(A,riverid(mississippi)))).

what states border hawaii ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(hawaii)))).

what is the lowest point in louisiana ?

answer(B, (low_point(A,B),equal(A,stateid(louisiana)))).

how high is the highest point in america ?

answer(C, (elevation(B,C),high_point(A,B),equal(A,countryid(usa)))).

what is the smallest city in hawaii ?

answer(B, smallest(B,(city(B),loc(B,A),equal(A,stateid(hawaii))))).

how many people live in minneapolis minnesota ¿

answer(B, (population(A,B),equal(A,cityid(minneapolis,mn)))).

how many people live in austin ?

answer(B, (population(A,B),equal(A,cityid(austin,_)))).

how long is the ohio river ?

answer(B, (len(A,B),equal(A,riverid(ohio)))).

give me the largest state ?

answer(A, largest(A,state(A))).

what is the lowest point of the us ?

answer(B, (low_point(A,B),equal(A,countryid(usa)))).

what is the population density of maine ?

answer(B, (density(A,B),equal(A,stateid(maine)))).

what is the population of new york city ?

answer(B, (population(A,B),equal(A,cityid('new york',_)))).

what is the capital of new hampshire ?

answer(B, (capital(A,B),equal(A,stateid('new hampshire')))).

what is the population of south dakota ?

answer(B, (population(A,B),equal(A,stateid('south dakota')))).

which states border south dakota ?

answer(B, (state(B),next_to(B,A),equal(A,stateid('south dakota')))).

which states have cities named austin ?

answer(B, (state(B),loc(A,B),city(A),equal(A,cityid(austin,_)))).

how high is the highest point in montana ?

answer(C, (elevation(B,C),high_point(A,B),equal(A,stateid(montana)))).

which state is the city denver located in ?

answer(B, (state(B),equal(A,cityid(denver,_)),loc(A,B))).

where is the highest point in montana ?

answer(B, (high_point(A,B),equal(A,stateid(montana)))).

what is the lowest point in the united states ?

answer(B, (low_point(A,B),equal(A,countryid(usa)))).

which state has the longest river ?

answer(B, longest(A,(state(B),loc(A,B),river(A)))).

which state has the largest city ?

answer(B, largest(A,(state(B),loc(A,B),city(A)))).

what state has the greatest population density ?

answer(B, largest(A,(state(B),density(B,A)))).

what is the lowest point in texas ?

answer(B, (low_point(A,B),equal(A,stateid(texas)))).

how many people live in riverside ?

answer(B, (population(A,B),equal(A,cityid(riverside,_)))).

what states border delaware ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(delaware)))).

what states does the ohio river go through ?

answer(B, (state(B),equal(A,riverid(ohio)),traverse(A,B))).

how long is the delaware river ?

answer(B, (len(A,B),equal(A,riverid(delaware)))).

what state has the city flint ?

answer(B, (state(B),loc(A,B),city(A),equal(A,cityid(flint,_)))).

what is the shortest river ?

answer(A, shortest(A,river(A))).

which states capital city is the largest ?

answer(B, largest(A,(state(B),capital(B,A)))).

which state is the smallest ?

answer(A, smallest(A,state(A))).

what states does the delaware river run through ?

answer(B, (state(B),equal(A,riverid(delaware)),traverse(A,B))).

what is the population of hawaii ?

answer(B, (population(A,B),equal(A,stateid(hawaii)))).

what are the major cities in alaska ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(alaska)))).

what is the highest point in the country ?

answer(B, (high_point(A,B),equal(A,countryid(usa)))).

what is the longest river in florida ?

answer(B, longest(B,(river(B),loc(B,A),equal(A,stateid(florida))))).

what is the largest state capital in population ?

answer(B, largest(A,(capital(B),population(B,A)))).

how long is the missouri river ?

answer(B, (len(A,B),equal(A,riverid(missouri)))).

what state contains the highest point in the us ?

answer(C, (state(C),loc(B,C),high_point(A,B),equal(A,countryid(usa)))).

how large is texas ?

answer(B, (size(A,B),equal(A,stateid(texas)))).

what is the population of erie pennsylvania ?

answer(B, (population(A,B),equal(A,cityid(erie,pa)))).

how long is the colorado river ?

answer(B, (len(A,B),equal(A,riverid(colorado)))).

which states does the mississippi run through ?

answer(B, (state(B),equal(A,riverid(mississippi)),traverse(A,B))).

how many rivers are there in idaho ?

answer(C, count(B,(river(B),loc(B,A),equal(A,stateid(idaho))),C)).

what is the population of tempe arizona ?

answer(B, (population(A,B),equal(A,cityid(tempe,az)))).

what is the capital of iowa ?

answer(B, (capital(A,B),equal(A,stateid(iowa)))).

what is the lowest point in california ?

answer(B, (low_point(A,B),equal(A,stateid(california)))).

which state borders florida ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(florida)))).

which state borders hawaii ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(hawaii)))).

how many rivers are in new york ?

answer(C, count(B,(river(B),loc(B,A),equal(A,stateid('new york'))),C)).

what state borders michigan ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(michigan)))).

what is the length of the mississippi river ?

answer(B, (len(A,B),equal(A,riverid(mississippi)))).

what is the largest river in washington state ?

answer(B, largest(B,(river(B),loc(B,A),equal(A,stateid(washington))))).

what is the population of seattle washington ?

answer(B, (population(A,B),equal(A,cityid(seattle,wa)))).

what are the major cities of texas ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(texas)))).

```
what is the area of south carolina ?
    answer(B, (area(A,B),equal(A,stateid('south carolina')))).
where is the highest point in hawaii ?
    answer(B, (high_point(A,B),equal(A,stateid(hawaii)))).
what is the lowest point in arkansas ?
    answer(B, (low_point(A,B),equal(A,stateid(arkansas)))).
what is the capital of utah ?
    answer(B, (capital(A,B),equal(A,stateid(utah)))).
what states surround kentucky ?
    answer(B, (state(B),next_to(B,A),equal(A,stateid(kentucky)))).
what is the biggest city in wyoming ?
    answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(wyoming))))).
how long is the mississippi river ?
    answer(B, (len(A,B),equal(A,riverid(mississippi)))).
what are the major rivers in ohio ?
    answer(B, (major(B),river(B),loc(B,A),equal(A,stateid(ohio)))).
what is the population of the capital of the smallest state ?
    answer(C, (population(B,C),capital(A,B),smallest(A,state(A)))).
how big is texas ?
    answer(B, (area(A,B),equal(A,stateid(texas)))).
what is the area of idaho ?
    answer(B, (area(A,B),equal(A,stateid(idaho)))).
what state has the capital salem ?
    answer(B, (state(B),capital(B,A),equal(A,cityid(salem,_)))).
which state borders most states ?
    answer(B, most(B,A,(state(B),next_to(B,A),state(A)))).
what is the highest point in iowa ?
    answer(B, (high_point(A,B),equal(A,stateid(iowa)))).
what is the population of utah ?
    answer(B, (population(A,B),equal(A,stateid(utah)))).
how many rivers are in colorado ?
    answer(C, count(B,(river(B),loc(B,A),equal(A,stateid(colorado))),C)).
what state has the highest elevation ?
    answer(B, highest(A,(state(B),high_point(B,A)))).
```

what is the biggest city in the us ?

    answer(A, largest(A,city(A))).

what states border montana ?

    answer(B, (state(B),next_to(B,A),equal(A,stateid(montana)))).

what is the highest point in colorado ?

    answer(B, (high_point(A,B),equal(A,stateid(colorado)))).

what is the smallest city in washington ?

    answer(B, smallest(B,(city(B),loc(B,A),equal(A,stateid(washington))))).

what is the biggest city in oregon ?

    answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(oregon))))).

what is the population of portland maine ?

    answer(B, (population(A,B),equal(A,cityid(portland,me)))).

what is the biggest river in illinois ?

    answer(B, largest(B,(river(B),loc(B,A),equal(A,stateid(illinois))))).

what is the area of wisconsin ?

    answer(B, (area(A,B),equal(A,stateid(wisconsin)))).

what is the highest point in montana ?

    answer(B, (high_point(A,B),equal(A,stateid(montana)))).

what rivers are in utah ?

    answer(B, (river(B),loc(B,A),equal(A,stateid(utah)))).

what is the population of tucson ?

    answer(B, (population(A,B),equal(A,cityid(tucson,_)))).

what is the biggest city in georgia ?

    answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(georgia))))).

what is the capital of north dakota ?

    answer(B, (capital(A,B),equal(A,stateid('north dakota')))).

what is the lowest point in massachusetts ?

    answer(B, (low_point(A,B),equal(A,stateid(massachusetts)))).

give me the cities in virginia .

    answer(B, (city(B),loc(B,A),equal(A,stateid(virginia)))).

what is the population of oregon ?

    answer(B, (population(A,B),equal(A,stateid(oregon)))).

what is the highest point in wyoming ?

    answer(B, (high_point(A,B),equal(A,stateid(wyoming)))).

what is the capital of vermont ?

answer(B, (capital(A,B),equal(A,stateid(vermont)))).

which rivers flow through alaska ?

answer(B, (river(B),traverse(B,A),equal(A,stateid(alaska)))).

what is the longest river in texas ?

answer(B, longest(B,(river(B),loc(B,A),equal(A,stateid(texas))))).

what states border kentucky ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(kentucky)))).

what is the most populous state ?

answer(B, largest(A,(population(B,A),state(B)))).

what is the area of the smallest state ?

answer(B, (area(A,B),smallest(A,state(A)))).

what is the area of the largest state ?

answer(B, (area(A,B),largest(A,state(A)))).

which state has the highest elevation ?

answer(B, highest(A,(state(B),high_point(B,A)))).

which states does the missouri river run through ?

answer(B, (state(B),equal(A,riverid(missouri)),traverse(A,B))).

which state has the highest peak in the country ?

answer(B, highest(A,(state(B),high_point(B,A)))).

how many people live in spokane washington ?

answer(B, (population(A,B),equal(A,cityid(spokane,wa)))).

how many major rivers cross ohio ?

answer(C, count(B,(major(B),river(B),traverse(B,A),equal(A,stateid(ohio))),C)).

what is the capital of washington ?

answer(B, (capital(A,B),equal(A,stateid(washington)))).

which state has the highest population density ?

answer(B, largest(A,(state(B),density(B,A)))).

what is the smallest city in the usa ?

answer(A, smallest(A,city(A))).

what is the shortest river in the us ?

answer(A, shortest(A,river(A))).

how many square kilometers in the us ?

answer(B, (area(A,B),equal(A,countryid(usa)))).

what is the city with the smallest population ?

answer(B, smallest(A,(city(B),population(B,A)))).

what is the smallest city in alaska ?

answer(B, smallest(B,(city(B),loc(B,A),equal(A,stateid(alaska))))).

what is the largest city in rhode island ?

answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid('rhode island'))))).

how many cities are there in the us ?

answer(B, count(A,city(A),B)).

what is the shortest river in iowa ?

answer(B, shortest(B,(river(B),loc(B,A),equal(A,stateid(iowa))))).

what is the highest point in the usa ?

answer(B, (high_point(A,B),equal(A,countryid(usa)))).

which state border kentucky ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(kentucky)))).

what are all the rivers in texas ?

answer(B, (river(B),loc(B,A),equal(A,stateid(texas)))).

which state borders the most states ?

answer(B, most(B,A,(state(B),next_to(B,A),state(A)))).

which states border texas ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(texas)))).

through which states does the mississippi run ?

answer(B, (state(B),equal(A,riverid(mississippi)),traverse(A,B))).

which river runs through the most states ?

answer(B, most(B,A,(river(B),traverse(B,A),state(A)))).

what is the highest mountain in the us ?

answer(B, (high_point(A,B),equal(A,countryid(usa)))).

what are the high points of states surrounding mississippi ?

answer(C, (high_point(B,C),next_to(B,A),equal(A,stateid(mississippi)))).

what is the highest point in states bordering georgia ?

answer(C, highest(C,(high_point(B,C),next_to(B,A),equal(A,stateid(georgia))))).

which rivers run through states bordering new mexico /

answer(C, (river(C),traverse(C,B),next_to(B,A),equal(A,stateid('new mexico')))).

what are the major cities in oklahoma ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(oklahoma)))).

what is the capital of new jersey ?

answer(B, (capital(A,B),equal(A,stateid('new jersey')))).

what is the lowest point in nebraska in meters ?

answer(B, (low_point(A,B),equal(A,stateid(nebraska)))).

what is the highest point in nevada in meters ?

answer(B, (high_point(A,B),equal(A,stateid(nevada)))).

what major rivers run through illinois ?

answer(B, (major(B),river(B),traverse(B,A),equal(A,stateid(illinois)))).

what states border arkansas ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(arkansas)))).

how many people live in washington ?

answer(B, (population(A,B),equal(A,stateid(washington)))).

what is the shortest river in the usa ?

answer(A, shortest(A,river(A))).

what are the populations of states through which the mississippi river runs ?

answer(C, (population(B,C),state(B),traverse(A,B),equal(A,riverid(mississippi)))).

name the rivers in arkansas .

answer(B, (river(B),loc(B,A),equal(A,stateid(arkansas)))).

how many states does the mississippi river run through ?

answer(C, count(B,(state(B),equal(A,riverid(mississippi)),traverse(A,B)),C)).

which state has the highest point ?

answer(B, highest(A,(state(B),high_point(B,A)))).

what state has the highest population ?

answer(B, largest(A,(state(B),population(B,A)))).

which states does the mississippi river run through ?

answer(B, (state(B),equal(A,riverid(mississippi)),traverse(A,B))).

what is the highest elevation in new mexico ?

answer(B, (high_point(A,B),equal(A,stateid('new mexico')))).

what is the biggest city in arizona ?

answer(B, largest(B,(city(B),loc(B,A),equal(A,stateid(arizona))))).

what states border georgia ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(georgia)))).

what are the major cities in rhode island ?

answer(B, (major(B),city(B),loc(B,A),equal(A,stateid('rhode island')))).

what are the major cities in texas ?

    answer(B, (major(B),city(B),loc(B,A),equal(A,stateid(texas)))).

what is the population of dallas ?

    answer(B, (population(A,B),equal(A,cityid(dallas,_)))).

what is the smallest city in the us ?

    answer(A, smallest(A,city(A))).

what state has highest elevation ?

    answer(C, largest(B,(state(C),high_point(C,A),elevation(A,B)))).

name all the rivers in colorado .

    answer(B, (river(B),loc(B,A),equal(A,stateid(colorado)))).

what is the largest city in minnesota by population ?

    answer(C, largest(B,(city(C),loc(C,A),equal(A,stateid(minnesota)),population(C,B)))).

what is the population density of wyoming ?

    answer(B, (density(A,B),equal(A,stateid(wyoming)))).

what states border new hampshire ?

    answer(B, (state(B),next_to(B,A),equal(A,stateid('new hampshire')))).

what rivers run through new york ?

    answer(B, (river(B),traverse(B,A),equal(A,stateid('new york')))).

what rivers run through west virginia ?

    answer(B, (river(B),traverse(B,A),equal(A,stateid('west virginia')))).

whats the largest city ?

    answer(A, largest(A,city(A))).

what is the smallest state by area ?

    answer(B, smallest(A,(state(B),area(B,A)))).

what is the highest point in the us ?

    answer(B, (high_point(A,B),equal(A,countryid(usa)))).

which state has the greatest population ?

    answer(B, largest(A,(state(B),population(B,A)))).

how long is the mississippi ?

    answer(B, (len(A,B),equal(A,riverid(mississippi)))).

what is the highest point in rhode island ?

    answer(B, (high_point(A,B),equal(A,stateid('rhode island')))).

how many citizens live in california ?

    answer(B, (population(A,B),equal(A,stateid(california)))).

which states border arizona ?

answer(B, (state(B),next_to(B,A),equal(A,stateid(arizona)))).

what state is columbus the capital of ?

answer(B, (state(B),equal(A,cityid(columbus,_)),capital(B,A))).

what is the state with the lowest population ?

answer(B, smallest(A,(state(B),population(B,A)))).

Bibliography

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

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

Anderson, J. R. (1977). Induction of augmented transition networks. *Cognitive Science*, *1*, 125–157.

Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.

Banerji, R. B. (1992). Learning theoretical terms. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 93–110. Academic Press, New York, NY.

Beckwith, R., Fellbaum, C., Gross, D., & Miller, G. (1991). Wordnet: A lexical database organized on psycholinguistic principles. In Zernik, U. (Ed.), *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, pp. 211–232. Lawrence Erlbaum, Hillsdale, NJ.

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

Berwick, R. C., & Pilato, S. (1987). Learning syntax by automata induction. *Machine Learning*, *2*(1), 9–38.

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.

Black, E. e. (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.

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.

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

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

Charniak, E. (1993). *Statistical Language Learning.* MIT Press.

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. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 268–276 Austin, TX.

Cohen, W. W. (1993). Pac-learning a resticted class of recursive logic programs. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 86–92 Washington, D.C.

Cohen, W. (1992). Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 102–110 Aberdeen, Scotland.

Dahl, V., & McCord, M. C. (1983). Treating coordination in logic grammars. *American Journal of Computational Linguistics*, *9*(2), 69–91.

DeJong, G. F., & Mooney, R. J. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*(2), 145–176.

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.

Gazdar, G., & Mellish, C. (1989). *Natural Language Processing in Prolog*. Adison-Wesley Publishing Company, New York.

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

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.

Klein, S., & Kuppin, M. A. (1970). An interactive, heuristic program for learning transformational grammars. Tech. rep. TR-97, Computer Sciences Department, Univeristy of Wisconsin, Madison, Madison, WI.

Knowlton, K. (1962). *Sentence Parsing with a Self-Organizing Heuristic Program.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*(1).

Langley, P. (1982). Language acquisition through error recovery. *Cognition and Brain Theory*, *5*.

Langley, P. (1985). Learning to search: From weak methods to domain specific heuristics. *Cognitive Science*, *9*(2), 217–260.

Langley, P., & Carbonell, J. (1985). Language acquisition and machine learning. In MacWhinney, B. (Ed.), *Mechanisms of Language Acquisition*, pp. 115–155. Lawrence Erlbaum Associates, Inc/, Hillsdale, NJ.

Lapointe, S., & Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 273–281 Aberdeen, Scotland.

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

Lehman, J. F. (1992). *Adaptive Parsing.* Kluwer Academic Publishers, Boston.

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

Litman, D. J. (1994). Classifying cue phrases in text and speech using machine learning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* Seattle, WA.

Magerman, D. M. (1994). *Natrual 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.

Miikkulainen, R. (1993). *Subsymbolic Natural Language Processing: An Integrated Model of Scripts, Lexicon, and Memory.* MIT Press, Cambridge, MA.

Miikkulainen, R. (1995). Subsymbolic case-role analysis of sentences with embedded clauses. *Cognitive Science*. in press.

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.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 564–569 St. Paul, MN.

Mitchell, T. (1983). Learning and problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pp. 1139–1151 Karlsruhe, West Germany.

Mitchell, T., Utgoff, T., & Banerji, R. (1983). Learning problem solving heuristics by experimentation. In Michalski, R., Mitchell, T., & Carbonell, J. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Palo Alto, CA.

Mitchell, T. M. (1984). Toward combining empirical and analytic methods for learning heuristics. In Elithorn, A., & Banerji, R. (Eds.), *Human and Artificial Intelligence*. North-Holland, Amsterdam.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*(1), 47–80.

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, in press.*

Muggleton, S. (1992). Inverting implication. In *Proceedings of the Second International Workshop on Inductive Logic Programming* Tokyo, Japan.

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

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

Muggleton, S., 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.

Ng, H. T. (1988). A computerized prototype natural language tour guide. Tech. rep. AI88-75, Artificial Intelligence Laboratory, University of Texas, Austin, TX.

Pazzani, M., Brunk, C., & Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 432–436 Evanston, IL.

Periera, F., & Schabes, 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. (1986). The effect of noise on concept learning. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Volume II*, pp. 149–166. Morgan Kaufman.

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.

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

Rouveirol, C. (1992). Extensions of inversion of resolution applied to theory completion. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 63–86. Academic Press, New York, NY.

Selfridge, M. (1981). A computer model of child language acquisition. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 106–108 Vancouver, B.C.

Sembugamoorthy, V. (1981). A paradigmatic language acquisition system. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 106–108 Vancouver, B.C.

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

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

Soderland, S., & Lehnert, W. (1994). Corpus-driven knowledge acquisition for discourse analysis. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* Seattle, WA.

Solomonoff, R. (1959). A new method for discovering the grammars of phrase structure languages. In *Proceedings of the International Conference on Information Processing.*

St. John, M. F., & McClelland, J. L. (1990). Learning and applying contextual constraints in sentence comprehension. *Artificial Intelligence*, *46*, 217–257.

Stahl, I., Tausend, B., & Wirth, R. (1993). Two methods for improving inductive logic programming systems. In *Machine Learning: ECML-93*, pp. 41–55 Vienna.

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

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

VanLehn, K., & Ball, W. (1987). A version space approach to learning context-free grammars. *Machine Learning*, *2*(1), 39–74.

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

Wirth, R., & O'Rorke, P. (1991). Constraints on predicate invention. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 457–461 Evanston, IL.

Wirth, R. (1988). Learning by failure to prove. In *Proceedings of EWSL 88*, pp. 237–51. Pitman.

Wirth, R. (1989). Completing logic programs by inverse resolution. In *Proceedings of the European Working Session on Learning*, pp. 239–250 Montpelier, France. Pitman.

Wolff, J. G. (1982). Language acquisition, data compression, and generalization. *Language and Communication*, *2*, 57–89.

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

Zelle, J. M., & Mooney, R. J. (1993a). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial intelligence*, pp. 1106–1111 Chambery, France.

Zelle, J. M., & Mooney, R. J. (1993b). Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 817–822 Washington, D.C.

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* 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.

Vita

John Marvin Zelle was born in Waterloo, Iowa, on May 24, 1962, the son of Marie Gertrude (Meyer) Zelle and Marvin John Zelle. After completing his work at Waverly-Shell Rock Senior High School, Waverly, Iowa, in 1980, he entered Iowa State University in Ames, Iowa. He received the degree Bachelor of Science from Iowa State University in 1984, and the degree Master of Science from Iowa State University in 1986. During the following years he was employed as Assistant Professor of Computer Science and Mathematics at Wartburg College in Waverly, Iowa. In September of 1990 he entered the Graduate School of the University of Texas.

Permanent Address: 700 McNeil Rd. #521
Round Rock, TX 78681

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.