LEARNING BASED PROGRAMMING

BY

NICHOLAS D. RIZZOLO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Abstract

Machine learning (ML) is the study of representations and algorithms used for building functions that improve their behavior with experience. Today, researchers in many domains are applying ML to solve their problems when conventional programming techniques have proven insufficient. The first such were simple, computing the setting of only a single output variable. More recently, *structured learning* is adding its own set of challenges, and the specifications designed for such *learning based programs* are not scaling well, nor are the programs they represent.

This thesis introduces Learning Based Programming (LBP), the study of programming language formalisms that directly support programs that learn their representations from data. An LBP framework embodies a set of design principles that ensures learning based programs designed under its formalism are composable, prepared for an infinite feature space, and not necessarily dependent on any particular learning or inference algorithms, among other things. We submit that adherence to these principles is necessary to enable the implementation of learning based programs that scale beyond today's implementations.

To ensure independence from learning and inference algorithms, we present the Constrained Conditional Model (CCM) and demonstrate that it is general enough to subsume the majority of models described in the literature. We then present Learning Based Java (LBJ), our first attempt at implementing an LBP framework which supports an important subset of CCMs. LBJ is a *discriminative* modeling language that abstracts away implementation details of feature extraction, learning, and inference. It also features a First Order Logic inspired syntax for expressing constraints between independently trained classifiers. LBJ has already been used successfully in a variety of Natural Language Processing tasks.

We evaluate LBJ with both a comprehensive questionnaire and case studies. A variety of participants responded to the questionnaire, including varying levels of experience in ML and LBJ. They also applied LBJ on different tasks ranging in complexity. Simpler tasks tended to lead to a positive evaluation of LBJ, while those who required more advanced ML techniques, especially structured representations, found LBJ lacking. The case studies came to a similar conclusion while uncovering situations in which LBJ's syntactic sugar could have encoded specifications more succinctly, as well as specific pieces of syntax that attempted to patch LBJ's limitations.

The results of our evaluation show that LBJ leaves something to be desired; in particular, it cannot naturally specify an arbitrary CCM. As such, we present our second offering in the LBP line: a general purpose programming language called the Constrained Conditional Model Processing language and designed from the ground up to support CCMs. We also present a formal semantics for CCMP specified in the language of rewriting logic and consider several interesting test cases. CCMP is a robust and flexible solution for structured learning techniques at both training-time and inference-time.

Acknowledgments

I am eternally grateful to my advisor Dan Roth, without whom this work would not have been possible. Dan is a fountain of knowledge and ideas, and his unflappable optimism and patience motivates implicitly. We in the Cognitive Computations Group reach farther than we would have thought possible were it not for his guidance.

By the way, this CCG that Dan has created is continually full of bright, energetic researchers, and it has been my privilege to benefit from their diverse insights over the years. In particular, I owe thanks to Shivani Agarwal, Eric Bengtson, Rodrigo de Salvo Braz, Andy Carlson, Yee Seng Chan, Kai-Wei Chang, Ming-Wei Chang, James Clarke, Michael Connor, Chad Cumby, Quang Do, Yair Even-Zohar, Ashutosh Garg, Dan Goldwasser, Prateek Jindal, Alex Klementiev, Gourab Kundu, Xin Li, Jeff Pasternack, Vasin Punyakanok, Lev Ratinov, Alla Rozovskaya, Rajhans Samdani, Mark Sammons, Kevin Small, Vivek Srikumar, Yuancheng Tu, Tim Vieira, V.G.Vinod Vydiswaran, Wen-tau Yih, and Dav Zimak. Everyone on that list helped shape my thinking, thereby improving this work. Last but not least, I must express my gratitude to the programmers who helped me implement Learning Based Java 2: Jason Cho, Dan Muriello, Michael Paul, and Arindam Saha.

I'd also like to thank the members of my committee for their time and helpful comments in the last stages of my work: Gerald DeJong, Grigore Roşu, and Avrom Pfeffer. Jerry's keen insight exposes flaws at the heart of the matter, and his sage advice helps resolve them. Grigore's analytical framework for programming languages has directly influenced the development of my own ideas. Finally, it was an honor to have Avi on my committee, as his body of work at the intersection of machine learning and programming languages has been an inspiration to me throughout my graduate studies.

There will never be enough time or adequate words with which to thank my family and friends.

Of course, it all starts with my parents' support that I too often take for granted. They, and my brother and sister, and my entire extended family has always been there for me, and I'll never forget it. Special thanks also goes to the Thursday night poker crew for a camaraderie that kept my feet on the ground and everything in perspective. Finally, I am forever indebted to the most patient, forgiving, loving woman I know, my wife Lori. I couldn't have done it without her.

Table of Contents

List of Abbreviations

CCG	Cognitive Computations Group
CCMP	Constrained Conditional Model Processing
CCM	Constrained Conditional Model
CD	Cognitive Dimension
CM	Conditioned Model
CNF	Conjunctive Normal Form
CRF	Conditional Random Field
DAG	Directed Acyclic Graph
FGF	Feature Generation Function
FOL	First Order Logic
HMM	Hidden Markov Model
ILP	Integer Linear Programming
IM	Instantiated Model
LBJ	Learning Based Java
LBP	Learning Based Programming
LTU	Linear Threshold Unit
ML	Machine Learning
NLP	Natural Language Processing

Chapter 1

Introduction

Machine learning (ML) is the study of representations and algorithms used for building functions that improve their behavior with experience. It has become indispensable in any domain in which high performance hard-coded solutions elude developers. Common examples include decision trees learned by maximizing information gain, generative probabilistic models learned by maximizing likelihood, expectation, or entropy, and linear models learned by maximizing margin. In all cases, the ultimate goal is to derive from data a function that maps from input to output variable settings.

Today, researchers in many domains are applying ML to solve their problems when conventional programming techniques have proven insufficient. The fields of natural language processing (NLP), robotics, vision, human computer interaction, compilers, and others are all employing ML to harness the information in large data sets and to solve complex problems involving many competing interests. We refer to any such system with a learned component as a *learning based program* and to the study of learning based programs as *Learning Based Programming* (LBP).

The first learning based programs were simple, computing the setting of only a single output variable. Even so, non-trivial applications commonly demand a very high dimensional feature space, and provisions must be made to ensure efficiency. As researchers turned their attention to more challenging problems, they initially aimed to keep their approach simple. Individual components would again contribute the setting for a single output variable, but they would often include as input the outputs of other components. The resulting system would typically be referred to as a *pipeline*. Typical implementations of pipeline systems designed each component as a separate process with its own input and output formats and were unnecessarily large and unwieldy given the simplicity of their high level organization. The learned components and their feature extraction codes would be glued together with a scripting language such as PERL or bash, with hand-coded data representation translations at each stage. Implementing systems in this way can be tedious and error prone. This inefficient style of system design is borne not only out of researchers' desires to get results quickly, but also their need to understand and fine tune each component in isolation [Patel et al., 2008].

More recently, *structured learning* is adding its own set of challenges, and implementations of such learning based programs are not scaling well. Researchers wish to express the interactions and relationships between output variables either via features in a joint learning framework, via constraints enforced only at inference-time, or any combination of the two. A combination leads to the same implementation difficulties as those mentioned above. Furthermore, the initial choice of how to divide these labors can have far-reaching consequences for an implementation. For instance, given a jointly trained system with many output variables, it can be quite technically challenging to re-purpose the system as a pipeline with independently trained components, or vice-versa. It may even call for a new design starting from scratch, even though the quantities of learned parameters and their roles at inference-time will not change. We believe these issues can be addressed with a principled LBP formalism.

1.1 Design Principles

In Chapter 2, we survey several works that have made progress towards the design of an LBP language. None of them embody all of LBP's principles, which we will now explicate:

- *High-level primitives for feature extraction and inference*: An LBP language must abstract the bookkeeping of feature extraction and inference away from the application programmer. This includes both feature indexing, primarily a compile/training-time concern, and run/inference-time generation of constraints in terms of the input data.
- *Relational features*: Most non-trivial domains are *relational* in nature; i.e., they consist of objects that themselves have attributes, but that are also connected to each other via relations. The simplest example of this is the internet, which contains web pages that are linked to each other. An LBP language must be capable of representing features based on this link structure (as well as the attributes).

- *Infinite feature space* [Blum, 1992]: If learning based programs are to be scalable, features and constraints cannot be static elements of a model. In non-trivial domains, there are simply too many to name explicitly. Thus, an LBP language must be capable of expressing models in terms of *feature generation functions* [Cumby and Roth, 2003] over the data. This means the quantity of model parameters is not known until data is observed.
- *Customizable objective function*: Features and constraints are not the only opportunity for the application programmer to influence the outcome of inference. He should also be capable of re-weighting the objective function to (for example) emphasize the most important decisions being made.
- *Model composability*: An LBP language must provide for the construction of larger models based on smaller ones. Ideally, the models themselves will be first-class objects of the language. Most importantly, we need to be able to leverage today the models that were developed and learned yesterday.
- *Inference decomposability*: When there are many output variables, joint inference can quickly become intractable. Developers may wish to organize the computation in a *pipeline* in which some variables are computed before, and then act as input for, inference over other variables. In an LBP language, inference should be decomposable in this way without requiring a refactorization of the model code.
- *Algorithm independence*: The code that specifies a model in an LBP language must not assume that any particular learning or inference algorithms will be employed. Instead, it should simply establish the *shape* of the model (i.e., what parameters are involved and what features index them), and the various algorithms can accept or reject the model as appropriate.

1.2 Learning Based Programming

While algorithmic issues in learning and inference garner the bulk of researchers' attention, LBP also brings to light interesting theoretical and practical challenges in the development and run-time optimization of learning based programs. On the theoretical side, are there significant differences between popular models in the literature, or can they be described under a common framework?

Doing so is a prerequisite for algorithm independent composability. To address this question, in Chapter 3 we develop the Constrained Conditional Model (CCM) [Chang et al., 2008] and demonstrate that it is general enough to subsume the majority of models described in the literature.

On the practical side, can this framework allow simple, high level specifications of learning based programs from which efficient code can be generated? To address this question, we first present Learning Based Java (LBJ) [Rizzolo and Roth, 2007], our first generation LBP language in Chapter 4. It is the first *discriminative* modeling language to our knowledge; its learned models need not represent probability distributions. In addition to abstracting the feature extraction and indexing process, the language also provides a declarative, First Order Logic (FOL) syntax for writing constraints grounded in the Java objects that encapsulate the user's data. This syntax can then be compiled at run-time into linear inequalities suitable as input for an Integer Linear Program (ILP) solver. LBJ has already been used to develop several state-of-the-art resources. The LBJ POS tagger[1] reports a competitive 96.6% accuracy on the standard Wall Street Journal corpus. In the named entity recognizer of [Ratinov and Roth, 2009], non-local features, gazetteers, and Wikipedia are all incorporated into a system that achieves 90.8 F_1 on the CoNLL-2003 dataset, the highest score we are aware of. The co-reference resolution system of [Bengtson and Roth, 2008] achieves state-of-the-art performance on the ACE 2004 dataset while employing only a single learned classifier and a single constraint. Finally, LBJ's constraint and inference framework was used successfully to recognize authority in dialogue in [Mayfield and Rosé, 2011].

1.3 Evaluation

LBJ's success stories are proof that it is useful in certain situations. However, it is important not to rest on our laurels. Thus, we seek in Chapter 5 to perform a thorough and objective evaluation of LBJ; a proposition which is easier said than done. After a survey of alternative programming language evaluation methodologies from the literature, we employ the Cognitive Dimensions of Notations [Green, 1989, Blackwell and Green, 2000] to elicit constructive criticism from users of the language via a comprehensive questionnaire. The results of this study highlight important

[1] `http://cogcomp.cs.illinois.edu/page/software_view/3`

shortcomings of LBJ, including that it is not relational (i.e., structured), and that it hides too many low-level implementation details that can be crucial for modeling expressivity and scalability.

In addition, we present in Chapter 6 a case study of several learning based programs that collectively span the set of the design principles outlined in Section 1.1. LBJ implementations of all of them are possible, but sometimes not without making special accommodations outside of the language or forcing the issue with unnatural representations. The case studies uncovered situations in which LBJ's syntactic sugar could have encoded specifications more succinctly, as well as specific pieces of syntax that attempted to patch LBJ's limitations.

Chapters 5 and 6 inspire us to desire more in an LBP language; most importantly, support for structured models and learning algorithms.

1.4 Structured Learning Based Programming

To fully incorporate structure into an LBP language while simultaneously addressing the concerns of the evaluation participants, we developed the Constrained Conditional Model Processing language (CCMP) [2]. CCMP is a general purpose programming language in which features, vectors, and models (i.e. structured collections of learned parameters) are primitive data types and operators are provided for composing and accessing them. This model composition is completely divorced from any notion of learning or inference over the model and its parameters. Thus, it is up to the programmer to decide on the semantics of those learned parameters and to design (or select from a library) learning and inference algorithms that respect those semantics.

We also present a formal semantics for CCMP designed using the K technique [Roşu and Şerbănuţă, 2010] in the rewriting logic language Maude [Clavel et al., 2007]. The equations and rules therein give a precise, logical account of all the computations that take place when any operator is applied in a CCMP code. Since Maude code is executable, we also get an interpreter and debugger of CCMP for free, and thus we have been able to evaluate the capabilities of the language on several important test cases. CCMP is a robust and flexible solution for structured learning techniques at both training-time and inference-time.

[2]See [Rizzolo and Roth, 2010] for an intermediate step between LBJ and CCMP

Chapter 2

Related Work

Over the past 20 years, several machine learning modeling formalisms have been developed, all for the specification of probabilistic models. The first such languages were for modeling Bayes nets, a formalism in which models often enjoy a natural decomposition, but which can be awkward when causal relationships between variables are hard to justify. More recent languages focus on undirected models for relational data comprised of repeated structure in which relational features are associated with shared parameters.

The LBP principles embodied by these approaches are summarized in Table 2.1. As shown in the table, there is limited support for model composition (collectively), and few formalisms give the programmer access to the inference objective function. All of these languages act as interfaces to particular learning and inference algorithms, and typically aim to solve both problems globally. However, there are often big advantages to composing the model from separately trained components whose interactions are resolved only at inference-time [Punyakanok et al., 2008]. In particular, the simpler component models will require fewer training examples to converge, and the computational complexity of inference can be kept under control [Denis and Baldridge, 2007, Martins et al., 2009]. Finally, although data participates in the definition of each learned function, no existing framework leverages that data to generate better code.

2.1 Generative Modeling Languages

2.1.1 BUGS

The Bayesian inference Using Gibbs Sampling (BUGS) language [Gilks et al., 1994] is a declarative, propositional language for specifying the structure of a *hybrid* generative model (i.e., one whose

LBP Principle	AB	PRISM	PRM	IBAL	BLOG	Church	RMN	MLN	FACTORIE
High-level primitives	✓	✓		✓	✓	✓		✓	
Relational features		✓	✓	✓	✓	✓	✓	✓	✓
Infinite feature space		✓	✓	✓		✓	✓	✓	✓
Customizable objective function	✓								
Model composability		✓		✓		✓			
Inference decomposability	✓								
Algorithm independence									

Table 2.1: Embodiment of LBP design principles by the languages and formalisms surveyed in Chapter 2.

random variables may be either discrete or continuous). Backed by a library of Bayesian techniques, the specified model may involve variables drawn from various types of distributions (e.g. normal, beta, Dirichlet, Bernoulli, and binomial), and the parameterizations of the distributions can take a variety of exponential family forms. As the name of the system says, this language acts as an interface to Gibbs sampling [Casella and George, 1992] for both learning and inference. In addition, its language is propositional, and it only supports static features.

2.1.2 AutoBayes

AutoBayes [Fischer and Schumann, 2003] picks up where BUGS left off, providing an ambitious set of tools for processing Bayes nets. From a high level, the two languages look very similar. However, an AutoBayes source file may also contain the objective function to optimize during learning, which is usually the likelihood of the data given the parameters. Using a library of pattern matching schemas, AutoBayes then applies a recursive, symbolic optimization procedure in which the objective function is decomposed according to independence theorems, and subproblems are solved analytically when possible. When an analytic solution cannot be found, AutoBayes falls back on EM, which itself contains a maximization over which this entire recursive procedure proceeds. The result is an automatically derived algorithm tailored to the user's network structure. AutoBayes then generates heavily commented C++, separate documentation explaining the derivations, and synthetic testing data.

The AutoBayes language is propositional, and it only supports static features. But this should not draw our attention away from what the system achieves as a high-level language. A library of Bayes net routines cannot optimize the global learning procedure as AutoBayes does without a symbolic representation of the network. How can that representation be provided other than with a high-level language?

2.1.3 PRISM

PRISM [Sato and Kameya, 1997, Sato et al., 2005], short for PRogramming In Statistical Modeling, is a language that defines definite (i.e., Horn) clause logic programs with a special predicate that signals the presence of a nameable Boolean random variable. This "binary switch" predicate may only appear in the body of a clause. With the name of the produced Boolean variable being passed to the predicate as an argument, PRISM's models are in the infinite attribute domain and can also accept relational features. However, it is assumed that the parameters will be learned via EM given only the observed values of head predicates. PRISM does have enough expressivity to model arbitrary Bayes nets, even if in a somewhat more verbose syntax than we might have hoped for.

The system is implemented as an extension to Prolog, a Turing-complete logic programming language. The same source code that specifies the model's structure is then evaluated after learning with the parameters filled in. Thus, PRISM is a general purpose language whose programs are only partially specified until data is observed. Indeed, one of our main goals is to emulate and improve on this philosophy.

2.1.4 Probabilistic Relational Models

Probabilistic Relational Models (PRMs) [Getoor et al., 2000] extend Bayes nets to a relational context involving quantified relations between objects of different classes. To specify a PRM, one specifies a schema of object classes, their attributes including references to other objects, and dependencies between the attributes. When data is observed, the schema is instantiated into a Bayes net. Thus, it is very similar to the undirected frameworks discussed in Section 2.2, except for the following key points: (1) it is less flexible with respect to features, (2) special care must be taken to ensure that each instantiation is acyclic, and (3) whenever there is a many-to-one mapping from class X to class Y and attribute $Y.A$ depends on attributes $X.B$, these dependencies must be aggregated to keep the size of the conditional probability table constant across instantiations. These inconveniences are all due to the intended interpretation of the learned parameters as conditional probabilities, which we consider a very limiting restriction.

2.1.5 IBAL

The Integrated Bayesian Agent Language (IBAL) [Pfeffer, 2007] is a general purpose, functional programming language with a variety of novel primitives for representing learnable probabilistic and decision theory models. We focus on its probabilistic modeling capabilities here. In particular, primitive to the language is a value representing a learnable, parameterized distribution over other values. This distribution value may appear anywhere the generated values would be allowed. A program can then be interpreted as a generative story, conveniently specified in a declarative programming environment. IBAL can also represent undirected models as products of experts via declarative constraints. With all these capabilities fully integrated into a Turing-complete language, it can even capture the same types of long-range features as FACTORIE's imperatively defined factor graphs (see Section 2.2.3). It does, however, assume a particular style of learning and inference over statically declared parameters that must represent probabilities.

2.1.6 BLOG

The Bayesian Logic (BLOG) [Milch et al., 2007] language is a first-order language for specifying the structure of a Bayes net and querying that network given evidence. It is essentially an extension of PRMs to domains involving uncertainty in the existence or quantity of objects. However, learning has not been incorporated in the language, and inference is approximate and not guaranteed to terminate for all model structures.

2.1.7 Church

The Church language for specifying generative models [Goodman et al., 2008] is essentially Scheme[1] with built-in primitives for sampling values from distributions and conditional querying. Similarly to IBAL, Church expressions represent a distribution over possible return values, except that when evaluated, a sample from the distribution is returned instead of the distribution itself. Another feature that sets Church apart is its *stochastic memoization* facility. At the user's behest, any stochastic function can be memoized so that the returned sample is remembered in association with the function's arguments. This way, the same generative history can be referenced from different sections of the code, specifying (for example) the model and a conditional query over it. Learning is accomplished via conditioning, and MCMC inference is performed.

2.2 Undirected Graphical Modeling Languages

2.2.1 Relational Markov Networks

Relational Markov Networks (RMNs) [Taskar, 2002] incorporate repeated structure in an undirected graphical model by assuming an exponential form for the factors and associating a single set of parameters with a repeatedly instantiated *relational clique template*. The template itself is written in SQL, and it represents a relational query over domain elements selecting desired attributes to take part in conjunctive features. The result is a joint distribution whose log is a dot product of parameters and counts of feature occurrences, just like the discriminatively trained HMM of Collins [Collins, 2002]. The difference is that an RMN's scores must be normalized. Approximate inference can be performed with belief propagation or MCMC.

2.2.2 Markov Logic Networks

Markov Logic Networks (MLNs) [Richardson and Domingos, 2006] are exponential models in which the features are expressed as First Order Logic (FOL) formulae over a symbolic domain. They are named as such because the weighted features can be interpreted either as a Markov random field

[1]Scheme is a dialect of Lisp; see http://groups.csail.mit.edu/mac/projects/scheme

or as a probabilistic knowledge base. Given constant symbols representing objects, the weighted features can be thought of as inducing a Markov random field in which the nodes represent ground predicates, and there is an edge between two nodes iff the corresponding ground predicates appear in the same grounding of some feature formula. As the weights approach infinity, the formulae become hard constraints, and the network can be viewed as a knowledge base in the FOL sense.

However, the function of logic in this formalism is primarily for the specification of relational features. Learning and inference are accomplished via standard exponential model techniques. Indeed, an FOL feature formula is essentially an extension of, and a more compact syntax for, the RMN's relational clique template. The difference is that the features need not be conjunctive. However, the MLN language does not appear to provide a *conditioning operator* for disjunction or negation as in [Cumby and Roth, 2003]. Thus, non-conjunctive features are impractical in large, otherwise sparse domains such as natural language processing, since an intractable number of these features will always be active. Furthermore, MLNs do not address model composition and are designed for use with particular learning and inference algorithms.

2.2.3 FACTORIE

FACTORIE [McCallum et al., 2009] is a library for learning and inference over factor graphs. A factor graph is a bipartite, undirected graphical model where the two types of nodes represent random variables and factors of the joint distribution. Factor nodes are connected to all and only those variable nodes corresponding to the variables they contain. Thus, a factor graph can provide a more fine-grained illustration of the joint's factorization than the traditional graph composed of only random variable nodes.

The FACTORIE library allows exponential form factors and provides a factor template mechanism for specifying repeated structure, just as do RMNs and MLNs. Factor templates can include arbitrary native code to locate the pertinent random variables. This code acts as a hook into an appropriately implemented inference algorithm, enabling (effectively) the extraction of features that are difficult to specify in FOL. Thus, FACTORIE is said to instantiate *imperatively defined factor graphs* (IDFs).

Implemented as a library for Scala[2], FACTORIE cannot take advantage of high level information about the structure of a model to generate more efficient code for the given task. Instead, it is tailored specifically for use with MCMC in both learning and inference, providing hooks into the algorithm that the user must implement in accordance with the semantics of his model. Thus, they sacrifice choice of algorithms and some simplicity of model specification for flexibility within the MCMC framework. In addition, they once again do not address model composition.

[2]Scala is a language based on Java mixed with a variety of functional programming concepts; see http://www.scala-lang.org/

Chapter 3

The Constrained Conditional Model

In order to address all of LBP's design principles, we desire a learnable model that supports an infinite feature space, is composable, and, most importantly, makes no assumptions about the learning and inference algorithms used and as few as possible about the interpretation of the learned parameters themselves. If the model satisfies these properties, the remaining LBP principles will be fulfillable by a programming language adopting it, as we will show in Chapters 4 and 7.

Discriminative models will be of particular concern to us in this thesis, as they are conspicuously neglected by the formalisms described in Chapter 2. Algorithms such as Winnow [Littlestone, 1988], Perceptron [Rosenblatt, 1958], and SVM [Burges, 1998] remain popular choices in modern research because of their efficiency and good performance on real world problems. In this chapter, we show that a single formalism can capture both these models as well as the probabilistic models described in Chapter 2 (as used for discriminative purposes), in both theory and practice.

At the highest level of abstraction, a learnable model is a parameterized function $f_\theta : \mathcal{X} \to \mathcal{Y}$ that maps from an input space $\mathcal{X} \equiv \mathbb{R}^p$ of *feature values* to an output space $\mathcal{Y} \equiv \mathbb{R}^q$ of *labels* conditioned on a vector of parameters $\theta \in \mathbb{R}^n$. Each dimension of $\mathcal{X}$ corresponds to a different *feature*; i.e., a property of the data that has been evaluated to a real value. Each dimension of $\mathcal{Y}$ corresponds to a different *output variable*, which can also be thought of as a property whose value is determined by f_θ. *Learning* is the process through which θ is selected, and *inference* is the process through which f_θ is evaluated given input. Although features and output variables have real values in general, we will tend to focus on problems involving discrete features and output variables. In these contexts, Boolean values will simply be restricted to the values 0 and 1, and discrete features or output variables can be encoded with a separate Boolean for each discrete value.

In this Chapter, we first develop the *constrained conditional model* (CCM) of [Chang et al., 2008] in Section 3.1 to the play the role of f_θ. The CCM is conditional in that it supports most directly the modeling of interactions amongst output variables given the input features as opposed to jointly modeling both input and output. This may seem like a limitation, but it is an advantage in domains in which there is a strong distinction between input and output; modeling effort is never wasted on values that are always given. Note also that a fully joint model can be viewed simply as the special case with no input. Finally, the CCM is constrained in that it contains features governed by parameters in θ whose values are set by the system designer as opposed to the learning algorithm. We call these features *constraints*, and we go to the trouble of distinguishing them from features to emphasize the idea that handling them separately from the rest of the model can lead to big gains in efficiency.

After presenting the CCM, we spend the rest of the Chapter discussing its expressivity. We show that many common models in the literature fall under its umbrella. During the discussion, we keep an eye out for the design principles we aim to satisfy in a programming language implementation.

3.1 Definition

We submit the *constrained conditional model* (CCM) of [Chang et al., 2008] as a general, relational, discriminative modeling framework. A CCM can be represented by two weight vectors $\mathbf{w}$ and ρ standing for θ, a set of feature functions $\Phi = \{\phi_i : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}\}_{i=1}^{n_\phi}$, and a set of constraints $\mathcal{C} = \{c_j : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}\}_{j=1}^{n_c}$. The score for an assignment to the labels $\mathbf{y} \in \mathcal{Y}$ on an input instance $\mathbf{x} \in \mathcal{X}$ can then be obtained via the objective function

$$z_{\mathbf{w},\rho}(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{n_\phi} w_i \phi_i(\mathbf{x}, \mathbf{y}) \right) - \left(\sum_{j=1}^{n_c} \rho_j c_j(\mathbf{x}, \mathbf{y}) \right), \tag{3.1}$$

and inference is framed as an optimization problem computing the highest scoring label assignment:

$$f_{\mathbf{w},\rho}(\mathbf{x}) = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} z_{\mathbf{w},\rho}(\mathbf{x}, \mathbf{y}) \tag{3.2}$$

If the features and constraints are non-linear, this problem can be arbitrarily complex. However, there are also many interesting tractable forms the problem can take. We'll review several in this chapter. Most frequently, we'll restrict ourselves to the case where $\mathcal{Y} = \{0, 1\}^q$. Typically, this results in $\phi_i \in \mathbb{N} \cup \{0\}$ and $c_j \in \{0, 1\}$ as well. In these cases, a CCM can be solved as an ILP. Although ILP is NP-hard in general, the ability to solve a CCM as an ILP often yields efficient inference over models involving long range constraints that other modeling formalisms cannot support [Roth and Yih, 2005].

There are only two differences between the two summations in Equation (3.1) (and between features and constraints). One is that a feature's weight w_i is set by a learning algorithm, whereas a constraint's weight ρ_j is set by a domain expert. Thus, constraints are a mechanism for incorporating knowledge into the model. The other is that each c_j will be written to express the degree to which constraint j has been *violated*. ρ_j then represents the penalty incurred for a violation. This way, we can make a constraint *hard* by setting $\rho_j = \infty$ without disrupting the distinctions made by the first summation when constraints are satisfied.

The CCM is very general and subsumes many modeling formalisms. Note that CCMs are not restricted to any particular learning or inference algorithms. They can also be composed by simply summing objective functions, regardless of how the argument models were originally learned. Thus, the designer of the model can tailor the semantics of the features and weights for the task at hand. As such, many, if not all models developed in the literature fall under its umbrella. For the rest of this chapter, we will explore these claims in more depth by detailing several popular models in the CCM framework. Once they've been unified under this framework, they'll become easier to combine and augment.

3.2 Classical Models of Learning

3.2.1 Linear Threshold Units

Perceptron [Rosenblatt, 1958], Winnow [Littlestone, 1988], and other such Boolean classification algorithms represent their hypothesis with a weight vector $\mathbf{w} \in \mathbb{R}^n$ whose dimensions correspond to features of the input $\mathbf{x} \in \mathbb{R}^n$. The prediction of the model is then $y^* = I\{\mathbf{w} \cdot \mathbf{x} \geq 0\}$, where $I\{\cdot\}$ is 1 if the argument is true and 0 otherwise; i.e. the dot product between the weight vector and the features' values is compared with a threshold of 0. Thus, we refer to these models as linear threshold units (LTUs).

To cast this model as a CCM, our features' values will simply factor in the label $y \in \mathcal{Y} \equiv \{0, 1\}$:

$$\phi_i(\mathbf{x}, y) = y\, x_i \tag{3.3}$$

There are no constraints. Equations (3.1) and (3.2) then define inference. Since $\mathbf{w}$ is fixed and $\mathbf{x}$ is given, so the objective function is linear.

3.2.2 Multi-Class Classifiers

A popular approach to online multi-class classification instantiates for each class t a separate LTU $\mathbf{w}_t$, indexed by the same features of the input $\mathbf{x}$ [Carlson et al., 1999, Crammer and Singer, 2003]. The prediction is then simply the class associated with the highest scoring weight vector $y^* = \text{argmax}_{t \in \mathcal{T}}\, \mathbf{w}_t \cdot \mathbf{x}$, where $\mathcal{T}$ is the set of all classes.

To cast this model as a CCM, we first create a separate Boolean label $y_t \in \{0, 1\}$ for each $t \in \mathcal{T}$ along with a single constraint that ensures exactly one of these labels will be active given any input. Additionally, we duplicate the original features creating separate feature functions for each label and distribute the corresponding labels into them. These new feature functions take their original values if and only if the corresponding Boolean label is 1; otherwise they take the value 0. This arrangement effectively instantiates the aforementioned LTUs.

$$\phi_{t,i}(\mathbf{x}, \mathbf{y}) = y_t\, x_i \tag{3.4}$$

$$c^D(\mathbf{x}, \mathbf{y}) = I\left\{\sum_t^{\mathcal{T}} y_t \neq 1\right\} \tag{3.5}$$

$$z(\mathbf{x}, \mathbf{y}) = \left(\sum_{t,i} w_{t,i}\phi_{t,i}(\mathbf{x}, \mathbf{y})\right) - \infty\, c^D(\mathbf{x}, \mathbf{y}) \tag{3.6}$$

In equation (3.6), the objective function z from equation (3.1) is redefined with the new feature indexing scheme and single constraint whose penalty is $\rho = \infty$. Generative models used for multi-class classification such as naïve Bayes can also be viewed in this light [Roth, 1999].

3.2.3 Hidden Markov Models

The standard in sequential prediction tasks is the Hidden Markov Model (HMM) [Rabiner, 1989]. It is a generative model that incorporates (1) a probability of making each possible emission at step i and (2) a probability of being in each possible state at step $i+1$, both conditioned on the state at step i. These probabilities are usually organized into emission and transition probability tables, $P(e_i|s_i)$ and $P(s_{i+1}|s_i)$, respectively, where $s_i \in \mathcal{S}$ and $e_i \in \mathcal{E}$. During inference, the emissions e_i are fixed, the state variables s_i are our output variables, and our goal is to find the assignment that maximizes likelihood or, equivalently, log-likelihood:

$$\mathbf{s}^* = \operatorname*{argmax}_{\mathbf{s}} \prod_{i=1}^{m} P(s_i|s_{i-1})P(e_i|s_i) \tag{3.7}$$

$$= \operatorname*{argmax}_{\mathbf{s}} \sum_{i=1}^{m} \log(P(s_i|s_{i-1})) + \log(P(e_i|s_i)) \tag{3.8}$$

where m is the total number of observed emissions in a given sequence and s_0 is a sentinel state symbol placed at the beginning of every sequence.

Following [Collins, 2002], we can cast equation (3.8) as a CCM by first flattening the log probabilities into our weight vector. Next, we rearrange the equation to factor out the model's weights, which are just the individual probabilities in the two tables:

$$I_{\hat{r},\hat{r}'}(r, r') = I\{\hat{r} = r \wedge \hat{r}' = r'\} \tag{3.9}$$

$$\mathbf{s}^* = \operatorname*{argmax}_{\mathbf{s}} \sum_{\hat{s},\hat{e}}^{\mathcal{S}\times\mathcal{E}} \log(P(\hat{e}|\hat{s})) \left(\sum_{i=1}^{m} I_{\hat{s},\hat{e}}(s_i, e_i) \right) + \sum_{\hat{s},\hat{s}'}^{\mathcal{S}\times\mathcal{S}} \log(P(\hat{s}|\hat{s}')) \left(\sum_{i=1}^{m} I_{\hat{s},\hat{s}'}(s_i, s_{i-1}) \right) \tag{3.10}$$

It is now clear that our features simply count the number of occurrences of each (*state*, *emission*) pair and each pair of consecutive states in the sequence. Thus, with Boolean feature values $x_{e,i}$ indicating whether emission e is observed at step i and Boolean labels $y_{s,i}$ indicating whether the model is in state s at step i, we can formulate our CCM definition as follows:

$$\phi_{e,s}(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{q} I\{x_{e,i} = 1 \wedge y_{s,i} = 1\} \tag{3.11}$$

$$\phi_{s,s'}(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{q} I\{y_{s,i-1} = 1 \wedge y_{s',i} = 1\} \tag{3.12}$$

$$z(\mathbf{x},\mathbf{y}) = \left(\sum_{e,s}^{\mathcal{E}\times\mathcal{S}} w_{e,s}\phi_{e,s}(\mathbf{x},\mathbf{y})\right) + \left(\sum_{s,s'}^{\mathcal{S}\times\mathcal{S}} w_{s,s'}\phi_{s,s'}(\mathbf{x},\mathbf{y})\right) \tag{3.13}$$

where $q = m$.

Our objective function (3.13) is now linear in the conjunctive variables $I\{x_{e,i} = 1 \wedge y_{s,i} = 1\}$ and $I\{y_{s,i-1} = 1 \wedge y_{s',i} = 1\}$. As [Collins, 2002] notes, we can then solve equation (3.2) efficiently, as usual, with the Viterbi algorithm [Rabiner, 1989]. We can even have our learning algorithm set $w_{e,s} = \log(P(e|s))$ and $w_{s,s'} = \log(P(s|s'))$ if we wish to faithfully recreate an HMM.

3.3 Multivariate Models

In recent years, NLP systems in particular have moved away from models of single output variables to incorporate many decisions simultaneously. These multivariate models are sometimes also called *structured*. Their complexity comes not only from an expanded output space, but also because the size of that output space at inference-time is, in general, a function of the input. As such, it makes little sense to invent new learned parameters with each new decision to be made. Such a model would be utterly unprepared for an input larger than any it had been trained on previously. Furthermore, depending on the structure being modeled, it may not be clear which learned parameters should impact the behavior of which output variables. It makes much more sense to compose the multivariate model from smaller, more tractably learnable ones in an attempt to detect local patterns that apply repeatedly across the structure.

Thus, many researchers now use classical models as building blocks for the composition of their systems. They also use constraints to encode structural relationships between these building blocks as well as prior knowledge about their global behavior. Finally, they frequently infuse further knowledge into the system by controlling the behavior of the inference algorithm. CCMs can accommodate all of these modeling techniques.

3.3.1 HMMs Revisited

The observant reader will have noticed that the Hidden Markov Models discussed in Section 3.2.3, while classical, are also multivariate. We do count them as a prime example of a joint, multivariate model being composed from smaller, more tractably learnable ones. In particular, note how the number of output variables at inference-time is a function of the number of elements in an input sequence. Instead of associating each successive element with fresh learnable parameters in the model, the HMM is composed of smaller models (the transition and emission tables, namely) which are applied to each pair of consecutive elements in the sequence. These sub-models govern a constant number of output variables, and training them can be quite efficient since we need not consider full sequences when doing so. CCMs permit this training strategy but do not require it; a joint learning algorithm such as [Collins, 2002] can train the same models.

3.3.2 A Discriminative Example

A prime example of the CCM modeling philosophy is the semantic role labeling (SRL) system of [Punyakanok et al., 2008]. In SRL, the input is a sentence of natural language text. The sentence must be segmented into non-overlapping phrases representing the arguments of a given verb in the sentence. These phrases must be identified and classified according to their types. While a solution to this problem could be learned in a joint probabilistic framework, Punyakanok, et al. composed it from two independently learned components and hard constraints encoding expert and structural knowledge enforced only at inference-time. They showed that this composition of simpler models resulted in more efficient learning requiring less training data as well as a fast inference strategy. We now discuss the realization of this system as a CCM.

Composition: The Punyakanok, et al. system consisted of two independently learned classifiers. They learned one LTU to act as an argument candidate filter and one multi-class classifier to predict argument types. Both classifiers classify a single argument candidate and were trained with input features $\mathbf{x}^F$ (F for filter) and $\mathbf{x}^T$ (T for type), respectively. The filter predicts either `yes` or `no` to indicate whether or not the argument candidate should be considered an argument of the given verb. The type classifier selects a prediction from $\mathcal{T} \cup \{\texttt{null}\}$ where $\mathcal{T}$ is the set of argument types (e.g. `A0`, `A1`, `A2`, ...), and `null` indicates the candidate argument is not actually an argument. So, the CCM will include $2 + |\mathcal{T}|$ Boolean labels, y^F_a and $y^T_{a,t}$ for each argument candidate a and type t, and constraints to ensure a single $y^T_{a,t}$ is active for any given a. If there are a total of $\mathcal{A}$ candidate arguments, these features and constraints can be written as follows:

$$\phi^F_i(\mathbf{x}, \mathbf{y}) = \sum_{a=1}^{\mathcal{A}} y^F_a \, x^F_i \tag{3.14}$$

$$\phi^T_{t,i}(\mathbf{x}, \mathbf{y}) = \sum_{a=1}^{\mathcal{A}} y^T_{a,t} \, x^T_i \tag{3.15}$$

$$c^D_a(\mathbf{x}, \mathbf{y}) = I \left\{ \sum_{t}^{\mathcal{T} \cup \{\texttt{null}\}} y^T_{a,t} \neq 1 \right\} \tag{3.16}$$

Constraints: If the filter predicts `no`, the type classifier must predict `null`. We will refer to this structural constraint as the *filter constraint*. In addition, there are the structural constraints ensuring that no two arguments overlap as well as knowledge about type regularities encoded in constraints such as

- no two arguments associated with any given verb may have type `A`k, for $k \in \{0, 1, 2, 3, 4, 5\}$, and
- if any argument associated with a verb v has reference type `R-A`k, then some other argument associated with v must have the referent type `A`k, for $k \in \{0, 1, 2, 3, 4, 5\}$.

The filter constraint can be written as:

$$c^F_a(\mathbf{x}, \mathbf{y}) = I \left\{ \neg \left(\neg y^F_a \Rightarrow y^T_{a,\texttt{null}} \right) \right\} \tag{3.17}$$

Constraints that establish a logical relationship between labels can be written to enforce the other structural and domain specific constraints in our SRL problem as well [Punyakanok et al., 2008]. We will return to a more detailed description of these constraints in Section 6.1.

Inference: At inference-time, the predictions of both classifiers compete with each other in a purposefully designed constrained optimization framework. The intention here is that the type classifier should have the last word on which argument candidates will actually be considered arguments. The role of the filter classifier is to trim away the vast majority of candidates that are easily recognized as bad choices so that the type classifier sees a more uniform distribution of argument types both during training and inference. During training, Punyakanok, et al. tuned the filter classifier for high recall so that a prediction of `no` would be very trustworthy. Provisions must then also be made at inference-time to ensure both classifiers have the intended semantics.

First, notice that we have already specified the filter constraint with merely implication and not double implication. The type classifier is thus unconstrained whenever the filter classifier predicts `yes`. We will need additional help, however, to ensure that the filter classifier's high quality `no` predictions propagate to the type classifier. Put another way, we don't want a strong but ill-advised prediction from the type classifier to override a highly accurate filter prediction of `no`. This behavior can be implemented in a CCM by artificially inflating the filter's scores by a constant α.

$$z(\mathbf{x},\mathbf{y}) = \alpha\,\mathbf{w}^F \cdot \Phi^F(\mathbf{x},\mathbf{y}) + \mathbf{w}^T \cdot \Phi^T(\mathbf{x},\mathbf{y}) - \infty\left(\sum_{a=1}^{\mathcal{A}} c_a^F(\mathbf{x},\mathbf{y}) + c_a^D(\mathbf{x},\mathbf{y})\right) \tag{3.18}$$

The model to prefer, in general, global assignments that agree with the filter classifier. Note also that the constraints are all *hard*; ie., if any constraint is violated, the score of the assignment is $-\infty$.

3.3.3 Exponential Family Models

We have already seen an example of an exponential family model expressed as a CCM in Section 3.2.3:

$$P(\mathbf{s},\mathbf{e}) = e^{z(\mathbf{x},\mathbf{y})} \tag{3.19}$$

where $z(\mathbf{x},\mathbf{y})$ is defined by Equation (3.13). By the same token, a Conditional Random Field (CRF) [Lafferty et al., 2001] can be expressed as a CCM by simply taking the log of the joint distribution's factorization. In the case of sequence tagging, the features are identical to Equations

(3.11) and (3.12); the associated weights merely need to be renormalized. Plus, when written as a CCM, long range constraints that are difficult to express in a probabilistic model can become tractable [Roth and Yih, 2005]. Furthermore, since CRFs are a generalization of Markov networks, the Markov Logic Network discussed in Section 2.2.2 can also easily be written as a CCM and will enjoy the same benefits.

Chapter 4

Learning Based Java

In this chapter, we describe Learning Based Java (LBJ), our first generation modeling language for describing CCMs. LBJ is not a general purpose programming language; instead, its aim is to facilitate the implementation of learned *classifiers*, perhaps constrained, that can be called just like normal Java methods in an external Java program. LBJ incorporates high level primitives for feature extraction, learning, and ILP inference under the assumptions that features involve a single output variable each and constraints are enforced only at inference-time. It supports model composability and inference decomposability, and admits both discriminative and probabilistic algorithms. In this chapter, we review the syntax and semantics of the LBJ language (Section 4.2), discuss implementation details (Section 4.3), and present an efficient algorithm for converting arbitrary FOL sentences into linear inequalities suitable for ILP (Section 4.5).

4.1 Scope

An LBJ program can be thought of as a specification for one or more CCMs of the following restricted form:

$$z_{\mathbf{w},\rho}(\mathbf{x},\mathbf{y}) = \left(\sum_{y \in \mathbf{y}} \sum_{t}^{\mathcal{T}_y} \sigma\left(\mathbf{w}_{y,t}, \Phi_y(\mathbf{x})\right) y_t \right) - \infty \sum_{j}^{n_c} c_j(\mathbf{x},\mathbf{y}) \tag{4.1}$$

$$\sigma(\mathbf{w},\mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x}) \tag{4.2}$$

where, with a slight abuse of notation, the y_t variables are grouped within $\mathbf{y}$ when they collectively represent a multi-class classification, $y_t \in \{0,1\}$, and g is any (usually monotonic) function[1]. Re-

[1] We also define $\infty \cdot 0 = 0$

garding the role of σ, the idea is that the dot products associated with each potential value of each discrete output variable (or some function g of them) can be precomputed, making the objective function shorter and simpler. The selection of g is LBJ's only mechanism for providing direct access to the inference objective function.

In this restricted view of the CCM, Boolean and discrete output variables are still supported, but all relationships between output variables are specified via constraints, since features are no longer functions of the output. In addition, all constraints are hard, as indicated by the coefficient of ∞ on the constraint term in Equation (4.1). So, while the HMM described in Section 3.2.3 cannot be described in LBJ, the multivariate SRL model in Section 3.3.2 can and has been implemented.

4.2 Syntax and Semantics

An LBJ program is a list of declarations that describe *classifiers*, *constraints*, and *inference problems*. A classifier is a function that takes a single "example" object having any Java type as input and returns any number of *features* as output. Learned classifiers are restricted to return a single feature, but they are also capable of producing a score for each possible value that feature might take. Constraints are defined in terms of classifiers applied to example objects, and inference problems collect the scores of classifiers used in constraints to form an objective function. All of these ideas are described in more detail in this section.

When applied to an LBJ source file, the LBJ compiler generates a Java class corresponding to each classifier, constraint, and inference problem. The code generated in these classes implements the semantics of the specified syntactic structures with the help of the LBJ run-time library. The run-time library contains implementations of inference and learning algorithms as well as frequently used internal representations and feature extracting classifiers (implemented in LBJ) for NLP tasks. It is our hope to expand this library to other domains in the future.

4.2.1 Features

Features in LBJ are values produced by classifiers that represent properties of example objects. They are used under the hood in LBJ to communicate with learning algorithms. The programmer has some control over how they are created and used, but they also have several implicit behaviors that are beyond the programmer's reach. But before discussing how they are created and used, we first enumerate their different varieties.

Feature Varieties

First of all, a feature can be either *discrete* or *real.* A discrete feature's value is represented as a string that comes from an unordered set which may or may not be described in the LBJ source code, whereas a real feature's value is represented as a floating point value. Whether a feature is discrete or real, it can be either *primitive*, *conjunctive*, or *referring.* A primitive feature has a name and a value. A conjunctive feature has a pointer to a feature on the left and another to a feature on the right (the order matters). Finally, a referring feature has a name and a pointer to another feature. Every feature, no matter the variety, is also distinguished by the name of the classifier that produced it, which is kept internally in a string.

Throughout this thesis, when we discuss the CCM formalism and the implementation of learning based programs, we often describe features in the context of vector spaces. This is largely because of our preference for learning and inference algorithms that operate in these spaces. However, LBJ's features, being either discrete or real, can be considered more general than that. While learning algorithms that want vector spaces are free to treat each value of a given discrete feature as a separate dimension, this behavior is not required by the language. A decision tree learner, for example, is free to keep the discrete feature in one piece.

Primitive Features A primitive feature's name is represented as a string, while its value is either a string if the feature is discrete or a double-precision floating point value if the feature is real. A discrete primitive feature may also internally identify itself as Boolean. In this case, its value is restricted to be either `"true"` or `"false"`. Furthermore, learning algorithms that take such a feature as input can choose to handle it specially. A vector space algorithm that normally allocates a separate dimension for each value of a discrete feature may choose to allocate only a single dimension for a Boolean feature. Primitive features are the only features that can appear as literals in an LBJ source code.

Conjunctive Features Conjunctive features do not have names. They are distinguished by the "child" features to which they refer. The conjunction of two discrete features is itself a discrete feature which indicates that both of the indicated properties are present in the input. Two real features may also be combined in a real, conjunctive feature whose value is the product of the child features' values. Finally, a discrete and a real feature may be conjuncted into a real, conjunctive feature whose value is the value of the real child feature. Doing so can be thought of simply adding extra distinguishing information to the real feature in much the same way as a referring feature.

Conjunctive features come in quite handy to increase the expressivity of a linear classifier. One may wonder why disjunctive and negated features are not also provided for. In short, the reason is that we have no way to automatically ensure their correct semantics in the infinite feature space [Blum, 1992]. For a more thorough discussion of this issue, see [Cumby and Roth, 2003].

Referring Features A referring feature contains a string name and a pointer to a child feature. The referring feature is discrete if the child feature is discrete or real if the child feature is real. As mentioned above, a referring feature is created to add extra distinguishing information to an existing feature. LBJ provides this facility for the same reason that it provides the facility for conjunctive features: to save memory in case the features in use by some classifier contain redundant information. This is frequently the case in domains such as NLP that contain a lot of repeated structure. For example, a classifier may need to know whether a word is capitalized, whether it contains a common prefix or suffix, whether it is a number, etc., and furthermore, it may need to know this information about every word in a sentence. When words are similar, these features will be identical, and we may then want to distinguish them by, e.g., their location in the sentence. The referring feature serves this purpose.

Using Features

As we will see in Section 4.2.2, a classifier's declaration indicates either that it returns discrete or real features. Nothing more specific is ever indicated in the code. Furthermore, while a feature should be thought of as a value, it can't be stored in a local variable in a hard-coded classifier or constraint as other values can.[2] Instead, features can only be created and returned by classifiers. The exact mechanism for doing so depends on the classifier; see Section 4.2.2 where we revisit this issue after describing the different varieties of classifiers.

4.2.2 Classifier Declarations

In LBJ, a classifier can be defined with Java code or composed from the definitions of other classifiers using special operators. As such, the syntax of classifier specification allows the programmer to treat classifiers as expressions and assign them to names. This section defines the syntax of classifier specification more precisely, including the syntax of classifiers learned from data. It also details the behavior of the LBJ compiler when classifiers are specified in terms of training data and when changes are made to an LBJ source file.

Classifier declarations are used to name classifier expressions. The syntax of a classifier declaration has the following form:

```
feature-type name(type name) [cached | cachedin field-access] <-
    classifier-expression
```

A classifier declaration names a classifier and specifies its input and output types in its header, which is similar to a Java method header. Either the `cached` or the `cachedin` modifier may appear after the input argument, though they are both optional. The declaration ends with a left arrow indicating assignment and a classifier expression which is assigned to the named classifier.

Semantically, every named classifier is a static method. In an LBJ source file, references to classifiers are manipulated and passed to other syntactic constructs, similarly to a functional programming language. The LBJ compiler implements this behavior by storing a classifier's definition in a static method of a Java class of the same name and providing access to that method through objects of that class. As we will see, learning classifiers are capable of modifying their definition, and by the semantics of classifier declarations, these modifications are local to the currently executing process, but not to any particular object. In other words, when the application continues to train a learning classifier on-line, the changes are immediately visible through every object of the classifier's

[2]Actually, it can, but it's inconvenient; the programmer would have to manually take it apart to make use of it, even to merely return it.

class.

Classifier Return Types

In total, there are 5 possible classifier declaration return types: `discrete`, `discrete%`, `real`, `real%`, and `mixed%`. The return type gives the following two pieces of information: *a*) whether the features returned by this classifier will be discrete, real, or could be either (`mixed`), and *b*) whether the classifier is capable of returning multiple features. The percent sign (`%`) in three of the possible return types indicates that the classifier is capable of returning multiple features. When this is the case, we may also refer to the classifier as a *feature generator*. When it is not, the classifier is guaranteed to return a single feature. Finally, the classifier expression after the left arrow in the classifier declaration must have a type that is at least as specific as the return type of the declaration. By definition, the `discrete` (`real`, respectively) type is more specific than the `discrete%` (`real%`) type, and both `discrete%` and `real%` are more specific than `mixed%`.

Caching

The optional `cached` and `cachedin` keywords are used to indicate that the result of a non-generator's computation will be cached in association with the input object. The `cachedin` keyword instructs the classifier to cache its output in the specified field of the input object. For example, if the parameter of the classifier is specified as `Word w`, then `w.partOfSpeech` may appear as the *`field-access`*. Alternatively, the `cached` keyword instructs the classifier to store the output values it computes in a hash table. A cached classifier (using either type of caching) will first check the specified location to see if a value has already been computed for the given example object. If it has, it is simply returned. Otherwise, the classifier computes the value and stores it in the location before returning it.

For learned classifiers that reuse previously predicted values as input features, caching can have a big impact on performance. Consider, for example, a sequence tagging classifier that uses as features the previous two predictions it made. Without caching, as the length of the sequence grows, the number of recursive calls made by this classifier grows like the Fibonacci sequence (i.e., exponentially). LBJ allows the programmer to reduce this sad state of affairs to a linear time execution with the addition of a single keyword.

In addition, learning classifiers cached with either keyword will not load their (potentially large) internal representations from disk until necessary (i.e., until an object is encountered whose cache location is not filled in). Thus, if the programmer has precomputed the predictions of this classifier as he may want to do when using it to provide features to another learned classifier, no time or space is wasted in passing this data down the pipeline.

Classifier Expressions

An LBJ classifier expression is one of the following syntactic constructs:

- a classifier name
- a method body (i.e., a list of Java statements in between curly braces)
- a conjunction of two classifier expressions
- a comma separated list of classifier expressions
- an inference invocation expression
- a learning classifier expression

Each of these options is discussed in turn in this section except for the learning classifier expression, which is deemed important enough for its own sub-subsection (4.2.3). Evaluation of a classifier expression (done at compile-time) results in an anonymous classifier, similar to a functional programming language. Anonymous classifiers can only receive a name in a classifier declaration.

Classifier Names The name of a classifier defined either externally or in the same source file may appear wherever a classifier expression is expected. If the named classifier's declaration is found in the same source file, it may occur anywhere in that source file (in other words, a classifier need not be defined before it is used). If the named classifier has an external declaration it must either be fully qualified (e.g., `myPackage.myClassifier`) or it must be imported by an import declaration at the top of the source file. The class file or Java source file containing the implementation of an imported classifier must exist prior to running the LBJ compiler on the source file that imports it.

Method Bodies A method body is a list of Java statements enclosed in curly braces explicitly implementing a classifier. When the classifier implemented by the method body returns a single feature, the familiar `return` statement is used to provide that feature's value. At that point, a primitive feature is implicitly instantiated and returned, and the name of the classifier serves as the name of the feature. If the feature return type is `real`, then the `return` statement's expression must evaluate to a `double`. Otherwise, it can evaluate to anything - even an object - and the resulting value will be converted to a string. Each method body takes its argument and feature return type from the header of the classifier declaration it is contained in.

The `sense` statement is used to instantiate a primitive feature that has been detected while executing a feature generator. In these contexts, any number of features may be sensed, and they are all returned in the order in which they were sensed upon completion of the method body.

The syntax of a `sense` statement is

`sense` *expression* `:` *expression* `;`

The first expression may evaluate to anything. After converting it to a string, the value serves as the name of the primitive feature. The second expression will be interpreted as the feature's value. It must evaluate to a `double` if the method body's return type is `real%`. Otherwise, it can evaluate to anything and the resulting value will be converted to a string. A `sense` statement may also appear with merely a single expression argument. In this case, the expression represents the feature's name, and that feature is assumed to be Boolean with a value of `true`.

Classifier Conjunction A classifier conjunction is written with the double ampersand operator (`&&`) in between two classifier expressions. The conjunction of two classifiers that each return a single feature is a classifier producing a conjunctive feature that points to the features produced by the conjunction's operands. When either of the operands is a generator, the result is a classifier returning all the conjunctive features in the cross product between the sets of features returned by the operands.

Composite Generators "Composite generator" is LBJ terminology for a comma separated list of classifier expressions. When classifier expressions are listed separated by commas, the result is a feature generator that simply returns all the features returned by each classifier in the list.

Inference Invocations Finally, an *inference invocation* looks similar to a Java method call and produces a classifier that obeys the constraints (Section 4.2.4) in an inference problem (Section 4.2.5):

```
inference-problem-name ( classifier-exp )
```

The *classifier-exp* will always be merely a name representing a classifier that participates in the constraints.

Invoking Classifiers

Under the right circumstances, any classifier may be invoked inside an LBJ method body just as if it were a method. The syntax of a classifier invocation is simply `name(object)`, where `object` is the object to be classified and `name` follows the same rules as when a classifier is named in a classifier expression (see the section on classifier names above). In general, the semantics of such an invocation are such that the value(s) and not the names of the produced features are returned at the call site.

More specifically:

- A classifier defined to return exactly one feature may be invoked anywhere within a method body. If it has feature return type `discrete`, a `String` will be returned at the call site. Otherwise, a `double` will be returned.

- Feature generators may only be invoked when that invocation is the entire expression on the right of the colon in a `sense` statement contained in another feature generator of the same return type[3]. In this case, the `sense` statement will instantiate for each feature returned by the generator a new referring feature whose name is the string value on the left side of the colon

[3] Any feature generator may be invoked in this context in a classifier whose feature return type is `mixed%`.

and which points to the returned feature. Thus, an entire set of features can be translated to describe a different context with a single `sense` statement.

4.2.3 Learning Classifier Expressions

Classifiers defined by data but that act the same as classifiers defined in more traditional ways are, of course, a central point of focus in learning based programs. They must be easy for the programmer to specify when he wants abstraction, yet expose enough implementation detail to be useful to a machine learning expert. In LBJ, learning classifier expressions have the following syntax:

```
learn [classifier-expression]                                  // Labeler
  using classifier-expression                                  // Feature extractors
  [from instance-creation-expression [int]]                    // Parser
  [with instance-creation-expression]                          // Learning algorithm
  [prune scope prune-strategy threshold]                       // Feature pruning
  [cval [int] split-strategy]                                  // K-Fold Cross Validation
  [normalizedby instance-creation-expression]                  // For the objective function
end
```

The first classifier expression represents a classifier that will provide label features for a supervised learning algorithm. The classifier expression in the `using` clause does all the feature extraction on each object, during both training and evaluation. It will often be a composite generator.

The `from` Clause

The instance creation expression in the `from` clause should create an object of a class that implements the `LBJ2.parser.Parser` interface in LBJ's run-time library. This clause is optional. If it appears, the LBJ compiler will automatically perform training on the learner represented by this learning classifier expression at compile-time. Whether it appears or not, the programmer may continue training the learner on-line in the application via methods defined in `LBJ2.learn.Learner`, also in the library.

When the `from` clause appears, the LBJ compiler trains the specified learning classifier at compile-time. First, it retrieves objects from the specified parser until it finally returns `null`. One at a time, the feature extraction classifier in the `using` clause is applied to each object, and the results are sent to the learning algorithm for processing. In addition, since many learning algorithms perform much better after being given multiple opportunities to learn from each training object, we also provide an integer addendum to this clause. The integer specifies a number of *rounds*, or the number of passes over the training data to be performed by the classifier during training.

The `with` Clause

The instance creation expression in the `with` clause should create an object of a class derived from the `LBJ2.learn.Learner` class in the library. This clause is also optional. If it appears, the generated Java class implementing this learning classifier will be derived from the class named in the `with` clause. Otherwise, a default learner selected by the system for the declared return type of this learning classifier will be substituted with default parameter settings.

The `prune` Clause

When the optional `prune` clause is present, the classifier employs one of the following heuristics to prune features from a feature extracted dataset before training begins. First, the number of occurrences of each feature encountered during feature extraction is counted internally by the classifier. Then, if the "prune-strategy" argument is `"count"`, the "threshold" argument will be interpreted as the minimum quantity of occurrences that a feature must amass lest it be pruned away. Otherwise, if the "prune-strategy" argument is `"percent"`, the decision to prune a feature will be made relative the most frequently occurring feature. The "threshold" argument will be interpreted as the minimum fraction of the most frequently occurring feature's occurrences that a feature must amass lest it be pruned away. Finally, the "scope" argument can be either `"global"` or `"perClass"`. When the classifier is multi-class, this argument controls whether the aforementioned heuristics are carried out across the entire dataset or within the subset of the data labeled by each possible prediction value individually. If the classifier is not multi-class, the "scope" argument must be `"global"`.

The `cval` Clause

The `cval` clause enables LBJ's built-in K-fold cross validation system. K-fold cross validation is a statistical technique for assessing the performance of a learned classifier. First the user's training data is partitioned into K subsets such that a single subset is held aside for testing while the others are used for training. For each potential testing set, the classifier is trained from scratch on the remaining partitions, and its testing performance is recorded. After all partitions are processed in this way, we now have K samples of our classifiers performance on unseen testing data over which statistics can be taken. LBJ automates this process in order to alleviate the need for the user to perform his own testing methodologies. The optional `split-strategy` argument to the `cval` clause can be used to specify the method with which LBJ will split the data set into subsets (folds).

The `normalizedby` Clause

The `normalizedby` clause specifies a function g for use in Equation (4.2). The LBJ library contains a set of functions that may be named here; e.g., sigmoid, softmax, and log.

Parameter Tuning

In addition to all of the clauses mentioned above, learning classifier expressions also support some extra syntactic sugar that utilizes K-fold cross validation to tune parameters' values. In this context, the parameters to which we are referring are those specified by the programmer to control how learning operates as opposed to the learned parameters in a model. These are parameters such as the number of training rounds, the arguments of the `prune` clause, and any arguments passed the the learning algorithm's constructor in the `with` clause.

The extra syntax specifies a set of values in between double curly braces ('`{{`' and '`}}`'). Inside the braces, the programmer may write either a comma separated list of literal values (strings, integers, doubles, or what-have-you) or a specification of linearly interpolated integers or doubles between given starting and ending values and separated by a given quantity (e.g., '`{{ .1 -> .5 : .1 }}`'). When this syntax is present in a learning classifier expression, the LBJ compiler will use K-fold cross validation as specified by the `cval` clause to assess the performance of the classifier given every combination of parameter values from the cross product of the sets specified with the new syntax. The parameter assignment that results in the best performance is then used to retrain the classifier over the entire dataset.

4.2.4 Constraints

LBJ constraints are written as declarative, first order logic (FOL) sentences interspersed within a pure Java method body. This way, general constraints may be expressed in terms of data whose exact shape is not known until run-time. In these sentences, classifiers and arbitrary Java expressions act as FOL functions, the domain of discourse is the set of all Java objects, and there are exactly two predicates: equality and inequality. These predicates accept any objects as arguments, but they in fact denote string comparison; Java's `toString()` method is invoked to do the conversion. Also, the usual operators and quantifiers are provided, as well as the `atleast` and `atmost` quantifiers, which are described below. The interpretations of Java expressions, of course, are fixed. With all this in place, the inference algorithm will search for interpretations of the classifiers that satisfy the constraints.

Syntactically, an LBJ *constraint declaration* starts with a header indicating the name of the constraint and the type of object it takes as input, similar to a method declaration with a single parameter. The body of the constraint may then contain arbitrary Java code interspersed with declarative constraint statements. At run-time, when a constraint is invoked, the conjunction of all constraint statements encountered during its execution is returned as the result. Thus, the constraint declaration is semantically a method that creates run-time constraints given an object representing an *inference problem* (Section 4.2.5) as input.

Each constraint statement contains a single constraint expression which takes one of the following forms:

- An equality predicate ϵ_1 `::` ϵ_2 where ϵ_i is an arbitrary Java expression or the application of an LBJ classifier to an object.
- An inequality predicate ϵ_1 `!:` ϵ_2 .
- The negation of an LBJ constraint `!`γ .

- The conjunction of two LBJ constraints γ_1 `/\` γ_2 .
- The disjunction of two LBJ constraints γ_1 `\/` γ_2 .
- An implication γ_1 `=>` γ_2 .
- The equivalence of two LBJ constraints γ_1 `<=>` γ_2 .
- A universal quantifier `forall` (ρ `in` ϵ) γ where ρ is a formal parameter with type τ and identifier ι, ϵ is a Java expression evaluating to a Java `Collection` containing objects of type τ, and γ is an arbitrary constraint expression which may be written in terms of ι.
- An existential quantifier `exists` (ρ `in` ϵ) γ .
- An *at least quantifier* `atleast` ϵ_1 `of` (ρ `in` ϵ_2) γ where ϵ_1 is a Java expression evaluating to an `int` and the other parameters play similar roles to those in the universal quantifier.
- An *at most quantifier* `atmost` ϵ_1 `of` (ρ `in` ϵ_2) γ .
- A constraint invocation `@` $\nu(\epsilon)$ where ν is the constraint's name and ϵ is an arbitrary Java expression.

The `atleast` and `atmost` quantifiers do not add expressivity to FOL. However, an equivalent Boolean expression involving only the conjunction and disjunction operators will contain $\binom{m}{\epsilon_1}$ terms, where m is the size of the `Collection` represented by ϵ_2. Fortunately, as we will see in Section 4.5, these constraints can be translated to a constant number of linear inequalities for ILP inference.

4.2.5 Inference Problems

From an FOL perspective, an LBJ *inference problem* is a model, specifying the domain of discourse by providing example objects to the constraints, but lacking interpretations for the classifiers. From the CCM perspective, an inference problem must fully specify Equation (3.1). To accomplish this, the programmer specifies an inference problem template in which constraints are applied to a "head" object. The head object can be thought of as an encapsulation of the inference problem's domain of discourse; it must contain (or have references to) all example objects involved. At run-time, LBJ will instantiate the constraints on the head object, thereby discovering all applications of classifiers to example objects. The CCM template in Equation (4.1) is extended by including a new $y \in \mathbf{y}$ for each unique pair (φ, ε) discovered where φ is a classifier and ε is an example object. A new implicitly defined constraint of the form in Equation (3.5) is also added to the template for each such pair, along with the explicitly specified constraints, to form our new objective function.

Inference problem templates have the following syntax:

```
inference name head type name {
      [type name method-body]+
      subjectto method-body
      with instance-creation-expression
}
```

The header names the inference problem template and declares the head object. The body then contains the following three elements. First, it contains a list of "head finder" methods used to locate the head object given an example object from the argument of a classifier application in a constraint. Whenever the programmer wishes to apply the inference to produce the constrained version of a participating classifier, that classifier's input type must have a head finder method in the inference body. Second, the `subjectto` clause is actually a constraint declaration within which all relevant constraints should be specified or invoked. Finally, the `with` clause specifies which inference algorithm to use.

4.3 Feature Extraction

4.3.1 Sparse Representation

In general, LBJ learning classifiers do not assume anything a priori about how many features they will encounter or what types those features will have. As a result, learning algorithms implemented for LBJ need to work with sparse representations of feature and weight vectors. As new features are encountered, a learning algorithm will allocate additional space as appropriate for the internal representation of its hypothesis.

4.3.2 Automatic Indexing

For learning algorithms that represent their hypothesis as a weight vector, LBJ internally converts every encountered feature into an integer index and an associated *strength*. A strength is just the weight of a feature in an example. Thus, examples are represented by an array of indexes and an array of strengths. The learning algorithm records a *lexicon* mapping features to indexes. This way, the weight vector can also be implemented as an array, making dot products as efficient as possible. This automatic feature indexing is most beneficial when the algorithm requires multiple passes over the data, as many algorithms do.

In domains such as NLP that require the infinite feature space, the lexicon can become very large very quickly. Thus, efficiency in both time and space is an important issue. Furthermore, the integer indexes returned by the lexicon must be packed tightly so that the classifiers' weight vectors are only as big as they need to be. LBJ's lexicon utilizes its feature representation to conserve space and time. Adding a conjunctive or referring feature to the lexicon involves looking up and potentially adding their children as a by-product. All the while, we must be careful: child features added to the lexicon indirectly in this way should not be assigned an index unless also added directly as a parent feature at some other time.

This is all further complicated by the pruning process which could remove any feature from the entire dataset. When this happens, we'd like to free up the index associated with the feature in the lexicon to save space in classifiers' weight vectors as well. This can be a big help if many features are pruned. However, the programmer may also benefit from the option to cache features (and counts) pruned from both the lexicon and the dataset on disk in case he wants to change the parameters of the `prune` clause in the learning classifier expression. That would save the time of re-extracting the features. Thus, the learning classifier's pruning operation is implemented to rearrange the storage of features in the lexicon and dataset in such a way that it saves space and is prepared for the programmer's next experiment.

When the lexicon is written to disk, space and time is also conserved by writing merely integers that point to referring features instead of the features themselves. The models and lexicons themselves are only read from disk when absolutely necessary, and during training, the lexicon is expunged from memory after feature extraction completes. All of this is completely transparent to the programmer who simply passes arbitrary Java objects to his classifiers.

4.4 Development Cycle

An LBJ source file also functions as a makefile in the sense that the operations it performs (e.g., code generation, feature extraction, learning, etc.) are only executed if they are deemed necessary to bring the compiled system up to date with the source code. To implement this behavior, the generated source code of every classifier, constraint, and inference problem stores in a comment a compressed representation of its LBJ specification. When the compiler runs again, this representation is compared against the current version of the code. This tells the compiler which declarations have been explicitly revised.

Next, we implemented a data-flow analysis [Kildall, 1973] to determine which additional classifiers, constraints, and inference problems have been affected by the detected revisions. To do this, we first create a control-flow graph whose nodes are declarations and whose directed edges represent dependencies between the declared elements. Edges are created when when a classifier or constraint is invoked by another classifier or constraint in a method body and when a classifier is defined as a combination of other classifiers via a classifier expression. The data-flow analysis then propagates the information indicating which classifiers have been revised along the edges in the graph so that each declaration may respond appropriately.

In addition, each learning classifier expression is represented by several nodes in this graph; one for each of the operations it may perform. When another classifier points to a learning classifier expression, it actually points to its feature extraction node. The feature extraction node points to the pruning node, which points to the training node, which points finally to the parameter tuning node. When the learning classifier expression points to another declaration in the graph, it is in fact the parameter tuning that does the pointing.

All of this is done to streamline the programmer's development cycle. If there are several learned classifiers in an LBJ source file and a feature extraction classifier is changed, only those learned classifiers that depend on it will be retrained. If the change made to a learning classifier expression is merely a new value for a learning algorithm parameter, we can skip past the expensive feature extraction and pruning operations and head straight for training. In fact, feature extraction can be skipped even when changing the pruning parameters, since we went to the trouble of caching pruned features on disk as described in the previous section.

4.5 Inference with Integer Linear Programming

Once constraints are specified, they can serve as input to a variety of optimization algorithms, frequently based on search; e.g. beam search and A^* search. An alternative to these techniques that is increasingly popular in the literature is ILP. This popularity comes despite a constraint representation, the linear inequality, that is non-intuitive and tricky to encode knowledge in. LBJ offers a practical solution by translating the more intuitively represented FOL constraints to linear inequalities automatically at run-time.

First, the constraint specified in the `subjectto` clause of the inference problem becomes instantiated by the head object. It is then *propositionalized*, meaning that all quantifiers are "unrolled" into flat, propositional sentences. From there, the conversion to linear inequalities could be accomplished naïvely in a conceptually simple way by first converting the propositionalized FOL sentence to conjunctive normal form (CNF) and then generating a single linear inequality of the form $\sum_i b_i \geq 1$ for each disjunctive clause in the CNF. Since an ILP must satisfy all its linear inequalities, the entire CNF will be satisfied.

Unfortunately, conversion to CNF is NP-hard, and the reason is that it may produce an exponential number of clauses in the resulting form. So instead of converting to CNF, we present here an algorithm that translates the original form directly while creating auxiliary ILP variables along the way that are constrained to represent the values of subexpressions. The algorithm is based on a well-known idea from the SAT-solving community [Plaisted and Greenbaum, 1986, Thiffault et al., 2004].

4.5.1 Propositionalization

When propositionalizing the constraint, there are a few issues to consider. First, the `atleast` ϵ_1 `of` (ρ `in` ϵ_2) and `atmost` ϵ_1 `of` (ρ `in` ϵ_2) quantifiers are given the propositional analogs `atleast` m `of` β and `atmost` m `of` β, where m is an integer and β is a list of propositional Boolean expressions. The only alternative would be to generate a Boolean expression involving only conjunction and disjunction of size potentially exponential in the size of the original sentence.

Second, we consider the equality and inequality predicates. Their arguments are arbitrary Java expressions and applications of classifiers to arbitrary Java expressions. Once instantiated, each argument becomes either a string or a classifier application (φ, ε) where φ is a classifier and ε is an example object. The comparison of two strings propositionalizes to a constant. The comparison of a classifier application and a string t becomes a Boolean propositional variable $b_t^{(\varphi,\varepsilon)}$. Note that if $t \notin \mathcal{T}$ where $\mathcal{T}$ is the range of the classifier, we can immediately replace $b_t^{(\varphi,\varepsilon)}$ with `false`. Finally, the comparison of two classifier applications becomes an expression of the following form: $\bigvee_{t \in \mathcal{T}_1 \cap \mathcal{T}_2} b_t^{(\varphi_1,\varepsilon_1)} \wedge b_t^{(\varphi_2,\varepsilon_2)}$. Here, the propositionalization process benefits from knowledge of the ranges of the two classifiers; only those classes t shared by their ranges need be included in this expression.

Finally, we must consider the form that our final propositionalized constraint will take. Motivated by the algorithm described in Section 4.5.2, we simplify our propositionalized constraint representation in the following ways. Equivalences and implications are expanded. Conjunctions containing conjunctive terms are recursively flattened into an n-ary conjunction representation. A similar flattening is applied to disjunctions. Negation operators are moved inside of connective expressions using DeMorgan's law until the only negation operators left are applied directly to propositional variables. Other obvious simplifications, such as $b \wedge$ `true` $\equiv b$ are also applied.

4.5.2 Generating Linear Inequalities

The algorithm that generates linear equalities is given in Figure 4.1. It is a recursive algorithm that traverses the syntax tree of a propositional constraint expression looking for conjunctions, disjunctions, `atleast`, and `atmost` expressions that contain only *literals* (i.e., a propositional variable that may or may not be negated). These are the base cases of the recursion, and each one corresponds to either one or two linear inequalities depending on whether it represents a *top* or *lower* level constraint (defined below), respectively. If there are λ literals and μ Boolean operators in the input constraint, the running time of the algorithm is $O(\lambda + \mu)$.

First, consider the form of the simplified, propositional constraint. The top of its syntax tree is a conjunction. Its children, the top level constraints, are all either literals, disjunctions, `atleast`, or `atmost` expressions. Literals at this top level are translated to linear constraints trivially. Disjunctions, `atleast`, and `atmost` expressions at the top level can also be directly translated to a single linear inequality.

At lower levels of the constraint's syntax tree, our base case expressions have different semantics than they did at the top level. They no longer describe properties of the variables that *must* be enforced at all times; instead, they describe some property of the variables that the parent expression is taking into consideration. For a positive literal, this property is the value of the propositional variable itself. For a negative literal, it's the opposite of the variable's value. In these cases, TranslateToILP returns an *ILP literal* (either y or $1 - y$, where $y \in \{0, 1\}$ is an ILP variable) for use in generating inequalities for the parent expression. For all other base case expressions, TranslateToILP creates a new ILP variable $q \in \{0, 1\}$ to represent the input subexpression's value. Two linear inequalities constrain q to take the value 1 if and only if the subexpression it represents is `true` with respect to its children. TranslateToILP then returns q so that the parent expression can substitute q for this subexpression.

When a conjunction, disjunction, `atleast` m, or `atmost` m expression β of literals appears as a subexpression, the new ILP variable q is defined as in Equations (4.3), (4.4), (4.5), and (4.6), respectively. In these inequalities, $\mathcal{L}$ is the set of ILP literals representing the subexpressions of the expression we are translating, and m is a constant. The inequality on the left of each group enforces a constraint of the form $q = 1 \Rightarrow \beta$. It does not impose any restriction on the variables in $\mathcal{L}$ when $q = 0$. Similarly, the inequality on the right enforces a constraint of the form $q = 0 \Rightarrow \neg\beta$ and does not impose any restriction on the variables in $\mathcal{L}$ when $q = 1$. Thus, with both inequalities enforced by an ILP solver, we have $q = 1 \Leftrightarrow \beta$.

$$\left(\sum_{l}^{\mathcal{L}} l\right) - |\mathcal{L}|q \geq 0 \qquad \left(\sum_{l}^{\mathcal{L}} l\right) - q \leq |\mathcal{L}| - 1 \quad \text{defines } q \text{ as conjunction} \tag{4.3}$$

$$\left(\sum_{l}^{\mathcal{L}} l\right) - q \geq 0 \qquad \left(\sum_{l}^{\mathcal{L}} l\right) - |\mathcal{L}|q \leq 0 \quad \text{defines } q \text{ as disjunction} \tag{4.4}$$

$$\left(\sum_{l}^{\mathcal{L}} l\right) - mq \geq 0 \qquad \left(\sum_{l}^{\mathcal{L}} l\right) - |\mathcal{L}|q \leq m - 1 \quad \text{defines } q \text{ as } \texttt{atleast}\ m \tag{4.5}$$

$$\left(\sum_{l}^{\mathcal{L}} l\right) + (|\mathcal{L}| - m)q \leq |\mathcal{L}| \qquad \left(\sum_{l}^{\mathcal{L}} l\right) + (m+1)q \geq m + 1 \quad \text{defines } q \text{ as } \texttt{atmost}\ m \tag{4.6}$$

4.6 Efficiency Improvements

In general, integer linear programming is NP-hard. There is, however, a well known class of ILP problems for which efficient linear program solvers will yield an integer solution. In this class of ILP problems in which the constraints are written $\mathbf{Ax} \leq \mathbf{b}$, the constraint matrix $\mathbf{A}$ is *totally unimodular*. A totally unimodular matrix is one in which the determinant of every square sub-matrix is either -1, 0, or 1. When the constraint matrix is totally unimodular and $\mathbf{b}$ is integral, the solution to the linear programming (LP) relaxation of this ILP problem is also integral [Hoffman and Kruskal, 1956].

From the definition of total unimodularity, it follows that all entries of a totally unimodular matrix must be -1, 0, or 1. Thus, a CCM with constraints involving lower level subexpressions will cause TranslateToILP to output an ILP constraint matrix that is not totally unimodular. This is a serious limitation. However, high level information about the CCM can help mitigate the situation. In particular, consider the case that the set of all arguments to a disjunction is $\{y_{\nu(o)=s_i}\}$, where ν is a classifier applied to object o, and the s_i are strings. This can happen when there is some interesting subset of values in the range of ν.[4] Since we know that exactly one value in the range of ν must be selected, the disjunction can be treated in exactly the same fashion as a single ILP literal, regardless of level or parent expression. After all, $\sum_i y_{\nu(o)=s_i} \in \{0, 1\}$.

This idea is most important at the top level. For example, consider a top level implication $(\bigvee_i \nu(o) = s_i) \Rightarrow b \equiv \neg (\bigvee_i \nu(o) = s_i) \vee b$. Normally, the negation would be distributed through

[4]A disjunction of this form including all values in the range of ν is trivially true.

the disjunction via DeMorgan's law creating a lower level conjunction. The inequalities (4.3) would then leave us with a non-totally-unimodular constraint matrix. However, because of our high level knowledge about these variables, the implication can be written using only the top level disjunction inequality: $y_b + 1 - \sum_i y_{\nu(o)=s_i} \geq 1$.

Now imagine we're doing first order sequence classification with LBJ, and we've trained a classifier that takes an emission from the sequence as input and outputs a label $l \in \mathcal{S} \times \mathcal{S}$ representing both the previous and current states, where $\mathcal{S}$ is the state space. This classifier will be applied to every emission in an input sequence, making predictions of the form $y_{i,s',s}$ for every index i, where s is the predicted state at index i and s' is the predicted state at index $i - 1$. These predictions must be constrained to ensure they are coherent. We must add a constraint of the form $\bigvee_{s' \in \mathcal{S}} y_{i-1,s',s} \Rightarrow \bigvee_{\hat{s} \in \mathcal{S}} y_{i,s,\hat{s}}$ for each state s. Using the knowledge that both disjunctions in this constraint can be treated as ILP literals, our translation to inequalities will yield the efficiently solvable, totally unimodular constraint matrix given in [Roth and Yih, 2005].

4.7 Discussion

Learning Based Java already boasts several success stories. The LBJ POS tagger reports a competitive 96.6% accuracy on the standard Wall Street Journal corpus. In the named entity recognizer of [Ratinov and Roth, 2009], non-local features, gazetteers, and Wikipedia are all incorporated into a system that achieves 90.8 F_1 on the CoNLL-2003 dataset, the highest score we are aware of. The co-reference resolution system of [Bengtson and Roth, 2008] achieves state-of-the-art performance on the ACE 2004 dataset while employing only a single learned classifier and a single constraint. The semantic role labeling system of [Punyakanok et al., 2008] which includes multiple classifiers and constraints has been re-implemented in LBJ, and this new implementation is just as fast and performs just as well. Finally, the constraint and inference framework was used successfully to recognize authority in dialogue in [Mayfield and Rosé, 2011].

While we believe that LBJ makes many tasks in common learning based programs easier, there is also cause for concern. The original motivation behind the design of Learning Based Java was the common experience of many machine learning researchers that many ML techniques, while conceptually simple, often take a large engineering effort to implement. We were inspired by the idea that we could bring ML to the masses who would use it largely as a black box to help them implement whatever function they didn't know how to implement themselves. However, as designed, LBJ may tend to abstract away too many important details of a learning based program.

Our first cause for concern came from the observation that classifiers in LBJ represent only a single output variable in an inference problem, and as such, notwithstanding the discussion in Section 4.6, simple models such as a first order HMM would be decidedly inconvenient to specify. Most importantly, the relationships between output variables in the HMM inference problem are conceptually implicit, and are most efficiently enforced by hard coding them as an implementation of the Viterbi algorithm. In LBJ, there would be two options. First, the programmer employs a general purpose inference algorithm and manually implements the constraints necessary to create a sequence prediction. Second, a Viterbi implementation conforming to LBJ's general purpose inference interface is provided in the library along with a set of provisos that must be observed in the programmer's code in order to make it work. Neither option is desirable. A programming language should assist the programmer in thinking about solutions as he prefers to think about them rather than demanding a particular formula.

Another key concern was that researchers of machine learning turned out to be our main audience, and they tend to want complete access to and control over a learned model. However, LBJ forces a model to be specified in association with a particular learning algorithm. Thus, if a researcher wishes to learn and evaluate his model with three different learning algorithms, he is forced to make three copies of his learning classifier expression changing only the `with` clause in each. This is again an undesirable state of affairs. Copying and pasting of code is generally frowned on in practice, so a language that leaves one with no alternatives probably has room for improvement.

```
TRANSLATETOILP($constraint$, $level$)
    if $constraint$ is a literal containing propositional variable $b$
        if ILP variable $y_b \in \{0,1\}$ does not exist, create it
        if $level = $ TOP
            if $constraint$ contains negation, output $y_b = 0$
            else, output $y_b = 1$
        else
            if $constraint$ contains negation, return $1 - y_b$
            else, return $y_b$
    else
        if $constraint$ is a conjunction and $level = $ TOP
            for each child $\beta$ of $constraint$
                TRANSLATETOILP($\beta$, LOWER)
        else
            let $\mathcal{L} \leftarrow \emptyset$ be a set of ILP literals
            for each child $\beta$ of $constraint$
                $\mathcal{L} \leftarrow \mathcal{L} \cup \{$ TRANSLATETOILP($\beta$, LOWER) $\}$
            create new ILP variable $q_{constraint} \in \{0,1\}$
            if $constraint$ is a conjunction
                output equations (4.3)
            else if $constraint$ is a disjunction
                if $level = $ TOP, output $\sum_{l \in \mathcal{L}} l \geq 1$
                else, output equations (4.4)
            else if $constraint$ is an atleast $m$
                if $level = $ TOP, output $\sum_{l \in \mathcal{L}} l \geq m$
                else, output equations (4.5)
            else if $constraint$ is an atmost $m$
                if $level = $ TOP, output $\sum_{l \in \mathcal{L}} l \leq m$
                else, output equations (4.6)
            return $q_{constraint}$
```

Figure 4.1: Algorithm for generating linear inequality constraints from a propositional logic representation. TRANSLATETOILP takes a propositional constraint and *level* $\in \{$TOP, LOWER$\}$ as input. It then *outputs* linear inequalities and *returns* ILP literals for use in parent expressions.

Chapter 5

Evaluation

How can we tell that a programming language is designed well, or that one programming language has a better design than another? This is a complex, multi-faceted question for which quantitative answers are nearly impossible. One can certainly count semicolons or non-commented lines of some selected example codes, but these measures hardly give an accurate portrayal of the effort expended by the programmer, the quality of the resulting software, or whether any conclusions drawn truly generalize. A code written in many lines may have taken less time to write, may be easier to read and maintain, and the program it implements may be more robust and even more efficient.

Thus, we are often left with qualitative analyses that attempt to identify and characterize the most salient aspects of the programming language. A variety of approaches have been employed, including case studies which attempt to assess the success of a single programming language by studying specific codes written in that language, direct comparisons between programming languages usually also in the context of specific codes, and user studies in which the experience of multiple users of the language is characterized. In this chapter, we first give in Section 5.1 a brief survey of these methods, with a particular focus on the Cognitive Dimensions of Notations (CDs). We then present in Section 5.2 the results of our evaluation of LBJ in the form of a CD questionnaire filled out by several users who applied LBJ to a variety of different tasks.

5.1 Survey

Several different kinds of qualitative evaluation strategy have been proposed in the literature. Many authors are concerned primarily with criteria for selecting a teaching language for first-year computer science students, and so they place high weight on elegant abstractions and a low bar for entry (e.g., [Chandra and Chandra, 2005], [Gupta, 2004], [Hadjerrouit, 1998], [Parker et al., 2006], and [Mannila and de Raadt, 2006]). Others select a specific language and consider the inherent merits of its design via case studies [Thies and Amarasinghe, 2010, Schmager et al., 2010]. There is also a lot of literature whose purpose is to make direct comparisons between specific languages, also usually in the context of specific programming tasks [Chandra and Chandra, 2005, Holtz and Rasdorf, 1988, Shaw et al., 1981]. We will review a few of these approaches in this section.

We then take a more detailed look at one evaluation strategy in particular; namely, the *Cognitive Dimensions of Notations* [Green, 1989, Roast and Siddiqi, 1996] in Section 5.1.3. This approach to programming language evaluation enlists users of the language to fill out a public domain questionnaire which has been developed and extended over many years by several different authors to be as general and comprehensive as possible. We use this approach to evaluate LBJ in Section 5.2.

5.1.1 Case Studies

A case study in the style of [Thies and Amarasinghe, 2010] can give insight into the coverage of a general purpose programming language by examining its behavior in a wide variety of applications. These case studies should be designed to showcase the novel features of the language and illustrate as many of its strengths and weaknesses as possible. Generalization of the arguments made therein is intended to be implied by sheer volume and diversity the cases. Also, the more programmers contribute cases to the study, the better. This way, the success of the language in enabling high productivity is not entirely attributed to the language designer's intimate knowledge of his own language.

Another recently proposed methodology for assessing the coverage of a language is to use *design patterns* as a proxy for the accumulated knowledge of best practices in software engineering over the years (especially in object-oriented programming languages). A design pattern is an abstract solution for a commonly arising design problem in software engineering [Gamma et al., 1995]. The reasoning is that if a programming language makes easy the instantiation of most or all of the 23 design patterns proposed in [Gamma et al., 1995], there is a good chance that its ease of use will extend to many other contexts. Specifically, [Schmager et al., 2010] study the programming language Go, designed at Google, by implementing all 23 design patterns as well as porting the "pattern dense" drawing software HotDraw into Go. Their conclusions were essentially that Go is different, but not necessarily easier to use than other object oriented languages with which the authors have experience.

5.1.2 Direct Comparisons Between Languages

There have also been many studies done to compare the advantages and disadvantages of programming languages with respect to each other. [Chandra and Chandra, 2005] do a comparison between Java and C# from the perspective of first-year programming courses. They take a close look at operators and how simple object-oriented design principles manifest themselves in each language. In the end, they recommend C# for teaching new computer science students because it "seems easier."

[Shaw et al., 1981] is an extensive work that also compares four languages: FORTRAN, Cobol, Jovial, and Ironman, a proposal for a language that hadn't yet been developed. They boil each language down to a "core" set of primitives which are regarded as comprising a programmer's "mental set" of language tools used to build programs. Then a distinction is drawn between "programming in the small," in which they consider the day-to-day concerns of programmers using primitives of the language to build modules whose behaviors will be considered atomic, and "programming in the large," in which modules designed by several different programmers evolve over years of maintenance and are used to compose larger and larger modules. The effect of the programmer's mental set in each language on these different programming contexts is considered. They conclude that none of the languages are ideal, but FORTRAN is the clear loser.

Finally, [Holtz and Rasdorf, 1988] compares FORTRAN, C, Pascal, and Modula-2 for their suitability in different software engineering contexts. In particular, they attempt to assess each language's respective capacity for numerical computation and organizing data; two criteria which they deemed especially important for modern programming tasks. In the end, they conclude that a specific, flat recommendation would be impractical; instead, they offer guidelines for selecting a programming language based on consideration of a given software engineering task. In this author's opinion, these conclusions sound the most reasonable.

A	*Visibility & Juxtaposability*	The ability to view/find one or more components easily.
B	*Viscosity*	Resistance to change.
C	*Diffuseness*	Wordiness; lack of conciseness.
D	*Hard Mental Operations*	Complexity in planning (part of) a solution.
E	*Error Proneness*	Mistakes are easy to make.
F	*Closeness of Mapping*	Notation matches well that which is being described.
G	*Role Expressiveness*	Easy to determine the function of each component.
H	*Hidden Dependencies*	A change to one component affects others in non-obvious ways.
I	*Progressive Evaluation*	Ability to test incomplete specifications.
J	*Provisionality*	Ability to be imprecise when it is unclear how to proceed.
K	*Premature Commitment*	Being forced to guess (or else do extra work) in order to proceed.
L	*Consistency*	Similar semantics are expressed using similar syntax.
M	*Secondary Notation*	Notation that enhances readability but does not carry semantics.
N	*Abstraction Management*	Ability to change the notation in a user-defined way.

Table 5.1: Summaries of Cognitive Dimensions in the order they appear in Section A.4.

5.1.3 The Cognitive Dimensions of Notations

Another qualitative evaluation strategy consists of enlisting users of the programming language to fill out a questionnaire based on the *Cognitive Dimensions of Notations* [Green, 1989] [Roast and Siddiqi, 1996], about which a large body of work has been written.[1] Most notably, Microsoft has used this framework to evaluate several APIs and languages including C# [Clarke, 2001]. The cognitive dimensions were originally conceived to articulate the concerns of visual programming language users along "dimensions" such as *closeness of mapping*, *abstraction management*, *hidden dependencies*, and *viscosity*. Since then, they have inspired questionnaires for language users (e.g. [Blackwell and Green, 2000]) which amounts to a qualitative user study. However, it is easier to administer, since participants need not be present in a controlled environment while filling it out.

One of the most attractive aspects of the particular questionnaire approach which began with [Blackwell and Green, 2000] is that it has been developed and extended by several authors over many years to be as general and comprehensive as possible. The researcher applying the questionnaire to evaluate his own language does not modify it in any way before presenting it to the participants,

[1]http://www.cl.cam.ac.uk/∼afb21/CognitiveDimensions/CDbibliography.html

thereby inspiring a sense of objectivity which some other approaches lacked.

As is evident in Appendix A, questions in the questionnaire are grouped according to the CD they are designed to investigate. However, while the participants can see the grouping of the questions, they are not given the names of the CDs. Table 5.1 gives the complete list of CD names with brief descriptions in the order they appear in Section A.4. The questionnaire also presents a section O which does not correspond to any particular CD; these questions merely ask the participant if he/she believes the questionnaire overlooked something important.

Terms that appear frequently in descriptions of CDs (and the questions) include "notation," "component," and "sub-device." In our particular case, the notation is the LBJ language itself. A component of the language might be a particular operator or syntactic structure. Finally, the constraint syntax could be considered a sub-device, since its syntax is markedly different from the rest of the language. However, these terms are open for interpretation by the participants, and their varying interpretations must be accounted for.

5.2 Evaluation of LBJ

We chose the Cognitive Dimensions of Notations questionnaire approach to evaluate the success of LBJ as an LBP implementation. In the end, this evaluation technique is still merely qualitative and does not lend itself towards comparison with other languages, but we believe it makes possible a less biased evaluation than other methods, primarily because it forces the administrator of the questionnaire to ask revealing questions.

5.2.1 Participants

We presented the questionnaire whose full text is reproduced in Appendix A to six participants. They were instructed to fill it out under the assumption that the word "notation" generally referred to LBJ and that they should simply skip questions they believed did not apply to LBJ. All other instructions were provided in the questionnaire itself. Every participant filled out the questionnaire without being supervised by the administrator in any way.

Four of our six participants came from a Machine Learning and Natural Language Processing

Participant	**Expertise**	**LBJ Experience**
1	ML/NLP	2 years
2	ML/NLP	2 years
3	NLP/ML	1 year
4	NLP/ML	1 year
5	CL	2 years
6	CL	3 years

Table 5.2: The participants in our CD questionnaire are all experienced LBJ users.

background, and the last two came from Computational Linguistics. In Table 5.2, we list "ML" and "NLP" in different orders for certain participants to emphasize the fact that the first two tended to be more interested in advanced ML techniques, while the second two merely required simple multi-class classifiers to do NLP. Unfortunately, only participant 2 has any experience with the constraint and inference syntaxes in the language. The CL participants also tended to require only simple multi-class classifiers.

All of them identified themselves as having a mid-range expertise with LBJ (modestly, perhaps), and all of them understood that the main purpose of LBJ codes is to specify functions that can learn their representation from data. To be sure, six participants is not a very large sample size, however as we can see in Table 5.2, every participant has at least a year of experience with the system. As a result, they were able to give many useful criticisms and comments.

5.2.2 Results

Just as the questionnaire was grouped according to CD, so do we organize our report on the results. We omit discussions of the Visibility and Secondary Notation CDs, however, since the participants essentially considered these concerns to be non-issues or not applicable to LBJ. The major theme in these results is that the two users interested in advanced ML techniques had several serious issues with the language, while the others were happy to be able to treat ML as a black box.

LBJ's Successes

LBJ was viewed favorably almost universally along the following dimensions: Closeness of Mapping, Role Expressiveness, Progressive Evaluation, Provisionality, Consistency, and Abstraction Management. This shows firstly that the participants felt LBJ did a good job of denoting that which it was intended to denote and making its various functionalities clear to the user. It was regarded as easy to sketch out ideas in the language before they were fully formed and to evaluate a system at various stages in its development before it was completely specified. Furthermore, the syntax was fairly consistent, and the language was effective in enabling the definition of abstractions to make further coding easier.

The only exceptions to these rules were as follows. Participant 3 was not sure why the new type names `discrete` and `real` were invented when perfectly suitable Java type names were already available. This participant also did not see the purpose of the assignment operator (`<-`). In addition, participant 1 expressed that other people's LBJ code was not always easy to read, and participant 2 identified inconsistencies in the semantics of the various forms of the `sense` statement.

Criticisms of LBJ

Criticisms of LBJ were made along the following dimensions, and almost always by the advanced ML users.

Viscosity: The advanced ML users expressed dissatisfaction with LBJ's lack of exposure of the inner workings of learning and inference algorithms. Workarounds could usually be devised, but this resulted in a rather viscous system architecture that could be quite difficult to change. Nevertheless, they did also identify particular types of modifications that fit into LBJ's preconceived notion of how classifiers work, and they considered these modifications to be quite easy. The rest of the participants agreed.

Diffuseness: Once again, it was an advanced ML user who was most unhappy with the expressivity of language primitives. For example, it takes quite a lot of coding to initialize an LTU with a given weight vector. In fact, this coding cannot even take place inside the LBJ source file; it must happen directly in the Java application. Nevertheless, this must be counted as a deficiency of LBJ, since it purports to make learning classifiers easy.

Hard Mental Operations: Participant 2 noted that a lot of planning was required in order to make his large systems scale well. Technical details such as the precise representations of models and lexicons were not as accessible as might have been desired, leading to a lot of workarounds.

Error Proneness: The first three participants each noted some simple, usually syntactic mistakes that are easy to make, but that would probably be easy to catch in an IDE. The most serious such complaint was that discrete classifiers are too lenient as to the values that can be returned in their features. In particular, they accept even numerical values, and when the programmer's intention is to return real valued features, this results in very non-intuitive behavior that is hard to diagnose. Thankfully, this particular issue can be fixed with some more restrictive type checking in the LBJ compiler.

Hidden Dependencies: Participant 2 notes that while LBJ's "Makefile behavior" (see Section 4.4) can be useful, it does not take into account Java codes on which the LBJ source file depends. As a result, when a change is made on only the Java side, LBJ will not re-learn affected learning classifiers. This can lead to mysterious behavior that is hard to diagnose and to the programmer adopting a development cycle in which learning classifiers are retrained from scratch at every iteration; the latter defeats the purpose of the Makefile behavior.

Premature Commitment: Participant 2 noted that the language forces the programmer to commit to the limited CCM representation it supports, which doesn't always preclude structured learning functionality, but certainly makes a lot of preparation necessary in those cases. In addition, extra preparations are also necessary when providing the connection between a learning classifier and training data via the `from` clause.

5.2.3 Take Away Messages

Several of our participants had few criticisms of LBJ, if any. These learning based program designers needed simple multi-class classifiers, and they were not interested in managing low-level details. From their perspective, it seems LBJ suited their purposes well. We count these positive data points as signs of encouragement in the same vein as the success stories of real-world LBJ systems in Section 4.7.

However, it is also crucial that the shortcomings brought to light in this evaluation be addressed for the LBP paradigm to truly make an impact. In particular, future LBP languages should:

- provide explicit support for structured models and learning algorithms; i.e., support fully the CCM and
- increase access to all details of a model's training and evaluation, including learned parameters, lexicons, and algorithms, while maintaining as much abstraction as possible.

Our next generation LBP language, the Constrained Conditional Model Processing language described in Chapter 7 is designed to address these concerns.

Chapter 6

Case Studies

In this chapter, we'll take a closer look at some real-world examples of LBJ code while highlighting both what LBJ does well as well as the criticisms brought to light in Chapter 5. First, in Section 6.1, we revisit the Semantic Role Labeling system of [Punyakanok et al., 2008] described in Section 3.3.2. Its design fits LBJ's model (see Equations (4.1) and (4.2) in Section 4.1) quite neatly, largely because it was one of the inspirations for LBJ's design. Next, in Section 6.2, we take a close look at some Coreference Resolution systems from the literature. Some of these systems were designed with an ILP inference engine but without the use of LBJ. We'll see how LBJ could have made their constraint design more intuitive. Finally, in Section 6.3, we discuss the implementation of the Illinois POS tagger. To the NLP savvy reader, this likely sounds like the least exciting case, but do not underestimate it. The POS tagging task turns out to be an excellent stress test of an LBP language's flexibility.

6.1 Semantic Role Labeling

The SRL implementation involves two independently learned classifiers and a host of hard constraints governing their behavior at inference-time. While the constraints often involve several discrete output variables simultaneously, the features each depend on a single output variable in a simple way. In particular, each feature can be on only when its associated discrete output variable takes a particular value. Put another way, we can consider all output variables to be Boolean and simply say that each feature is the conjunction of a single output variable and some function of the input. Thus, we have the exactly the situation described by Equation (4.1), and LBJ should make the implementation of the system much easier.

```
1.  discrete% CandidateFeatures(Argument a) <-
2.     PhraseType, HeadWordAndTag, LinearPosition, Path, WordsAndTags, WordLength,
3.     ChunkLength, Chunk, ChunkPattern, ChunkPatternLength, ParsePosition,
4.     ClauseCoverage
5.
6.  discrete{false, true} ArgumentIdentifier(Argument a) <-
7.  learn ArgumentIdentifierLabel
8.     using CandidateFeatures, PredicatePOS && PhraseType,
9.           PredicatePOS && HeadWordAndTag, PredicatePOS && ParsePosition,
10.          VerbNegated && LinearPosition, VerbNegated && Path,
11.          ContainsModal && LinearPosition, ContainsModal && Path
12.    with new SparsePerceptron(.1, 0, 4)
13.    from new FilterParser(Constants.chunkTrainingData) 10 rounds
14. end
```

Figure 6.1: SRL argument identification: The learned classifier is specified as a function of feature expressions, a learning algorithm, and data.

6.1.1 Learning Classifiers

In Figure 6.1, we see the specification of the argument identifier classifier as a function of features (line 8), a learning algorithm (line 12), and data (line 13). A multi-class classifier named `ArgumentTypeLearner` (not pictured) implements the argument type classifier and is very similar. The `using` clause beginning on line 8 lists all classifiers used as feature extractors. In this case, they are all conjunctive feature expressions with the notable exception of `CandidateFeatures`, which is specified on line 1 as a list of several hard coded classifiers. The learning algorithm is instantiated on line 12 with parameters hard-coded, though we could also have used LBJ's parameter tuning syntax and `cval` clause (Section 4.2.3) to tune these parameters automatically. Finally, `Argument` objects are parsed from the data by the user-defined `FilterParser` class on line 13, and the LBJ compiler will perform 10 passes over this labeled data at training-time.

Typically, the vast majority of classifiers used for feature extraction are hard-coded. When hard coding a classifier as in Figure 6.2, arbitrary Java 1.4 syntax is available to collect the information relevant to the extracted features. If a more modern Java release would be more convenient, the user can simply implement the bulk of the classifier's functionality in the external Java source codes and retrieve the computed results in the classifier specification. Lines 1 and 2 in Figure 6.2 do precisely

```
1.  discrete PredicateLemma(Argument a) <- { return a.getVerb().lemma; }
2.  discrete PredicatePOS(Argument a) <- { return a.getVerb().partOfSpeech; }
3.  discrete% Predicate(Argument a) <- {
4.     sense PredicateLemma(a);
5.     sense PredicatePOS(a);
6.  }
```

Figure 6.2: SRL features: Hard coded classifiers can use arbitrary Java, and any classifier can be called as if it were a method elsewhere in an LBJ source file.

that. In fact, this has become a preferred strategy of LBJ programmers in general. Nonetheless, the strategy does LBJ-specific syntactic sugar such as the ability to call a classifier as if it were a regular method (lines 4 and 5) still comes in handy.

6.1.2 Enforcing Constraints

The two learned classifiers `ArgumentIdentifier` and `ArgumentTypeLearner` are trained completely independently of each other, meaning that neither makes use of the other's learned parameters or predictions in any way. In fact, neither makes use of its own predictions on other arguments during training or testing either. This scenario can lead us to a set of predictions on the arguments of a given verb that are incoherent when taken as a whole. For example, we know from the task's definition that no two overlapping phrases should be given types by the type classifier. Second, according to the semantic definition of the classifiers, the filter constraint (see Section 3.3.2) must hold. Finally, we can use the PropBank[Kingsbury and Palmer, 2002] frames to narrow our output space down to only those predictions that make sense for the given verb.

Figure 6.3 lists the LBJ source code of a constraint that prevents the argument type classifier from giving a non-null type to any two overlapping arguments in a sentence. A closer look at the code shows that it is in fact implemented as a series of small constraints involving all and only those arguments that contain a given word. Note that each `Argument` object is associated with a particular verb, since the goal is to determine its role with respect to that verb. The loops on lines 2 and 5 iterate over the verbs and words in the sentence respectively. The loop on line 8 iterates over the arguments associated with the current verb while creating a list of those arguments that

```
1.  constraint NoOverlaps(SRLSentence sentence) {
2.     for (int i = 0; i < sentence.verbCount(); ++i) {
3.        ParseTreeWord verb = sentence.getVerb(i);
4.        LinkedList forVerb = sentence.getCandidates(verb);
5.        for (int j = 0; j < sentence.words.length; ++j) {
6.           if (sentence.words[j] == verb) continue;
7.           LinkedList containsWord = new LinkedList();
8.           for (Iterator I = forVerb.iterator(); I.hasNext(); ) {
9.              Argument candidate = (Argument) I.next();
10.             ParseTreeNode constituent = candidate.getConstituent();
11.             if (constituent.firstWordIndex() <= sentence.words[j].index
12.                 && sentence.words[j].index <= constituent.lastWordIndex())
13.                containsWord.add(candidate);
14.          }
15.          atmost 1 of (Argument a in containsWord)
16.             ArgumentTypeLearner(a) !: "null";
17.       }
18.    }
19. }
```

Figure 6.3: SRL structural constraint: The task's definition says that typed arguments of a given verb cannot overlap. Violations of that definition are disallowed explicitly with a hard constraint.

contain the current word. Finally, the LBJ constraint syntax on lines 15 and 16 makes sure that at most one of the arguments in the list can be given a non-null type. This FOL-like syntax will be translated to linear inequalities at run-time.

The constraint in Figure 6.4 expresses the relationship between the two classifiers as designed by the programmer. Note how this concise definition of the filter constraint bears a striking resemblance to its formal representation in Equation (3.17), albeit with longer variable names. Of course, we've already seen that this won't always be the case, but moving pure Java code into external static methods often makes it possible.

Finally, expert knowledge derived from external resources can be injected directly into the system via constraints, as illustrated by Figure 6.5. In this case, we have a Java loop on line 2 iterating over the verbs in the sentence and LBJ syntax on lines 6-8 which will eventually encode Prop Bank's knowledge in linear inequalities. Note that the objects iterated over by LBJ's quantifiers need not be arguments of classifiers. On line 7, the `exists` quantifier iterates over a list of strings which happen to function as prediction values.

```
constraint FilterConstraint(SRLSentence sentence) {
   forall (Argument a in sentence.allArguments())
      ArgumentIdentifier(a) :: false => ArgumentTypeLearner(a) :: "null";
}
```

Figure 6.4: SRL classifier semantics: The argument identifier and type classifier are related to each other by user-design.

```
constraint LegalArguments(SRLSentence sentence) {
   for (int i = 0; i < sentence.verbCount(); ++i) {
      ParseTreeWord verb = sentence.getVerb(i);
      LinkedList forVerb = sentence.getCandidates(verb);
      LinkedList legal = PropBankFrames.getLegalArguments(verb.lemma);
      forall (Argument a in forVerb)
         exists (String type in legal)
            ArgumentTypeLearner(a) :: type;
   }
}
```

Figure 6.5: SRL domain knowledge: Constraints can also infuse expert knowledge into the system.

An important characteristic that all these constraints have in common is that they all take `SRLSentence` objects as input. That's because a single sentence is the smallest data structure that encapsulates all of the information needed to pose the SRL CCM. We know this because all of the features employed by the two classifiers operate on a single argument, and none of the constraints operate over arguments from different sentences. Thus, a sentence corresponds directly to a CCM optimization problem. In LBJ syntax, we call each `SRLSentence` instance a *head* object, and we define SRL's inference problem in terms of it.

Figure 6.6 shows just such a definition. Each inference specification has a name, and we see this inference's head object is a sentence in line 1. Lines 4-8 are a constraint declaration with exactly the same syntax inside the curly braces as those we've already seen. This one simply invokes a set of constraints defined elsewhere, conjuncting them all together. At run-time, this constraint is evaluated given the head object, and in so doing, all of the output variables involved in the inference

```
1.  inference SRLInference head SRLSentence sentence {
2.     Argument a { return a.getConstituent().getSentence(); }
3.     normalizedby new Softmax()
4.     subjectto {
5.        @NoOverlaps(sentence) /\ @NoDuplicates(sentence) /\ @VA1CV(sentence)
6.        /\ @References(sentence) /\ @Continuences(sentence)
7.        /\ @LegalArguments(sentence);
8.     }
9.     with new ILPInference(new GLPKHook())
10. }
11.
12. discrete ArgumentType(Argument a) <- SRLInference(ArgumentTypeLearner)
```

Figure 6.6: SRL inference: The CCM finally comes together in the specification of an inference problem. The result is a constrained version of the learned classifier.

problem are discovered. The optimization problem's objective function is implicitly linear in those output variables with coefficients coming from the corresponding classifiers' scores. However, those coefficients can be customized using normalization functions such as softmax, as shown in line 3. Finally, the inference algorithm is specified in line 9.

The observant reader will have noticed that we neglected to describe the so-called "head-finder" declaration[1] on line 3. In order to better understand its purpose, we need to understand how the user will retrieve the output of the CCM's optimization. With the inference structure of Figure 6.6 in place, it is now possible to define the constrained incarnation of each learned classifier involved in the CCM. The syntax on line 12 shows the constrained version of `ArgumentTypeLearner` being "pulled out" of `SRLInference`. The new `ArgumentType` classifier's output given its *argument* input will be the optimal prediction as determined by `SRLInference` for the containing *sentence*. So, in order for `SRLInference` to do its job, we need to find the head object given a learned classifier's input object. That's precisely what line 3 does for arguments and sentences.

[1]See Section 4.2.5.

6.1.3 Discussion

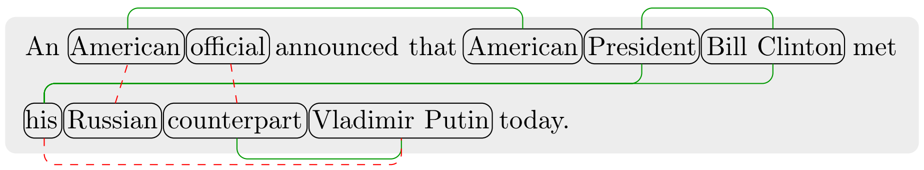
This SRL system involves a couple of independent classifiers and inference-time constraints intended to be reconciled by an ILP solver. LBJ's abstractions work well for it. In particular, learning classifier expressions automatically handle feature extraction and management, and the declarative, FOL constraint syntax enables the programmer to think about his task in terms of logical expressions rather than forcing him to solve a linear inequality puzzle. Furthermore, the end result is constrained incarnations of the original learned classifiers, making integration of ILP's complex inference infrastructure completely transparent to an end-to-end system that calls the classifiers.

Nonetheless, there is still some lingering viscosity and diffuseness in learning classifier specification (as noted in Section 5.2.2) as well as non-intuitive syntax in inference specification. The root cause of the learning classifier syntax unfriendliness is transparency. LBJ treats each learning classifier as a black box, making it hard to do any analysis or post processing over the learned parameters themselves. Workarounds can be devised on the Java side, but this adds external complexity to the development cycle, meaning that LBJ can't help the programmer manage it.

Furthermore, the inference syntax is difficult to comprehend for a first time user of the language. Without a lot of background knowledge, the purpose of the head object and head finder declarations are not obvious. Even with that background knowledge, the `inference` block structure feels like a kludge. The underlying issue is that the models we wish to express involve features over multiple output variables, while the features LBJ is capable of expressing can be applied only to a single output variable. Instead of expanding classifier related syntax to accommodate the more general case, the `inference` and `constraint` syntax attempts to tether multiple instances of the specific case to each other, but fails to render for the user a complete, global picture of the problem being solved.

6.2 Coreference Resolution

Coreference resolution is the task of determining which words and phrases in one or more documents refer to the same real world entity. ACE [NIST, 2004] defines a noun phrase referring to a real world entity as a *mention* and to mutually exclusive sets of coreferring mentions as *entities*, and



An American official announced that American President Bill Clinton met
his Russian counterpart Vladimir Putin today.

Figure 6.7: Pairwise coreference variables depicted as a graph. The nodes are mentions of real world entities. Edges between pairs of mentions represent the variables. A solid, green edge indicates that the variable is turned on, whereas a dashed, red edge indicates the opposite.

we will use that terminology here as well. Entities may be people, organizations, political groups, locations, etc., and mentions may be proper names, pronouns, nominal noun phrases, or premodifier references. For example, in Figure 6.7, the head word of each mention is circled, and solid, green edges indicate which mentions refer to the same entity. The task of coreference resolution is to take such a sentence as input and determine which phrases should be circled and which should have edges between them. Coreference resolution is an important prerequisite for many other natural language processing endeavors such as information extraction, textual entailment, question answering, and the assessment of coherence in a document.

The focus of this section will be a discussion of the issues encountered when posing coreference resolution as a CCM. We begin in Section 6.2.1 with a simple, ad-hoc model that performs very well in practice. Section 6.2.2 then introduces a class of constraints that enable the trivial incorporation of expert knowledge into a coreference resolution system: transitivity constraints. Finally, section 6.2.3 discusses the combination of multiple models such that their predictions are weighed against each other for their mutual benefit.

6.2.1 Implicit Clustering

One of the best performing coreference systems available today is the simple, pairwise model of Bengtson and Roth [Bengtson and Roth, 2008]. This model uses a comprehensive feature set to produce a high precision, low recall prediction indicating whether a given pair of mentions is coreferent. It follows that the model is often wrong when it predicts that two mentions are *not* coreferent, and in turn that predictions are often inconsistent when taken collectively. For example, given mentions $\{m_i, m_j, m_k\} \subset \mathcal{M}$, if mention pairs $\langle m_i, m_j\rangle$ and $\langle m_j, m_k\rangle$ are predicted coreferent, but mention pair $\langle m_i, m_k \rangle$ is predicted non-coreferent, how should we resolve the ambiguity?

Following [Ng and Cardie, 2002], Bengtson and Roth use the simple "Best-Link" clustering algorithm, which was found to outperform the "Closest-Link" clustering algorithm from the seminal work of [Soon et al., 2001]. In Best-Link clustering, we process the document from left to right, considering each mention m_j in turn. If there exists at least one mention m_i, $i < j$ such that the classifier produces a score on the pair $\langle m_i, m_j \rangle$ higher than some predetermined threshold θ, then a link is established between m_j and the particular m_i yielding the highest score from the classifier. This process results in a forest of links between mentions. Finally, we take the transitive closure over this forest to produce the final clustering.

Best-Link clustering is easy to implement in any programming language. However, if we encode the approach as a CCM and solve it with ILP, we may be able to add additional domain knowledge via constraints that would be difficult to incorporate in a procedural implementation. Assume we have a document $\mathcal{D}$ containing a set of mentions $\{m_i\}_{i=1}^n, m_i \in \mathcal{M}$, a pairwise scoring function $c : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$, and the threshold θ described above. We can then write the CCM objective function

$$z(\mathbf{m}, \mathbf{y}) = \sum_{j=2}^{n} \sum_{i=1}^{j-1} (c(m_i, m_j) - \theta) y_{i,j} \tag{6.1}$$

where $y_{i,j} \in \{0, 1\}$ represents the Boolean decision to establish the link $\langle m_i, m_j \rangle$. Note that an incentive to establish each link exists only when $c(m_i, m_j) > \theta$, exactly as intended. The scoring function can be learned as a standard LTU. Bengtson and Roth used the regularized, averaged Perceptron learning algorithm [Freund and Schapire, 1999].

However, this objective function will not yield the same prediction as a Best-Link decoder yet. To ensure that it does, we need to design an appropriate constraint. This is not so difficult to do in an ILP setting:

$$\begin{aligned}
\max \quad & \sum_{j=2}^{n} \sum_{i=1}^{j-1} (c(m_i, m_j) - \theta)\ y_{i,j} \\
\text{s.t.} \quad & \sum_{i=1}^{j-1} y_{i,j} \le 1 \quad \forall j \\
& 0 \le y_{i,j} \le 1 \quad \forall i, j \mid i < j \\
& y_{i,j} \ \text{ integer} \quad \forall i, j \mid i < j
\end{aligned} \tag{6.2}$$

```
discrete{false, true} PairwiseCoref(MentionPair p) <- // Definition omitted

inference BestLink head Document d {
   MentionPair p { return p.getDocument(); }
   subjectto {
      forall (Mention m_j in d)
         atmost 1 (Mention m_i in d.before(m_j))
            PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true;
   }
   with new ILPInference(new GLPKHook())
}
```

Figure 6.8: Best-Link decoding for coreference resolution expressed as an LBJ constraint.

It's also very easy to do in LBJ, as evidenced by Figure 6.8. In fact, the ILP instances generated by that code at run-time have exactly this form. Notice in particular that the object being classified is a mention pair obtained from the document object. Internally, the document is ensuring that only a single mention pair object exists for any given pair of mentions, and this solitary object is returned every time it is requested by the `getMentionPair(Mention, Mention)` method. This is important because LBJ treats classifications made on distinct objects as separate inference variables even if the programmer intended those distinct objects to represent, for example, the same mention pair.

It can be shown that all vertices of this ILP's feasible region have only integral coordinates using the theory of *total unimodularity*. Thus, no matter the objective function, efficient linear programming algorithms are sufficient to derive an integral solution [Hoffman and Kruskal, 1956]. As a result, Best-Link decoding can be carried out efficiently in an optimization framework. This result is not surprising considering the simplistic structure of the Best-Link algorithm. The antecedent selection for each mention happens completely independently of the antecedent selection for every other mention. Certain kinds of constraints such as the one in [Bengtson and Roth, 2008] which prohibits a mention that is not a pronoun from selecting a mention which is a pronoun as its direct antecedent can also be incorporated easily. However, see Section 6.2.2 for a discussion of the limitations of this formulation in general.

6.2.2 Explicit Clustering

While the inference regimen described in Section 6.2.1 is efficient and performs well, it leaves something to be desired in its linguistic motivations. Its output is a forest of mention links whose transitivity is merely implicit; simply take the transitive closure without verifying, by the classifier or any other means, the coreference of mentions not explicitly linked to each other. Thus, important information learned by the classifier about which mentions *cannot* be coreferent may be ignored. Additionally, it is difficult to enforce *as an LBJ constraint* the expert knowledge we may have about two mentions m_i and m_j that cannot be in the same cluster. Simply adding the constraint `PairwiseCoref(d.getMentionPair(m_i, m_j)) :: false` will not suffice, since solutions involving $\langle m_k, m_i \rangle$ and $\langle m_k, m_j \rangle$ for some $k < i$ will still satisfy all constraints even though they violate our intentions.

Furthermore, when expert knowledge prescribes that mentions m_i and m_j must be coreferent, the constraint `PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true` may not have the desired effect. This constraint overrides the classifier's decision to link m_j to some other mention m_k, $k < j$. In general, the sets of mentions connected to m_i and m_k are different, and switching m_j's antecedent in this way may not have a positive overall effect on the document's clustering. We'd prefer an optimization scheme that reorganizes our clustering under these conditions, but since antecedent selections are completely independent in Best-Link clustering, no such reorganization will occur.

We explore two possible CCM formulations for dealing with these issues. First, we try abandoning the Best-Link constraints for constraints that directly enforce transitivity. Next, we consider more complex formulations that preserve the spirit of Best-Link clustering while supporting additional constraints more effectively than the original Best-Link formulation on its own. Finally, we discuss some options available when the corresponding ILPs become intractable.

Encoding Transitivity Directly

The idea to enforce transitivity directly over a pairwise classifier is inspired by the notion that both positive and negative coreference evidence discovered by that classifier should be taken into account when arranging the final clustering. Managing these types of competing concerns is exactly what

CCMs are intended for. We merely need to express as constraints the idea that whenever mention pairs $\langle m_i, m_j \rangle$ and $\langle m_j, m_k \rangle$ are deemed coreferent, mention pair $\langle m_i, m_k \rangle$ should be as well. That way, when there is strong evidence against the coreference of $\langle m_i, m_k \rangle$, the ILP solver will be forced to reconsider the other two mention pairs.

The aforementioned constraints need to be enforced over all triples of mentions $\langle m_i, m_j, m_k \rangle$. We can accomplish this in an ILP setting with constraints such as:

$$\begin{aligned} \max \quad & \sum_{j=2}^{n} \sum_{i=1}^{j-1} c(m_i, m_j)\, y_{i,j} \\ \text{s.t.} \quad & y_{i,j} + y_{j,k} \leq y_{i,k} + 1 \quad \forall i, j, k \\ & 0 \leq y_{i,j} \leq 1 \quad\quad\quad \forall i, j \mid i < j \\ & y_{i,j} \;\; \text{integer} \quad\quad\quad \forall i, j \mid i < j \end{aligned} \tag{6.3}$$

The first constraint says that when both $y_{i,j}$ and $y_{j,k}$ are 1, $y_{i,k}$ will have to be 1 for the constraint to be satisfied. If either $y_{i,j}$ or $y_{j,k}$ is 0, then no requirement is enforced on the value of $y_{i,k}$. For a given triple of mentions, say $\langle m_1, m_2, m_3 \rangle$, we will have the constraints

$$y_{1,2} + y_{2,3} \leq y_{1,3} + 1 \tag{6.4a}$$

$$y_{1,2} + y_{1,3} \leq y_{2,3} + 1 \tag{6.4b}$$

$$y_{2,3} + y_{1,3} \leq y_{1,2} + 1 \tag{6.4c}$$

all of which are necessary to achieve the desired effect.

The alert reader will have noticed that ILP (6.3) actually includes many redundant linear inequality constraints. For example, if we've already enforced constraint (6.4a), there is no need to include $y_{2,3} + y_{1,2} \leq y_{1,3} + 1$ as well. Figure 6.9 illustrates LBJ code that, with the help of some convenient in-line Java, judiciously generates only as many constraints as are needed.

The CCM described in Figure 6.9 will generate $O(n^3)$ constraints, and it can incorporate additional constraints that force particular mention pairs to be coreferent or non-coreferent easily. Unfortunately, we can't use total unimodularity to guarantee that it will be efficient as we did in

```
1.  constraint Transitivity(Document d) {
2.     ArrayList allMentions = d.allMentions();
3.     for (int k = 0; k < allMentions.size(); ++k) {
4.        Mention m_k = allMentions.get(k);
5.        for (int j = 0; j < k; ++j) {
6.           Mention m_j = allMentions.get(j);
7.           MentionPair pair_jk = d.getMentionPair(m_j, m_k);
8.           for (int i = 0; i < j; ++i) {
9.              Mention m_i = allMentions.get(i);
10.             MentionPair pair_ij = d.getMentionPair(m_i, m_j);
11.             MentionPair pair_ik = d.getMentionPair(m_i, m_k);
12.             PairwiseCoref(pair_ij) :: true /\ PairwiseCoref(pair_jk) :: true
13.                => PairwiseCoref(pair_ik) :: true;
14.             PairwiseCoref(pair_ij) :: true /\ PairwiseCoref(pair_ik) :: true
15.                => PairwiseCoref(pair_jk) :: true;
16.             PairwiseCoref(pair_jk) :: true /\ PairwiseCoref(pair_ik) :: true
17.                => PairwiseCoref(pair_ij) :: true;
18.          }
19.       }
20.    }
21. }
```

Figure 6.9: An LBJ constraint that enforces transitivity on a pairwise coreference classifier. Lines 12 and 13, 14 and 15, and 16 and 17 generalize linear inequalities (6.4a), (6.4b), and (6.4c) respectively.

the Best-Link case. It's easy to see that the constraint matrix will not be totally unimodular by noting that the constraints (6.4), which are themselves a sub-matrix of any ILP generated by Figure 6.9 involving 3 or more mentions, yield a constraint matrix whose determinant is -4.

However, Finkel and Manning [Finkel and Manning, 2008] found that the vast majority of the documents in the MUC-6 and ACE Phase 2 testing sets could be processed in a reasonable amount of time. They also showed an improvement (over tractable documents) in a variety of scoring metrics as a result of enforcing transitivity. It is important to note as well that their pairwise classifier was wisely trained on all pairs of mentions. This choice is likely more amenable to their inference regimen than, for instance, the more selective set of mention pairs sampled by Bengtson and Roth [Bengtson and Roth, 2008] at training-time.

```
discrete{false, true} Connected(MentionPair p) <- learn using {} end

constraint ConnectPaths(Document d) {
   forall (Mention m_j in d.allMentions())
      forall (Mention m_i in d.before(m_j))
         Connected(d.getMentionPair(m_i, m_j)) :: true
         <=> (PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true
               \/ (exists (Mention m_k in d.allMentions())
                     m_i != m_k && m_j != m_k :: true
                     /\ PairwiseCoref(d.getMentionPair(m_i, m_k)) :: true
                     /\ Connected(d.getMentionPair(m_j, m_k)) :: true));
}
```

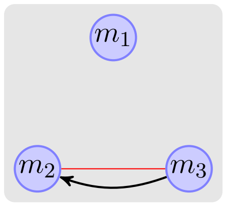
Figure 6.10: The untrained `Connected` classifier's classifications are intended to serve as auxiliary variables indicating when two mentions are connected via positive classifications of the `PairwiseCoref` classifier. However, this intuitive formulation has unwanted side effects.

Applying Constraints to Best-Link Clustering

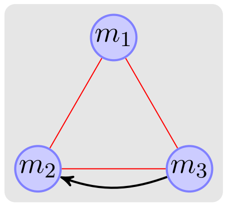
Best-Link clustering, while ad-hoc, does perform very well, so it is natural to attempt improving it by adding expert knowledge in the form of constraints. However, as discussed above, this will not be as straight forward as we might hope. We cannot simply add the transitivity constraints in Figure 6.9 to the Best-Link constraints in Figure 6.8 since they contradict each other.[2] So, we will first create a set of auxiliary variables to keep track of which mentions are coreferent by implication given a Best-Link forest. The semantics of these variables must be enforced by a carefully designed set of constraints. Then we can enforce arbitrary constraints over the auxiliary variables.

Starting from ILP (6.2) we wish to define auxiliary variables $t_{i,j} \in \{0,1\}$, $1 \leq i < j \leq n$ such that $t_{i,j} = 1$ if and only if mentions m_i and m_j are connected via some path established by the y variables. Of course, the $t_{i,j}$ variables won't actually have this property unless we constrain them to have it. The rest of this section will discuss the design of appropriate constraints for this purpose. We'll see that in LBJ, the auxiliary variables manifest themselves as the Boolean classifications of an untrained classifier on the mention pairs. It is untrained so that the scores it produces are all 0, thus leaving the objective function unaffected.

[2]In particular, if an entity contains more than two mentions, the transitivity constraints will want to link all but the first two to more than one prior mention.



(a) Auxiliary variables yielding transitive closure satisfy constraint (6.5).



(b) Auxiliary variables that merge the two Best-Link trees into one cluster satisfy constraint (6.5) as well.

Figure 6.11: A Best-Link forest (black, curved arrows) and auxiliary variables (red, straight edges) intended to represent the transitive closure. Unfortunately, both instantiations of the auxiliary variables satisfy constraints (6.5).

A Recursive Definition For our first attempt, we design our constraints "recursively," like this:

$$t_{i,j} \Leftrightarrow y_{i,j} \vee \ [\exists k,\ k \neq i \wedge k \neq j \wedge y_{k,i} \wedge t_{k,j}] \tag{6.5}$$

This constraint breaks up the definition of $t_{i,j}$ into two clauses. First, it is clear that $t_{i,j}$ should be 1 whenever $y_{i,j} = 1$. If $y_{i,j} \neq 1$, perhaps there is a mention m_k occurring elsewhere in the document such that *a*) a Best-Link edge has been directly established between m_k and m_j, and *b*) m_k is known to be connected to m_i through other Best-Link edges as indicated by our auxiliary variables (thus giving this constraint a "recursive feel"). In total, the constraint will translate to $O(n^3)$ linear inequality constraints.

Once again, this constraint has a very straight forward implementation in LBJ, as seen in Figure 6.10. Here, the `Connected` classifier applied to `d.getMentionPair(m_i, m_j)` plays the role of the $t_{i,j}$ variable. The implicit universal quantifications in constraint (6.5) are made explicit in lines 4 and 5, and the two clauses as described above can be found in line 7 and lines 8 through 11 respectively. Note that it would be quite a mentally taxing and error prone process to design manually the linear inequalities that LBJ automatically generates from this specification.

It should be clear that if the $t_{i,j}$ variables are given settings that correspond to the transitive closure of the Best-Link edges, then constraint (6.5) will be satisfied. However, this constraint also allows undesirable auxiliary variable settings. In particular, any Best-Link tree containing more than one mention can be merged into a single cluster with any other Best-Link tree via auxiliary variable settings without violating the constraint. Figure 6.11 illustrates this scenario, wherein the black, curved arrows represent Best-Link variables that have been set to 1, and the red, straight edges represent auxiliary variables that have been set to 1. In Figure 6.11b, the auxiliary variables have merged the two Best-Link trees into a single cluster.

So why should constraint (6.5) allow $t_{1,2} = 1$ and $t_{1,3} = 1$? Considering $t_{1,2}$, the second clause of the constraint is invoked, relying on the illegitimate assignment $t_{1,3} = 1$ to make its case[3]. Considering $t_{1,3}$, the second clause of the constraint is invoked again, relying on the illegitimate assignment $t_{1,2} = 1$ to make its case. In this way, the two illegitimate assignments have reinforced each other. Neither one could exist alone, but with both in place, the constraint is satisfied.

An Inductive Definition The problem with the recursive definition was that it might merge clusters that aren't supposed to be merged. One might try to prevent this by simply penalizing the auxiliary variables in the objective function. While this can coerce constraint (6.5) to give us the correct clusters, it might not be flexible enough in general. The reason is that the final score given by the objective function to the complete assignment ends up penalizing larger clusters disproportionately more than smaller clusters. If we wish to combine our coreference model with another model, this may become problematic.[4]

We will now explore the possibility of designing our auxiliary variables so that they need not be penalized. They will be designed inductively, in a series of tiers, with each new tier representing a longer path length between all pairs of mentions. The variable named $t_{i,j}^{(l)}$ will indicate whether or not there exists a path of length exactly l along Best-Link edges (possibly containing repeated edges) between mentions m_i and m_j. We will also retain the variables $t_{i,j}$ with their originally intended semantics, and their definitions will be made possible by the observation that there can be no path between two mentions (without repeated edges) of length more than $n - 1$.

[3]Note that we abuse notation slightly here, allowing e.g. $t_{1,2}$ and $t_{2,1}$ to refer to the same variable. The Java implementation of `getMentionPair(Mention, Mention)` operates similarly on lines 10 and 11 of Figure 6.10

[4]On the other hand, there may be good linguistic motivation to penalize clusters in exactly this way, since most documents have many more small clusters than large ones.

```
constraint ConnectPaths(Document d) {
   forall (Mention m_j in d.allMentions())
      forall (Mention m_i in d.before(m_j))
         (ConnectedLength2(d.getMentionPair(m_i, m_j)) :: true
          <=> (exists (Mention m_k in d.allMentions())
                  m_i != m_k && m_j != m_k :: true
                  /\ PairwiseCoref(d.getMentionPair(m_i, m_k)) :: true
                  /\ PairwiseCoref(d.getMentionPair(m_j, m_k)) :: true))
         /\ (ConnectedLength3(d.getMentionPair(m_i, m_j)) :: true
             <=> (exists (Mention m_k in d.allMentions())
                     m_i != m_k && m_j != m_k :: true
                     /\ PairwiseCoref(d.getMentionPair(m_i, m_k)) :: true
                     /\ ConnectedLength2(d.getMentionPair(m_j, m_k)) :: true));
}
```

Figure 6.12: LBJ classifiers can't be parameterized; thus it is hard to create auxiliary variables that represent connectivity over pairwise coref links. However, we can achieve short-range transitivity with tiers of auxiliary variables that increase path lengths one step at a time.

Below, we give the inductive definition of our auxiliary variables. Note that tier 2 serves as the base case, since tier 1 would be equivalent to the existing Best-Link variables.

$$t_{i,j}^{(2)} \Leftrightarrow \exists k,\ k \neq i \wedge k \neq j \wedge y_{k,j} \wedge y_{k,i} \tag{6.6}$$

$$\forall l,\ 2 < l < n,\ \ t_{i,j}^{(l)} \Leftrightarrow \exists k,\ k \neq i \wedge k \neq j \wedge y_{k,j} \wedge t_{k,i}^{(l-1)} \tag{6.7}$$

$$t_{i,j} \Leftrightarrow y_{i,j} \vee \exists l,\ 2 \leq l < n,\ t_{i,j}^{(l)} \tag{6.8}$$

Under this formulation, the $t_{i,j}$ variables have the intended semantics without the need to give them non-zero coefficients in the objective function. However, it translates to $O(n^4)$ linear inequality constraints as opposed to $O(n^3)$ in the recursive formulation.

Unfortunately, there is no easy way to implement these constraints generally in LBJ. The problem is that the quantity of auxiliary variable tiers depends on the number of mentions in the document. LBJ does not provide any mechanism for parameterizing a classifier's definition aside from the lone input object argument. However, one can still implement a predefined constant number of tiers as shown in Figure 6.12. This leaves us a notion of short-range transitivity, which may be sufficient much of the time, and which will be more tractable than full transitivity.

To reiterate, either the recursive or the inductive formulation will now support arbitrary constraints over the auxiliary variables. In particular, we can add constraints that force a particular pair of mentions either to be in the same cluster or not via `PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true` and `PairwiseCoref(d.getMentionPair(m_i, m_j)) :: false`, respectively. The ILP solver will then reorganize the Best-Link edges in accordance with these constraints.

6.2.3 Combining Multiple Models

To this point, the discussion about our learned classifiers has been concerned mainly with making sure that a particular classifier's predictions are consistent when applied repeatedly. In this section, we'll see how CCMs can combine multiple models such that they all benefit from their shared information. This type of model combination should be distinguished from so-called "pipeline" approaches such as the SRL implementation of [Punyakanok et al., 2008] in which models are evaluated in a predefined order with earlier models used as filters or features for later models. In a pipeline approach, the predictions made by each model are non-negotiable in later stages of the pipeline. In the combination approach discussed presently, however, no predictions are made until a consensus is reached among all models. Knowledge encoded in the constraints can thereby serve to improve every model's prediction quality.

Two prime examples of model combination involving coreference resolution models came from the work of Denis and Baldridge, who paired an anaphoricity model [Denis and Baldridge, 2007] and a named entity model [Denis and Baldridge, 2009] with their coreference classifier using ILP. We now discuss their approaches paying particular attention to how their linear inequality constraints were designed and how they can be implemented in LBJ.

Anaphoricity

A phrase is considered anaphoric (roughly) if it refers to some real world entity described more explicitly elsewhere in the discourse. Determining the anaphoricity of a mention differs from the coreference resolution task in two major ways. First, anaphoricity is a Boolean decision on a single mention; we are not interested in determining which entity is being referred to. Second, explicit references to the entity (e.g. proper names) are not considered anaphoric; instead, they are the mentions to which anaphoric mentions refer. That said, we will make the simplifying assumption that the only non-anaphoric mention in any cluster of coreferent mentions is the one occurring earliest in the document.

Given a model of anaphoricity, we'll need new ILP variables to represent its predictions. Let $y_i^{(a)} \in \{0, 1\}$, $1 \leq i \leq n$ indicate whether or not mention i is an anaphor. We'll also need a new scoring function $a : \mathcal{M} \rightarrow \mathbb{R}$ giving the learned anaphoricity model's score on a given mention. Next, we need to decide how our joint coreference-anaphoricity model decomposes the probability of a document. For the purposes of this exposition, let's define our joint model to simply multiply all the relevant coreference and anaphoricity probabilities together.

Finally, we will constrain our two types of variables to agree with each other. Since a mention is anaphoric (by Denis and Baldridge's definition) if it has an antecedent, we must ensure that $y_j^{(a)}$ is 1 whenever $y_{i,j}$ is 1 for some $i < j$. Furthermore, it would be inconsistent to allow $y_j^{(a)}$ to be 1 if there isn't any pair of mentions $\langle m_i, m_j \rangle, i < j$ predicted as coreferent by the pairwise classifier. Using this logic, we arrive at the formulation proposed in [Denis and Baldridge, 2007] in which the first two linear inequalities are borne of the arguments above.

$$\begin{array}{ll}
\max & \displaystyle\sum_{j=2}^{n}\sum_{i=1}^{j-1} c(m_i, m_j)\, y_{i,j} + \sum_{i=1}^{n} a(m_i)\, y_i^{(a)} \\
\text{s.t.} & y_{i,j} \leq y_j^{(a)} \quad \forall i, j \mid i < j \\
& \displaystyle\sum_{i=1}^{j-1} y_{i,j} \geq y_j^{(a)} \quad \forall j \\
& 0 \leq y_{i,j} \leq 1 \quad \forall i, j \mid i < j \\
& 0 \leq y_i^{(a)} \leq 1 \quad \forall i \\
& y_{i,j} \text{ integer} \quad \forall i, j \mid i < j \\
& y_i^{(a)} \text{ integer} \quad \forall i
\end{array} \tag{6.9}$$

Alternatively, LBJ can help us encode these ideas by writing the constraint shown in Figure 6.13. Here, we again see the above arguments explicated in a logical form. However, this form of expression expresses our requirements more directly and intuitively; we did not need to search for

```
1.  constraint ConsistentCorefAndAnaphoricity(Document d) {
2.     forall (Mention m_j in d.allMentions())
3.        forall (Mention m_i in d.before(m_j))
4.           PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true
5.              => Anaphoricity(m_j) :: true;
6.     forall (Mention m_j in d.allMentions())
7.        Anaphoricity(m_j) :: true
8.           => (exists (Mention m_i in d.before(m_j))
9.                 PairwiseCoref(d.getMentionPair(m_i, m_j)) :: true);
10. }
```

Figure 6.13: Constraining the predictions of the anaphoricity classifier to be consistent with those of the pairwise coreference classifier.

linear expressions that capture the desired logic. At run-time, the constraint expressions beginning on lines 2 and 6 of Figure 6.13 will be automatically translated into the first two linear inequalities of ILP (6.9) respectively.

Note that without transitivity constraints, a post-processing step is required in which an implicit transitive closure is taken over the links in the CCM's solution. In fact, the coreference classifier is completely unconstrained other than with respect to the anaphoricity classifier. If not for those constraints, the ILP solver would simply turn on all and only those links for which $c(m_i, m_j) > 0$. With those constraints, however, the scores from the two models are weighed against each other. This causes the predictions of the joint formulation to differ from those of the Boolean coreference classifier on its own whenever and only when the two models disagree and the anaphoricity model "wins." There are exactly two such situations:

(i) $\exists j,\ a(m_j) > 0 > \max_{i<j} c(m_i, m_j)$ and $a(m_j) > -\max_{i<j} c(m_i, m_j)$; therefore it is beneficial to link the highest (albeit negative) scoring mention pair so that the anaphoricity reward can also be collected.

(ii) $\exists j,\ a(m_j) < 0,\ |\mathcal{I}^+(j)| > 0,$ and $-a(m_j) > \sum_{i \in \mathcal{I}^+(j)} c(m_i, m_j)$, where $\mathcal{I}^+(j) = \{i \mid i < j \wedge c(m_i, m_j) > 0\}$; therefore it is beneficial to turn off all links in $\{\langle m_i, m_j \rangle \mid i \in \mathcal{I}^+(j)\}$ so that the anaphoricity penalty is not incurred.

Denis and Baldridge found that situation (i) was the more prevalent of the two, leading to an increase

$$\begin{array}{llll}
\max & \displaystyle\sum_{j=2}^{n}\sum_{i=1}^{j-1} c(m_i,m_j)\; y_{i,j} + \sum_{i=1}^{n}\left(a(m_i)\; y_i^{(a)} + \sum_{t\in\mathcal{T}} \nu(m_i,t)\; y_{i,t}^{(n)} \right) & & \\
\text{s.t.} & y_{i,j} \le y_j^{(a)} & \forall i,j \mid i<j & \multirow{2}{*}{\Big\} \text{ Anaphoricity}} \\
 & \displaystyle\sum_{i=1}^{j-1} y_{i,j} \ge y_j^{(a)} & \forall j & \\
 & 1 - y_{i,j} \ge y_{i,t}^{(n)} - y_{j,t}^{(n)} & \forall i,j,t \mid i<j, t\in\mathcal{T} & \\
 & 1 - y_{i,j} \ge y_{j,t}^{(n)} - y_{i,t}^{(n)} & \forall i,j,t \mid i<j, t\in\mathcal{T} & \Big\} \text{ Named entities} \\
 & \displaystyle\sum_{t\in\mathcal{T}} y_{i,t}^{(n)} = 1 & \forall i & \\
 & 0 \le y_{i,j}, y_i^{(a)}, y_{i,t}^{(n)} \le 1 & \forall i,j,t \mid i<j, t\in\mathcal{T} & \Big\} \text{ Boolean variables} \\
 & y_{i,j}, y_i^{(a)}, y_{i,t}^{(n)} \;\; \text{integer} & \forall i,j,t \mid i<j, t\in\mathcal{T} &
\end{array}$$

Figure 6.14: Coreference, anaphoricity, and named entity models all take part in a single ILP in which each model vets the predictions of the other two.

in the number of coreference links established by their model. This, in turn, lead to a performance boost as measured by the MUC scoring metric, which is known to favor larger clusters [Luo, 2005], and a performance degradation as measured by both B^3 and CEAF [Denis and Baldridge, 2009].

Named Entity Classification

During coreference resolution, Denis and Baldridge assume that mention boundaries are given. Since it is a reasonable assumption that all named entities appearing in a document constitute separate mentions, the named entity recognition task becomes simply named entity classification. The ACE dataset on which they evaluated their approaches offers five named entity types: *person*, *organization*, *location*, *geo-political entity*, and *facility*.

To combine a named entity model with our existing models, we wish to follow the same line of reasoning that we used when combining the anaphoricity model with the coreference model. Thus, we simply need to add a multi-class classifier's output variables as described in Section 3.2.2. We'll call these variables $y_{i,t}^{(n)} \in \{0,1\}$, $1 \le i \le n$, $t \in \mathcal{T}$, where $\mathcal{T}$ is the set of named entity types. Thus, in contrast to the Boolean models where a single ILP variable was sufficient, in this case we need one ILP variable for each possible prediction value. They must also be constrained so that only a single

```
constraint ConsistentCorefAndNamedEntities(Document d) {
   forall (Mention m_j in d.allMentions())
      forall (Mention m_i in d.before(m_j))
         PairwiseCoref(d.getMentionPair(m_i, m_j))
         => NamedEntityTypeLearner(m_i) :: NamedEntityTypeLearner(m_j);
}
```

Figure 6.15: The clever, but hard to envision named entity linear inequalities in Figure 6.14 could be replaced with this equivalent and straight forward encoding of the same constraint.

variable takes the value 1 for any given i. We'll also need a new scoring function $\nu : \mathcal{M} \times \mathcal{T} \rightarrow \mathbb{R}$ giving the learned named entity model's score on a given mention and entity type.

Finally, we must define the relationship between our new named entity model and the other two models. Denis and Baldridge impose the constraint that coreferential mentions must have the same named entity type. Adding these ideas to ILP 6.9, we get the formulation shown in Figure 6.14. Take a look at the first two named entity constraints in that figure. They are identical, except that the variables on the right hand side of the inequality switch places. In either case, the right hand side will always evaluate to a value in $\{-1, 0, 1\}$. Thus, these constraints have no effect when $y_{i,j} = 0$. However, when $y_{i,j} = 1$, we now require both versions to take a value less than or equal to 0. Since the two versions are opposites of each other, the only option is 0. In that case, for all t, $y^{(n)}_{i,t} = y^{(n)}_{j,t}$. A clever mechanism to be sure, but it likely took more time and effort to design these linear inequalities than to write the equivalent LBJ code in Figure 6.15.

We should expect the incorporation of these named entity constraints to increase the precision of our coreference model, and we hope they'd improve the accuracy of our named entity classifications as well. The precision of the coreference model would increase because of cases where it believes two mentions are coreferent, but the named entity model more strongly believes their entity types are different and disallows the link. Named entity classifications would see an improvement in accuracy if there are mentions whose classification is unclear, but which are resolved by strong coreference links. Denis and Baldridge [Denis and Baldridge, 2009] confirm these intuitions about coreference precision, even when the anaphoricity constraints are not present, although the boost is not quite as great as simply using Closest-Link clustering with no other constraints. Unfortunately, named entity classification accuracy was not reported.

6.3 Part of Speech Tagging

From a modeling perspective, LBJ was largely successful at describing solutions for the Semantic Role Labeling and coreference resolution tasks. Both tasks are large-scale structured learning problems in which many competing concerns must be reconciled. However, the complexity and scale of the tasks themselves as well as their nascence might have turned out to be an unfair bias in LBJ's favor; the investigating researchers may well have been encouraged to keep their initial approaches as simple as possible as a result.

The Part of Speech (POS) Tagging task, however, has been studied heavily for many years, and implementations thereof have been finely tuned. Furthermore, it is very natural to think of POS tagging as structured, since the tag of any given word certainly depends on the tags of the other words in the sentence. Indeed, the design of a competitive and performant POS tagger places a variety of demands on an LBP language which deserve consideration. As we consider them, we use examples from the Illinois POS Tagger[5] based on [Roth and Zelenko, 1998] as implemented in LBJ.

The Illinois POS Tagger uses two independently trained linear POS models; one classifies words that are "known" and the other classifies words that are "unknown." A word is considered known if it was observed in the labeled training data. During training, the unknown word classifier is trained on those words in the corpus that occur less often than some predefined threshold. The two classifiers have essentially the same features, which we should expect to simplify their mutual design. However, there's a complication to watch out for: the tags of the previous two words are used as features when classifying the current word. During training, tags are available for all words, and Roth and Zelenko use these labels as values for the previous tag features at that time. Of course, they are not available during testing, so the same features will take their values from the predictions made previously by whichever classifier was applied.

Both models also use the predictions of another learned model as feature values. This model is called the *baseline* classifier, and it very simply classifies each word as the most popular tag observed

[5]`http://cogcomp.cs.illinois.edu/page/software_view/3`

```
discrete POSTagger(Token w) <- {
   if (baselineTarget.observed(wordForm(w)))
      return POSTaggerKnown(w);
   return POSTaggerUnknown(w);
}

discrete baselineTarget(Token w) <-
learn POSLabel
   using wordForm
   from new POSBracketToToken(Constants.trainingAndDevData)
   with new POSBaselineLearner()
   preExtract false
end
```

Figure 6.16: At inference-time, the `POSTagger` classifier calls on one of two models depending on whether the target word was encountered during training. The baseline classifier learns using a custom learning algorithm implementation that defines an `observed(String)` method.

for that word during training. Its predictions are then used both as a noisy signal directly indicating the current word's tag as well as the next two tags in the sentence. Just as for the previous two tags, the values for the next two tags can be taken from the labels during training. However, for computational reasons, there's no good way to use the POS models' predictions on both the previous two tags and the next two tags during testing, hence the use of the baseline classifier for the next two tags.

In addition, both models are constrained to select from only a subset of all possible parts of speech as a function of the word being classified. In the case of known words, the prediction is constrained to be one of the tags observed for that word in the training data. Unknown words are restricted according to the rules generated by an inductive process described in [Mikheev, 1997].

6.3.1 Models and Features

As we can see in Figure 6.16, the Illinois POS Tagger employs one of two learned models to make a prediction on each word encountered during testing. The choice of model is determined by querying the baseline classifier which knows whether a word was observed during training or not. Implemented as `baselineTarget`, the baseline classifier uses a custom learning algorithm

```
1.  discrete labelOneBefore(Token w) <- {
2.     if (w.previous != null) {
3.        if (POSTaggerKnown.isTraining) return POSLabel(w.previous);
4.        return POSTagger(w.previous);
5.     }
6.     return "";
7.  }
8.
9.  discrete labelOrBaseline(Token w) <- {
10.    if (POSTaggerKnown.isTraining) return POSLabel(w);
11.    return baselineTarget(w);
12. }
13.
14. discrete labelOneAfter(Token w) <- {
15.    if (w.next != null) return labelOrBaseline(w.next);
16.    return "";
17. }
```

Figure 6.17: The Illinois POS Tagger's features often derive their values from different sources during training than during testing.

implemented in Java which provides some extra methods for accessing its compiled statistics, such as the `observed(String)` method. Note also that the learned classifiers `POSTaggerKnown` and `POSTaggerUnknown` are called as if they were methods on lines 3 and 4.

Most of the features used by the two learned models are functions of the tags (or some surrogate thereof) in a window around the target word. In this implementation, we use two tags both before and after the word being classified. However, as mentioned above, labels are only present during training, so our features must be capable of obtaining their values from different sources in different contexts. To make this possible, LBJ generates in each learned classifier a flag named `isTraining` that it sets at LBJ compile-time to inform features such as those in Figure 6.17 that training data is currently being processed.

The full list of features employed by the known word model can be found on lines 3 and 4 of Figure 6.18. Similarly to `labelOneBefore`, `labelTwoBefore` will call `POSTagger` on the word two before the target. These recursive calls create a computational issue. For each successive call to the tagger on a new word, all previous tags will be needlessly recomputed.[6] Hence the invention of

[6]The situation is even worse since we have two such recursive calls. In this case, the number of recursive calls that

```
1.  discrete POSTaggerKnown(Token w) cachedin w.partOfSpeech <-
2.  learn POSLabel
3.     using wordForm, baselineTarget, labelTwoBefore, labelOneBefore,
4.           labelOneAfter, labelTwoAfter, L2bL1b, L1bL1a, L1aL2a
5.     with SparseNetworkLearner {
6.        SparseAveragedPerceptron.Parameters p =
7.           new SparseAveragedPerceptron.Parameters();
8.        p.learningRate = .1;
9.        p.thickness = 2;
10.       baseLTU = new SparseAveragedPerceptron(p);
11.    }
12.    from new POSBracketToToken(Constants.trainingAndDevData) 50 rounds
13.    evaluate valueOf(w, baselineTarget.allowableTags(wordForm(w)))
14. end
```

Figure 6.18: The Illinois POS Tagger's known word model uses `cachedin` to memoize its features' recursive calls and `evaluate` to invoke an alternate classification method at inference-time.

the `cachedin` keyword (and its brothers) on line 1 of the same figure. A classifier defined with this keyword will be augmented with code that checks its argument (which must be a field access) for a `null` value. The classifier's prediction is computed and stored in this field if and only iff `null` is found. Thus, the recursive calls are memoized.

Next, we'd like to define our unknown word model using essentially the same features with the addition of some suffix related features that seem to work well. Using LBJ's composite generator syntax, it should be very simple to factor out the features in `POSTaggerKnown`'s `using` clause so they can be reused elsewhere. However, our implementation of any feature making a recursive call (which is most of them) is dependent on `POSTaggerKnown`'s `isTraining` flag. So we have two choices: either we guard each recursive call with checks on both models' `isTraining` flags or we implement new versions of all our features whose only difference is a check on `POSTaggerUnknown`'s `isTraining` flag instead of `POSTaggerKnown`'s. In the first case, the classifiers are made artificially dependent on each other; if one tries to call the other while it is training but the other isn't, the behavior will likely be unexpected.[7] In the second case, we end up with a lot of redundant code.

end up being made for each successive word in a sentence grows like the Fibonacci sequence.

[7]Such calls are not currently made being made.

6.3.2 Constraints

While the tagger's implementation does not make use of LBJ's inference infrastructure, it does still apply simple constraints over its learned models' outputs. In particular, in the case of known words, the prediction is constrained to be one of the tags observed for that word in the training data. To effect this behavior as efficiently as possible, the `evaluate` clause was invented so that the programmer could specify an alternate classification method for use whenever the classifier is invoked. `SparseNetworkLearner`, the tagger's chosen learning algorithm, would normally consider every possible POS tag whenever it computed a prediction. However, with the inclusion of the `evaluate` clause on line 13 of Figure 6.18, `SparseNetworkLearner`'s `valueOf(Object, Collection)` method is called instead, and only the tags in the provided `Collection` will be evaluated.

Unknown words are restricted according to the rules generated by an inductive algorithm. Like `baselineTarget`, the `MikheevTable` classifier, whose LBJ specification is not reproduced here, is learned with a custom Java learner implementation and invoked in `POSTaggerUnknown`'s `evaluate` clause (also not reproduced here). The training data for this learner is the same as for the unknown word model, and the induced rules in this implementation are only those that associate sets of candidate tags with prefixes and suffixes of the unknown words. See [Mikheev, 1997] for more possibilities.

6.3.3 Discussion

LBJ simplified several aspects of the Illinois POS tagger's design. Like all classifiers designed in LBJ, feature extraction from indexing to pruning is managed internally, and classifiers are easy to reuse as features for other classifiers or simply as methods called in a native Java program. However, there were also some addendums afforded to LBJ's syntax that are, perhaps, overly specified. In particular, the `evaluate` clause amounts to an inference hook that pushes all the interesting decisions to the Java side. Even the `cachedin` keyword, which can provide an impactful computational improvement, is essentially a patch for LBJ's lack of expressivity. A more general solution is desired, and it is the subject of the next chapter.

Chapter 7

Constrained Conditional Model Processing Language

Our experience with Learning Based Java lead us to desire an LBP language with native support for structured models in addition to the models that LBJ supported; i.e., a language for composing and applying general Constrained Conditional Models. We present in this chapter the Constrained Conditional Model Processing language (CCMP) which is designed from the ground up to support CCMs. Unlike LBJ, CCMP is a general purpose, Turing complete programming language in which features, sparse vectors, and models are primitive data types, and a host of novel operators are provided for indexing features, computing with vectors, and composing new models from features and existing models. CCMP affords the programmer a new level of abstraction for designing models and their combination of which no previous LBP-like formalism is capable.

So as to be unambiguous about how CCMP works, we have formalized its semantics under the K technique [Roşu and Şerbănuţă, 2010] in the rewriting logic language Maude [Clavel et al., 2007], and we report on those semantics in Section 7.1. The equations and rules therein give a precise, logical account of all the computations that take place when any operator is applied in a CCMP code. After we have given the reader a sense for our formal semantics framework, we move on to formalizations of the new types of values CCMP will be processing in Section 7.2. A discussion of the processing itself then takes place in Section 7.3. Finally, since Maude is executable, we have been able to test some simple implementations of structured learning based programs as described in Section 7.4.

7.1 Formal Semantics

CCMP is a general purpose, untyped, statically scoped, procedural programming language with some functional and declarative flavor. We now report on our experience developing an operational semantics for CCMP designed using the K technique [Roşu and Şerbănuţă, 2010] in the rewriting logic language Maude [Clavel et al., 2007]. The equations and rules therein give a precise, logical account of all the computations that take place when any operator is applied in a CCMP code. Since Maude code is executable, we get an interpreter and debugger of CCMP for free, and thus we have been able to evaluate the capabilities of the language on several important test cases, presented in Section 7.4. This section also serves to introduce the general purpose programming facilities of CCMP, laying the foundation for Section 7.3, in which we discuss its novel LBP primitives.

The CCMP language is described in around 4500 lines of Maude code specifying over 1000 rewrite rules, making a blow-by-blow account of these semantics infeasible. Instead, this section intends to introduce the K technique to the reader using the general purpose portion of CCMP as its running example.

7.1.1 Overview

Maude The Maude language is an implementation of a rewriting logic (RL) [Meseguer, 1992] with a sound and complete proof system and initial model semantics. Maude rewrites *terms* described under a *signature* modulo structural axioms such as associativity, commutativity, and identity. A signature is essentially a pair (Σ, E) where Σ is an alphabet of function symbols[1] and E is a set of equations over Σ. Terms are composed of function symbols applied to other function symbols while adhering to any syntactic rules put forth for them. The equations E describe with parameterized terms the rewrites Maude is expected to perform. Logical unification determines if the parameterized term on the left side of a given equation matches any sub-term the term currently being rewritten anywhere within its syntax. If so, the parameterized term on the right side is filled in by the unification and substituted for the matched sub-term.

In this chapter, we develop an alphabet that mimics the syntax of CCMP as well as a set of

[1]We will often use the terms "function symbol" and "operator" interchangeably in this chapter.

equational rules that describe the run-time computations it performs at each step of an execution. Maude then allows us to write terms that represent CCMP programs and their input, and from those terms, it deduces the entire execution of the programs logically.

The K Technique Since Maude is a full-fledged logic programming language, we have a wide variety of techniques at our disposal for defining the semantics of a programming language. We chose the K technique [Roşu and Şerbănuţă, 2010], which we introduce briefly here. In K, we represent program state as a *configuration*, which is a nested collection of *cells*, each of which records information about the progress of the program at a given moment in time. For example, cells may hold *stores* which map locations (read: "memory locations") to values, *environments* which map, e.g., variable names to locations, *stacks* which control the flow of execution, and even a *continuation* representing the computation itself. All of these constructs are given function symbols in Maude, and terms composed of them are rewritten in response to computational events.

The computation being executed is represented as a continuation, which is a nested stack of operations with the next operation to perform at the top of the stack. As such, the continuation sequentializes the computation being performed according to rules that break down a program's syntax into tasks. These rules are referred to as *heating/cooling* rules, and they operate as follows. Say we have an expression in our language representing a computation such as $a_1 + a_2$ where a_1 and a_2 are also arbitrary expressions in our language. A heating rule would rewrite the expression as $a_1 \curvearrowright \square + a_2$, moving a_1 closer to the top of the stack. A cooling rule would rewrite the continuation $a_1 \curvearrowright \square + a_2$ back to $a_1 + a_2$. Both rules apply conditionally; the heating rule applies only when a_1 does not represent a computed value (e.g., a literal integer), and the cooling rule applies only when it does. If we wish, we can further condition these rules to ensure that one of the arguments is evaluated before the other, but a that's language- and operator-specific decision.

In this way, computation tasks float to the top of the stack where they will be evaluated by rules that define the semantics of the language's operators in terms of primitive values, and results settle back into place where they wait for their respective contexts to become the top of the stack. All the while, the rest of the configuration is available to help effect and to be effected by these computations.

```
function insertionSort(cmp) -> (a) {
   for (var i from 1 to length a) {
      var j = i - 1 ;
      while (j >= 0 && cmp(a[j], a[j + 1]) > 0) {
         var t = a[j] ;
         a[j] = a[j + 1] ;
         a[j + 1] = t ;
         j -= 1 ;
      }
   }
   return a ;
}

function increasing(a, b) { return a - b ; }
function decreasing(a, b) { return b - a ; }
function absIncreasing(a, b) { return abs(a) - abs(b) ; }

function sortAndPrint(cmp, a) {
   var sorter = insertionSort(cmp) ;
   write array2String(sorter(a)) + "\n" ;
}

main() {
   var a = @(6, 1, -4, 7) ;
   call sortAndPrint(increasing, a) ;    // -4, 1, 6, 7
   call sortAndPrint(decreasing, a) ;    // 7, 6, 1, -4
   call sortAndPrint(absIncreasing, a) ; // 1, -4, 6, 7
}
```

Figure 7.1: Sorting a list of integers in CCMP.

7.1.2 General Purpose Programming

In this section we give an overview of the language's syntax and semantics, restricting our discussion to the general purpose programming constructs at the language's foundation. We thereby aim to *a*) set the stage for the introduction of LBP-specific operators and *b*) give the reader a sense for how the K technique in RL formalizes the semantics of a programming language in the context of familiar programming language constructs. We then move on to novel constructs in Sections 7.3.

We begin with an example program that illustrates the major concepts of the general purpose portion of the language. The program in Figure 7.1 simply sorts an array of integers three different ways and prints the resulting array each time. The program is self-contained except for the

Name	**Contents**	**Description**
`k`	The continuation	The stack of tasks representing the computation.
`genv`	The global environment	Maps identifiers to locations; only changes when names of functions and models are added to it.
`env`	The local environment	Maps identifiers to locations; always contains only those identifiers accessible by the task on top of the k stack.
`store`	The store	Maps locations to values, like memory.
`nextLoc`	An integer	The next location to be allocated in the store.
`stack`	The local execution stack	Each element stores either the local environment as it appeared when entering a new scope or a loop continuation.
`fstack`	The function stack	Each element composed of local environment, continuation, and local stack as they appeared when a function was called.
`stdin`	A string	Provides the program's standard input.
`stdout`	A string	Stores the program's standard output.

Table 7.1: CCMP's general purpose configuration cells.

`array2String` function whose details are left to the reader's imagination. The output of the program is summarized in comments adjoining lines 25, 26, and 27.

Configuration

A CCMP program is a list of function and model declarations, models being described in Section 7.3. Inside functions, local variables must be declared before they are used, and they remain live within the lexical scope in which they are declared as delimited by curly braces. Variable names declared in inner scopes shadow names declared in outer scopes, and the global scope contains only function and model names (i.e., there are no global variables). CCMP programs can read input provided as a string and can produce string output. In support of these basic operational aspects, we declare a configuration with the cells described in Table 7.1.

Data Types and Common Operations

The native values manipulated by CCMP code include: a `null` value, Booleans, integers, floating point values, strings, closures, arrays, and lists. Common operators from mainstream languages such as C++ and Java are provided for manipulating Booleans, integers, floating point values, and strings, and the semantics provide for implicit type coercion when, e.g., an integer is added to a floating point value.

Variables can be declared locally and used to store any value. Of course, values can also be stored in arrays and lists, and all via the same assignment operator. This provides a good opportunity for our first examples of semantic rewrite rules. An assignment statement treats its left hand side (LHS) differently from its right hand side (RHS). The RHS is evaluated via the normal heating and cooling rules, while the LHS waits for that evaluation to complete. Then the following rule establishes a new context in which to evaluate the LHS expression:

$$\frac{\langle \kappa_1 = R; \curvearrowright \kappa \rangle_k}{\langle \text{l-value}(\kappa_1) \curvearrowright \square = R \curvearrowright \kappa \rangle_k}$$

This notation states that when the named elements of the configuration above the line are found to be in the given form, they should be collectively rewritten as the form below the line. We shall refer to the term appearing above the line as the *head* of the rule and to the term below as the *body*. Logical unification determines if the head of the rule matches the configuration term currently being reduced anywhere in its substructure, and this unification happens modulo associativity and commutativity of the syntax comprising the configuration. Thus, it is necessary only to mention in the head those cells in the configuration (or those portions of the syntax in general) that participate in the rule.

A continuation (as represented above by κ_1) wrapped in the l-value operator is rewritten by rules that replace it with a location wrapped in the loc operator. For example, an identifier in this context simply looks up its location in the local environment:

$$\frac{\langle \text{l-value}(I) \curvearrowright \kappa \rangle_k \; \langle M \rangle_{env}}{\langle \text{loc}(M(I)) \curvearrowright \kappa \rangle_k \; \langle M \rangle_{env}}$$

In this rule, I is an identifier, and M is a *map*. Maps are terms in the rewriting logic with user-defined syntax and semantics designed specifically for use as a control structure in the semantics of a programming language; they are not values in the CCMP language. Nevertheless, rules for accessing and manipulating maps are defined side-by-side with CCMP's semantics. Here, M represents the local environment, and $M(I)$ looks up the identifier, returning an integer location.

To complete the assignment, the result value R is written to the specified location:

$$\frac{\langle \text{loc}(J) \curvearrowright \square = R \curvearrowright \kappa \rangle_k \ \langle M \rangle_{store}}{\langle \kappa \rangle_k \ \langle M[J \mapsto R] \rangle_{store}}$$

where J is an integer, M now represents the store (since variables can play different roles in each new rule), and the syntax $M[J \mapsto R]$ evaluates to a map in which the value associated with J has been replaced with R.

Closures in CCMP are only created when either a function is declared in the global scope or when a function is partially applied in a local scope. In the former case, there is no extra environment to close over; in the latter, only the arguments in the current and previous partial applications of this function are closed over. Functions are applied (partially; see below) with the syntax $C(\textit{list})$ where C is a closure and *list* is a comma delimited syntactic list of values (as opposed to a list value, discussed below).

Array values consist of a location and an integer length accessible via the `length` operator. Empty arrays can be allocated with the syntax `new array[`*int*`]`, and arrays filled with specified values can be allocated with the `@(`*list*`)` syntax in which *list* is a comma delimited syntactic construct as opposed to a list value. An example of the latter can be found on line 24 of Figure 7.1. Either way, contiguous locations in the store are allocated for the elements of the array. The array access syntax $A[J]$, where A is an array value, can then either be evaluated to the corresponding value in the store, or, in l-value context, to the location of that element. See examples of the l-value on lines 6 and 7 of Figure 7.1.

Lists work just like lists in Lisp [McCarthy, 1960]. First of all, the `nil` operator represents the empty list, and operators `first` and `rest` which are analogous to *car* and *cdr* are provided. `first` and `rest` can also be evaluated in l-value context. Next, the rough equivalent of *cons*, which takes any two values as arguments in Lisp but whose second argument must be a list value in CCMP, is the infix, binary operator `++`. When evaluated, a pair of locations is allocated in the store for the two arguments, and the operator is replaced at the top of the continuation stack with the resulting list value:

$$\frac{\langle R \mathbin{++} L \curvearrowright \kappa\rangle_k \ \langle M\rangle_{store} \ \langle J\rangle_{nextLoc}}{\langle \mathrm{list}(J) \curvearrowright \kappa\rangle_k \ \langle M[(J,, J +_{\mathrm{int}} 1) \mapsto (R,, L)]\rangle_{store} \ \langle J +_{\mathrm{int}} 2\rangle_{nextLoc}}$$

In this rule, R is a literal value, L stands for a literal list value such as list(J), κ is a continuation, M is a map, and J is an integer. Like the map, a *K-list* is a control structure whose user defined double comma syntax distinguishes the term from syntax of CCMP involving lists delimited by single commas. The syntax $M[\textit{K-list}_1 \mapsto \textit{K-list}_2]$ extends the map assignment syntax seen previously to work in a pairwise manner with two K-lists. Finally, note that mathematical addition is represented by the operator $+_{\mathrm{int}}$ to distinguish it from the `+` operator which is part of the syntax of CCMP.

Control Flow

CCMP has the usual flow control elements; e.g., `if` statements, `for` and `while` loops, and a `return` statement. The `for` statement on line 2 of Figure 7.1 illustrates one form of the looping construct in which we find expressions representing bounds on the induction variable after the `from` and `to` keywords. In that example, `i` will take each successive integral value from 1 (inclusive) to `length a` (exclusive). In general, a `for` loop is merely syntactic sugar for a `while` loop and a few extra variable declarations. As such, the rewrite rules for `for` loops simply replace them in this way, without needing to consult the rest of the configuration.

Functions

The syntax of function declaration should be straight-forward given the examples in Figure 7.1, with the possible exception of the right arrows. These help effect partial function application, which is partially supported in the following sense. The argument list of a function may be partitioned into sections such that each section represents the arguments of a function returned by supplying values for the arguments in the previous section. These sections must be declared explicitly, and all values for all arguments in a section must be supplied when invoking that section. An example of partial function application happens on line 19 of Figure 7.1.

When a function is declared, a closure representing it is stored in the global environment in association with the function's name. Finally, there is also a special `main()` function taking no arguments in which execution always begins.

7.2 LBP Preliminaries

CCMP offers several new primitive value types that directly support LBP design principles. The most important is the *feature* on which all other new types are based. Features are inspired by propositional logic, so we review a few principles thereof in Section 7.2.1 before delving into CCMP details.

7.2.1 Propositional Logic Extended

Usually, propositional logic consists of an alphabet of variable names and a set of familiar Boolean connectives. For our purposes in CCMP, we will extend propositional logic with *constant symbols* standing for objects (as opposed to Boolean truth values), a single unary *function* A, and a single binary *predicate* E[2]. A *term* is either a constant symbol or an application of A. We also introduce the notion of an *interpretation*, which is a pair $(D, \cdot^I)$ composed of a set $D \subseteq \mathcal{D}$ of objects from a domain $\mathcal{D}$ and a function $\cdot^I$ that maps predicate, function, and constant symbols to elements in D.

When CCMP interprets a formula, D will contain strings, *attribute* symbols, and implementations of A and E. The interpreted function $A^I : \mathcal{S} \rightarrow \mathcal{A}$ maps from the set of all strings $\mathcal{S}$ to the set of all attribute symbols $\mathcal{A}$. Regarding the relation E^I, our intention is to understand it as an assertion that its two arguments are equivalent, and so it will always be the case that $\forall\, y \in D$, $(y, y) \in E^I$, however there will be additional tuples detected by user code. Further discussion of this issue is deferred until Section 7.3.

In addition, we also introduce two "propositional quantifiers" which connect the elements of a list of propositional formulae. Formally, if i is an integer, $\varphi \in \mathcal{P}$ is a propositional formula from the set of all such formula, and L is a (possibly empty) list of propositional formulae, then $atleast\, i\, (L)$

[2]In fact, E replaces all Boolean variable names.

and *atmost* $i\,(L)$ are also propositional formulae whose semantics are defined inductively as follows:

$$(\text{atleast}\, i\,(L))^I \equiv i \leq 0 \text{ if } L \text{ is empty}$$

$$(\text{atleast}\, i\,(\varphi, L))^I \equiv \begin{cases} \mathit{true} & \text{if } i \leq 0 \\ (\text{atleast}\, i-1\,(L))^I & \text{if } i > 0 \wedge \varphi^I \\ (\text{atleast}\, i\,(L))^I & \text{if } i > 0 \wedge \neg\varphi^I \end{cases}$$

$$(\text{atmost}\, i\,(L))^I \equiv i \geq 0 \text{ if } L \text{ is empty}$$

$$(\text{atmost}\, i\,(\varphi, L))^I \equiv \begin{cases} \mathit{false} & \text{if } i < 0 \\ (\text{atmost}\, i-1\,(L))^I & \text{if } i \geq 0 \wedge \varphi^I \\ (\text{atmost}\, i\,(L))^I & \text{if } i \geq 0 \wedge \neg\varphi^I \end{cases}$$

These quantifiers allow a linearly sized representation for concepts that would normally occupy exponential notational space in traditional propositional logic.

7.2.2 Features

CCMP uses propositional logic to help specify *features*. A feature $F \in \mathcal{F} \equiv \mathcal{P} \times \mathbb{R}$ is a pair which we say is composed of a propositional "name" and a real value. We also define the *strength* $\varsigma^I : \mathcal{F} \to \mathbb{R}$ of a feature (φ, r) under an interpretation I as

$$\varsigma^I((\varphi, r)) \equiv \begin{cases} r & \text{if } \varphi^I \\ 0 & \text{otherwise} \end{cases}$$

In this way, propositional logic is used simultaneously to help determine a feature's strength and for distinguishing it from other features. The latter is useful when features are used to index the learned parameters in models, as discussed in Section 7.2.5. We will also be frequently referring to a propositional formula φ as a feature when it is clear from context that it can be viewed as $(\varphi, 1)$. We do this especially frequently in the context of Feature Generation Functions. If a distinction needs to be made, we can also refer to the latter form as a *real valued feature*.

Following [Cumby and Roth, 2003] who developed a very similar line of reasoning for a restricted form of FOL, we can also view a feature as a function $\chi : \mathcal{I} \to \{0, 1\}$ that maps an interpretation $I \in \mathcal{I}$ to either zero or one (*false* or *true*, respectively) indicating whether or not the feature's propositional formula is *true* under the interpretation. A feature whose propositional formula is *true* under I is said to be *active* in I. This perspective is useful for defining Feature Generation Functions, the topic of the next section.

7.2.3 Feature Generation Functions

The features designed by the CCMP programmer (and vectors thereof; see Section 7.2.4) will serve as the input to learning and inference algorithms. Since we wish to admit features in the infinite feature space, and because explicitly writing in our code the strings we expect to find in our data makes our code domain specific, we now desire a mechanism for determining which features are active in a given interpretation automatically. Feature Generation Functions (FGFs) are designed to serve exactly this purpose.

Once again following [Cumby and Roth, 2003], an FGF is a function $G : \mathcal{I} \to 2^{\mathcal{P}}$ that maps an interpretation to a set of features $G(I) \subseteq \mathcal{P}$ such that $\forall \varphi \in G(I),\ \varphi^I = 1$. Note that the definition of an FGF makes no claim about how many or what types of features will be returned; these decisions are in the programmer's hands. We also define the *extension* of an FGF G as the set of all formulae that G returns for any interpretation: $ext(G) \equiv \bigcup_{I \in \mathcal{I}} G(I)$. In general, $|ext(G)|$ can be infinite even when $G(I)$ is not.

Note that any propositional formula can be viewed as an FGF whose extension has cardinality 1. The feature either generates itself or not, depending on the interpretation.

We now define a calculus over FGFs using connectives that are analogs of the usual propositional connectives. First, an *atomic FGF* is a function $\hat{G} : \mathcal{I} \to 2^{\mathcal{P}}$ such that $\forall \varphi \in \hat{G}(I)$ we have $\varphi \equiv E(t_1, t_2)$, where t_1 and t_2 are terms. Next, if G_1 and G_2 are both FGFs, then $G_1 \wedge_G G_2$ is also an FGF defined as follows:

$$(G_1 \wedge_G G_2)(I) \equiv \{\varphi_1 \wedge \varphi_2 \,|\, \varphi_1 \in ext(G_1) \wedge \varphi_2 \in ext(G_2) \wedge (\varphi_1 \wedge \varphi_2)^I\}$$

In other words, all combinations of features from the cross product of the extensions of the two FGFs are considered under the given interpretation to determine the range of the conjunction. Not to worry, though; because of the semantics of propositional conjunction, the resulting set is constructed quickly:

$$(G_1 \wedge_G G_2)(I) = \{\varphi_1 \wedge \varphi_2 \,|\, \varphi_1 \in G_1(I) \wedge \varphi_2 \in G_2(I)\}$$

However, the picture is not so rosy when we try to define disjunction and negation connectives analogously. The resulting FGFs will produce infinite sets for non-trivial cases. So, we define a conditioning operator denoted $|_C$ for a set C of propositional formulae of interest that, when applied to an FGF G, filters out formulae produced by G when they are not also found in C. Then, in practice, when we desire the FGFs $G_1 \vee G_2$, or $\neg G$ we will settle for $(G_1 \vee G_2)|_C$ or $(\neg G)|_C$, respectively, for some set C.

To conclude our discussion on FGFs, we note that in the context of FGFs and the FGF connectives defined above, a formula behaves similarly to a propositional variable in the context of the usual Boolean connectives. The only difference is that their truth values may be dependent on each other, whereas separate propositional variables would not exhibit such a dependency. Moreover, the semantics of the FGF connectives when applied only to formulae coincides exactly with the semantics of the usual Boolean connectives. Therefore, the CCMP language need not distinguish between between the two types of connectives; we instead supply only the FGF connectives and implicitly coerce our propositional formulae to be FGFs as needed.

7.2.4 Sparse Structured Vectors

CCMP provides a native vector data type used for associating real values with features. Vectors are sparse, meaning that they only store a value for an element if that value differs from the *default*, which is also kept as a value in the vector. Any value kept in a vector can be overwritten with a new value at the programmers behest. Vectors are also structured, meaning that instead of a flat, array representation, they take the form of a hierarchical map of maps which can be organized by feature or by arbitrary identifier into sub-vectors. There are no restrictions on the formulae used to effect this hierarchical organization. In particular, FGFs may also be used as keys.

At the lowest levels of the hierarchy, we have maps from integers to real values (which should be implemented as arrays in an implementation). These integers represent the indexed input components of features in a model. Here, we see the first sign that efficient feature extraction has been built into the language; we will see in Section 7.3 that the indexing operation itself is also built-in. The output components of features in the model are the keys at the next level up in the hierarchy, the values being the sub-vector maps from the lower level themselves. Semantically, this organization implies that the output feature or FGF used as a key was individually conjuncted with each of the extracted input features in its associated sub-vector. Higher levels in the hierarchy will use identifiers as keys, and these identifiers will represent the names of sub-models in the model hierarchy described in Section 7.2.5.

This vector data structure is used both as the weight vector in a model and as the example vectors extracted from data. Only example vectors will use the upper levels of the hierarchy where identifiers are keys. In addition, feature keys appearing in a weight vector will always be propositional formulae.

7.2.5 Models

A *model* in CCMP is an object that coordinates all the information necessary to perform feature extraction, learning, and inference. First and foremost, it is a collection of learned parameters collectively referred to as the *weight vector* and indexed by features and constraints. Thus, mathematically, it represents the optimization problem in Equation (3.2). However, there is not yet enough information present to run an inference algorithm that solves the optimization, since the input is not present.

In addition to the learned parameters, it holds the FGFs that compute $\{\phi_i\}$ and $\{c_j\}$ from raw data, manages the auxiliary information that supports efficient feature extraction, and provides an interface to the rest of the program that abstracts away these details from the programmer. Thus, similar to LBJ, task-specific feature extraction code resides side-by-side with learned parameters. Unlike LBJ, it will be the programmer's responsibility to execute the feature extraction code before invoking either a learning or an inference algorithm. Furthermore, CCMP introduces a *model composition* phase in between the point where the model receives input data and the execution of feature extraction. The rest of this section describes the model objects provided by CCMP for these

Identifier:	Names this model to other models that may contain it.
Sub-models:	A set of instantiated models declared as sub-models of this instantiated model.
CM constructor:	A lambda value representing the function that constructs the next phase of instantiation for the model (namely, the conditioned model).
Label lexicon:	A pointer to the label lexicon.
Feature lexicon:	A pointer to the feature lexicon.
Weight vector:	A pointer to the weight vector.

Table 7.2: A summary of the data managed by an instantiated model.

different execution phases.

Instantiated Models

Before one can use a model, one must first instantiate it. Once instantiated, an *instantiated model* (IM) object represents a learned function; thus, it is ready to accept input data. However, since it lacks an inference algorithm, the result it produces will not be computed values of the output variables. Instead, its task will be to condition the inference problem on the received input using code provided by the programmer (see Section 7.3). At that point, the $\mathbf{x}$ in Equations (3.1) and (3.2) will be considered fixed, and all that remains is to pick values for $\mathbf{y}$. The result of applying an IM, therefore, is a conditioned model, precisely the topic of the next section.

An IM is a 6-tuple consisting of the information summarized in Table 7.2. We notice first that IMs adopt other IMs as *sub-models*. These links from parent to child models must form a DAG. If a given IM object M_1 is a sub-model of another IM object M_2, then M_1 will have an identifier used by M_2 to keep its sub-models straight. An IM object also acts in code as the lambda value that constructs a conditioned model. This is discussed in more detail in Section 7.3.

Finally, when an IM object is itself instantiated, space is allocated for the weight vector and two *lexicons*, to which the IM keeps pointers. Lexicons are simply maps that keep track of every feature or label that has been encountered by the model during training. In the specific case of the feature lexicon, the map associates the name of each encountered feature with an integer index. These integers are used to index the weight vector, which can then be implemented as an array at the lowest levels of its hierarchy (see Section 7.2.4). Dot products between vectors composed of these

Output variables:	A list of symbols representing the declared output variables.
FGFs:	A map in which the keys are sets of output variables, and each key is associated with a list of FGFs.
Constraints:	A list of propositional formulae representing constraints.
Sub-models:	A map in which the keys are sets of output variables, and each key is associated with a list of CMs whose output variables have been bound to this CM.
Binding:	A map between the output variables of the parent CM (assuming this CM is a sub-model) and this CM.
IM:	The instantiated model from which this CM was created.

Table 7.3: A summary of the data managed by a conditioned model.

indexes are found in the innermost loops of learning and inference algorithms, so implementing the weight vector with arrays (as opposed to maps, for instance) yields a big advantage in performance. CCMP provides the additional advantage of managing the details for the programmer.

Conditioned Models

A *conditioned model* (CM) object is instantiated with the data that encapsulates the input to our inference optimization problem (i.e., the $\mathbf{x}$ in Equation (3.2)). A CM represents an instance of that inference problem and touts the facility to incorporate itself into other inference problem instances (i.e., other CMs) as a sub-model. The ensuing CM sub-model graph is a DAG just as it was for IMs, and the two DAGs are created in parallel. A CM C_1 can become a sub-model of another CM C_2 if and only if the IM M_1 from which C_1 was instantiated is a sub-model of the IM M_2 from which C_2 was instantiated. In this case, however, the relationship between parent and sub-model is defined in terms of specific output variables in the parent.

A CM is another 6-tuple as described in Table 7.3. First of all, a conditioned model keeps an account of its output variables, as any inference problem representation should. Next, a CM keeps the FGFs that produce the features in our inference problem organized according to the set of output variables involved in those features. This organization will make it easier to determine which features are involved in a "partial query" as described in Section 7.3. The CM also keeps a list of propositional constraints handy for, perhaps, analysis by an inference algorithm that wants to handle constraints separately from the rest of the model.

The next two elements in the CM's 6-tuple describe its relationship to with other CMs. First, it maintains a set of sub-models that participate in defining the structure over its own output variables, again organized by the set of output variables with which the sub-model associates. Second, if this CM is a sub-model of another CM, it will contain a map from its parent's output variables to its own so that it understands the queries passed down to it about its parent's output variables. Finally, a CM keeps a pointer to the IM from which it was instantiated.

7.2.6 Structured Examples

A *structured example* essentially encapsulates the same information as a conditioned model. The difference between the two is that in a structured example, we replace the collection of FGFs with structured vectors of indexed features, and we replace the collection of sub-models with a collection of sub-examples. Thus, we may think of a structured example as a CM that has undergone the feature extraction process. As such, operators supporting learning and inference applied over structured examples will be more efficient than if those same operators were applied over the CMs.

The reader may then be left wondering why CMs exist at all; shouldn't we always move straight to the structured example? The reason for both to exist is that the decision to extract features can have implications on the learned model. Recall that an instantiated model contains a feature lexicon. This lexicon serves not only to index features, but to inform the weight vector as to how much space it should allocate. These are issues over which the programmer will require precise control in order for his structured model to scale well. When extracting features, the programmer must choose whether or not previously unseen features should be added to the lexicon. This decision is best left outside the modeling code in which CMs compose other CMs, since that code is not aware of the intended purpose of its input. The programmer will then indicate whether the fully instantiated conditioned model shall be used for training (extending the lexicon) or merely testing (lexicon remains unchanged) outside of the modeling code.

Name	Contents	Description
`Y`	Output variables	A list of output variable symbols.
`FGF`	Feature generation functions	Functions generating features associated with the IM's learned parameters.
`con`	Constraints	Features associated with user specified weights in the IM.
`subm`	Model substructure	CMs describing further the relationships among a specified set of output variables.

Table 7.4: Configuration cells nested inside the `cmodel` cell.

7.3 LBP Operators

Having introduced the foundational boilerplate of the language, we now turn to its novel LBP enhancements. In this section, we describe the syntax and selected semantics of the new value types described in Section 7.2.5. This involves several new operators as well as new configuration cells, and we begin with the latter.

7.3.1 New Configuration

Two new cells are added to the top level of the configuration to support the composition of conditioned models. First, the `cmodel` cell is the only nested cell in CCMP's configuration; it holds a set of new cells that collectively describe a single CM as it is being composed. These sub-cells are summarized in Table 7.4. Second, the `cstack` cell comes into the picture when a new CM must be composed during the construction of another CM. In this case, the entire contents of the `cmodel` cell are pushed on top of the `cstack` as a single element where they wait to be replaced after the new CM's construction is complete.

7.3.2 Composing Features

Recalling the discussion in Section 7.2.1, in order to compose features, we need strings and attributes, and we already have strings. So, CCMP now provides two operators for creating attributes. The first is the `attr` operator, which is the interpretation of the lone function A in our propositional logic. This operator does not, in fact, have any semantic rules associated with it; it serves merely to wrap a string.

Additionally, we have the unary `<>` (read: diamond) operator which creates an *output attribute* with its argument serving as its name. We must be very careful to note here, however, that the application of this operator does *not* introduce a new output variable into a CM. It is merely a building block value for composing features, which themselves are not necessarily part of a model either. As we will see, CCMP allows features (whether they involve output attributes or not) to represent knowledge a model has already acquired as well as queries that ask how well specific new knowledge would fit with the existing model.

With attributes and strings at its disposal, CCMP can now provide an operator that stands for the interpretation of our lone predicate E. The binary, infix operator `::` expects each of its arguments to be either a string or an attribute. Once it has these, the term R_1`::`R_2 is rewritten to $R_1 ::_F R_2$, denoting a propositional atom value.

From here, the language begins providing operators that build larger and larger propositional formulae, all rewritten to formula values, analogously to `::`, that can be stored in variables, passed to functions, etc. The usual Boolean connectives are given: negation (`!`), conjunction (`/\`), disjunction (`\/`), implication (`=>`), and double implication (`<=>`). Additionally, operators inspired by FOL are provided to procedurally construct propositional formulae with a concise amount of code. For example, if we have an array value stored in variable `a` which contains strings, we may write:

```
var f = exists (var i from 0 to length a) attr("word") :: a[i] ;
```

which is equivalent to

```
var f = exists (var s in a) attr("word") :: s ;
```

and both are equivalent to

```
var f = false ; for (var s in a) { f \/= attr("word") :: s ; }
```

where `x \/= y` is syntactic sugar for `x = x \/ y`. The semantics of propositional formula values include simple patterns that reduce away the Boolean truth value `false` originally stored in `f` in the last example.

Boolean formulae can be used to create real valued features simply either by applying the overloaded `::` operator with a numerical value in the second argument, or by multiplying the formula value by a scalar with the overloaded `*` operator.

7.3.3 Composing FGFs

As described in Section 7.2.3, the connective operators we provide for features are, in fact, FGF connectives. Therefore, the examples we just saw above are all FGFs, though not very exciting ones. The most interesting FGFs provided by CCMP involve output attributes. In particular, the language provides for the type coercion of an output attribute to an FGF when in the context of FGF connectives. In particular, and since it is our intention to regard the predicate E as an equivalence relation (see Section 7.2.1), an output variable V would be coerced to the FGF

$$G(I) = \{E(V,x) \,|\, x \in \mathcal{S} \wedge (V,x) \in E^I\}$$

if it were found, for example, conjuncted to some other FGF. This facility enables us immediately to describe models that jointly model discrete output variables by simply conjuncting them together.

There is also an FGF denoted `<in>` that generates every possible input feature.[3] It is an error to attempt to install this FGF in a model; it can only be used in queries.

7.3.4 Composing Models

Models are composed inside model declarations, whose syntax is similar to that of a function declaration. See Figure 7.2 for an example. We have the keyword `model` followed by an identifier naming the model, followed by lists of arguments in the partial function application syntax described for functions. However, models must have exactly two sections in this argument list. Applying the first corresponds to the instantiation of the model into an IM, and applying the second conditions the IM, creating a CM. In the body of the model declaration, we have at our disposal four types of operators that assist us in describing the CCM optimization problem. We now discuss them in turn. Keep in mind that although the model in Figure 7.2 only contains statements that involve

[3]Theoretically, that is.

```
model CPT() -> (w) {
   out oneBefore = discrete ;
   out currentTag = discrete ;
   fgf oneBefore /\ currentTag ;
   fgf currentTag /\ attr("form") :: w ;
   constraint : w == "," => currentTag :: "," ;
   constraint : w == "." => currentTag :: "." ;
}
```

Figure 7.2: The CCMP specification of a model that summarizes the conditional probability tables of a first-order HMM. We include also some simple, hard constraints for exposition.

these operators, the rest of the language is also available.

Output Variable Declarations

Output variable declarations look like assignment statements, but are introduced with the keyword `out`. On the left hand side of the assignment is an identifier, and on the right is either of the keywords `boolean` or `discrete`, possibly followed by square brackets surrounding a list of expressions evaluating to integers. This syntax specifies the type of our output variables, and allows for multi-dimensional arrays of them as well. Only when an output variable is declared in this way will it become part of the model at inference-time.

The `boolean` output variable type enables the description of models of the form described in Section 3.2.1 as conjunctions between the output variable and all features derived from input. In general, a `boolean` output variable taking the value *false* plays the same role as the truth value *false* alone. Thus, it would make all the aforementioned conjunctions inactive.

Semantically, when an output variable declaration is encountered, we simply add the specification of every individual variable to the list in the `Y` cell of the configuration. The variables in any arrays are enumerated therein.

FGF Declarations

FGF declarations declare which (types of) features will index the learned parameters of the model. Real and propositional features and FGFs are supported with one caveat: it must be possible to *partition* the top level conjunction describing the feature or FGF into two clauses, one of which only contains atoms that involve at least one output attribute, and the other of which only contains atoms that do not. This restriction makes it possible to pre-extract the input-only portions of our features before any learning or inference takes place.

The formal semantics handling this declaration includes rules that effect the partitioning alluded to above, compute the set O of output variables involved in the FGF, and install the partitioned FGF in the `FGF` configuration cell's map under the key O. Note that the structured vector described in Section 7.2.4 will be very convenient for storing these partitioned FGFs in which the input portion has been indexed. The semantics of the vector are that an output-only FGF partition appearing as a key in the vector is implicitly conjuncted individually with each of the indexed features in the associated sub-vector to form an implicit, flat vector of conjunctive features.

Constraint Declarations

Similar to feature declarations, constraint declarations accept an FGF argument, but with a different caveat. This time, every atom in the FGF must contain at least one output variable. Note that the constraints in Figure 7.2 do satisfy this requirement, because the `==` operator reduces to a Boolean truth value immediately after receiving the input. Thus, the constraint on line 6, for example, is either satisfied trivially when `w` $\neq$ `","`, or else it becomes simplified to `currentTag :: ","`.

A constraint declaration can also take a floating point argument before its colon, and when it does, the semantic rules that handle constraints install the value in the model's weight vector in association with the constraint's FGF. When the floating point argument is omitted, the constraint is *hard* meaning that its negation is installed in the weight vector associated with a weight of $-\infty$ (see Section 3.1).

Composing Models from Models

The centerpiece of CCMP's model composition framework is the ability to co-opt the entire structure of one model into another at a specified point in the larger model's structure. It enables separate models to participate in joint inference together regardless of how they were trained while enabling

```
1.  model HMM(model cpt) -> (sentence) {
2.     out dummy = discrete ;
3.     out tags = discrete[length sentence] ;
4.     if (length sentence > 0) {
5.        dummy, tags[0] -- cpt(sentence[0]) --> <> oneBefore, <> currentTag ;
6.        for (var i from 1 to length sentence) {
7.           tags[i - 1], tags[i]
8.              -- cpt(sentence[i]) --> <> oneBefore, <> currentTag ;
9.        }
10.    }
11. }
```

Figure 7.3: Connecting CPTs together in the global HMM.

programmers to organize their modeling code in a modular, reusable way.

The example code in Figure 7.3 models a sequence tagging task in a model named `HMM` and intends to use the model `CPT` from Figure 7.2 as its weight vector. Note that while `CPT` always has exactly two output variables, the number of output variables in `HMM` varies with the size of the input. Thus, `CPT` will be applied repeatedly across the sequence. This, in fact, will give `HMM` an advantage over an approach which allocates new learned parameters for every position in the sequence, assuming that there exist local patterns that the smaller model can capitalize on.

In any case, the first step is to notify `HMM` that it will be receiving a sub-model in one of its arguments using the `model` keyword. This must happen in the first section of arguments, since it is the IM which needs to label its sub-models. Next, we use the syntax that looks like a long arrow labeled by a CM exemplified on lines 5 and 8 of Figure 7.3 to establish the following. When `CPT` is conditioned at the given point in the input sequence, the structure over `CPT`'s output variables named on the right hand side of the arrow also applies to `HMM`'s output variables named on the left hand side of the arrow. This means all the same features, constraints, and sub-models (recursively) apply and those features access `CPT`'s IM. We say that a subset of `HMM`'s output variables have been *bound* to `CPT`'s output variables.

Semantically, when one of these statements is encountered during the composition of `HMM`'s CM, our rewriting rules add `CPT`'s CM to the collection of CMs in the `sub-models` cell of the configuration after imbuing it with an output variable binding map. This map points in the same direction as the arrow in the syntax, so that `CPT`'s CM will understand queries over `HMM`'s output variables. Thus, the true semantics of a model composed from other models is realized only in the context of the operations we apply on them. See Section 7.3.6 for a discussion of these.

7.3.5 Extracting Structured Examples

At the user's behest, after an entire CM with all of its substructure has been instantiated, a structured example can be extracted from the CM with one of two built-in functions. The `extract` function creates the structured example without modifying the model's feature lexicon, while the `extract+` function updates the lexicon with mappings for the new features encountered. This choice is made by the programmer outside of the modeling code.

In either case, the hierarchical structure of the extracted example runs parallel to that of the CM. It retains all of the CM's structure except for the input features looked up in the lexicon, as well as the identifiers assigned to the sub-models' associated IMs from when those IMs were passed to their parent models in arguments labeled `model`. Thus, the structure of the example will line up with the IM hierarchy.

7.3.6 Learning and Inference

Both learning and inference are done with respect to an IM and structured examples. Learning algorithms wish to update the model's weight vector(s) via some function of the example's vector(s). Inference algorithms wish to find settings of a single example's output variables such that its dot product with the weight vector is maximized. Their task may be more efficient if they can somehow take advantage of the structure present in the example's features and constraints. To support these enterprises, CCMP supplies both high level operators that handle large portions or all of the structure in IMs and examples automatically, and low level operators for picking the structure apart and examining it.

To illustrate our discussions of these operators, we provide Figures 7.4 and 7.5. Figure 7.4 is a code snippet from an implementation of the Viterbi inference algorithm. The code pictured computes optimal scores of assignments to the output variables in a left-to-right fashion across the

```
1.  function viterbi(m, x) {
2.     var ov = outs x ; // retrieve Y in an array
3.     var vals = new array[length ov] ; vals[0] = @( "" ) ;
4.     var prevTable = @( 0 ) ;
5.
6.     for (var i from 1 to length ov) {
7.        vals[i] = m valuesof ov[i] ; // retrieve the values of a Y in an array
8.        var n = length vals[i] ;
9.        var currTable = new array[n] ;
10.
11.       for (var j from 0 to n) {
12.          currTable[j] = -infinity ;
13.          for (var k from 0 to length prevTable) {
14.             var q = ov[i - 1] :: vals[i - 1][k] /\ ov[i] :: vals[i][j] ;
15.             var score = prevTable[k] + m . x(q) ;
16.             // compute score in the model associated with a particular output
17.             // variable assignment.
18.
19.             if (score > currTable[j]) { currTable[j] = score ; }
20.          }
21.       }
22.
23.       prevTable = currTable ;
24.    }
25. }
```

Figure 7.4: A snippet of code from an implementation of the Viterbi inference algorithm. On line 14, a feature asserting that two consecutive output variables take specific values is composed. On line 15, the example uses that feature as an interpretation to instantiate its FGFs into a vector. The dot product of that vector and the model's weight vector is then taken.

sequence. Code that reconstructs the optimal decisions made along the way afterwards has been omitted. This example is given to show how a structured example can be queried for the portion of its structure corresponding to a selected subset of the output variables as represented by the returned vector. The score of the model on that substructure is then a simple matter of taking the dot product of that vector with the model's weight vector. The ability to compute scores on selected substructures in this way is crucial for dynamic programming algorithms such as Viterbi.

Figure 7.5 is an implementation of Collins' Perceptron [Collins, 2002] which updates the model in response to a single training example. The same querying operation is performed, but this time using complete assignments to all output variables. One such assignment representing gold labels

```
function collins(infer, r) -> (m, x, labels) {
   m <+= r * (x(labels) - x(infer(m, x))) ;
}
```

Figure 7.5: Collins' Perceptron implemented to update the model in response to a single training example.

was given in the training data, whereas one was computed by an inference algorithm such as Viterbi.

High Level Operators

The first high level operators we consider explore a model's ability to assess the utility of a potential assignment to the output variables. First, we have the structured example's partial query facility, as illustrated in the above examples. This operation takes an output variable assignment as represented in FGF syntax and returns a structured vector containing only those features from the example's extracted vectors that are active under that assignment. In this way, the FGFs acting as keys in the example's extracted vectors view the output variable assignment as an interpretation from which features can be instantiated. This process applies recursively over the example's sub-examples, which make use of their binding maps to instantiate their own features in response to the query. The vector produced over that substructure will include higher levels in its hierarchy indicating which IM a given sub-vector indexes using the same identifier that was originally labeled with the `model` keyword in the argument list of the parent IM.

Once a vector has been extracted in this way, the model can pass judgment on it by taking its dot product with the weight vector. This process is also recursive over the structures of the IM and the structured vector. The semantics of this operation deconstruct the vector according to its substructure as described above so that dot products between sub-models and sub-vectors line up appropriately.

In Figure 7.5, the querying operation described above is combined with some simple linear algebra operators to implement Collins' Perceptron. Here, we see that structured vectors can be added to each other and multiplied by scalars, and the `<+=` operator modifies a model's weight vector by adding in the expression on the right hand side.

Finally, a model's weight vector can also be queried with an arbitrary FGF. The result is a *slice* of the weight vector containing all and only those features in the extension of the query FGF. This operator is syntactic sugar, to be sure, since there are already low level operators for taking a vector apart feature by feature, but it's particularly convenient for updating large portions of a weight vector in a non-linear way. We need this facility, for example, if we train an HMM by counting feature occurrences instead of running Collins' Perceptron, and we then wish to convert those feature counts into conditional probabilities.

Low Level Operations

The lower level operations on examples and models are essentially used for examining their components and traversing their substructure. For example, the output features and constraints can be returned to the programmer in arrays. Operators that identify the connectives used in a propositional formula and extract its children formulae are also provided. The model's weight vector can be examined in a similar way. These operators make possible an implementation in CCMP of the algorithm that generates linear inequalities for ILP inference described in Section 4.5.2.

7.4 Test Cases

In this section, we take a more detailed look at some structured learning based programs implemented in CCMP. The sequence tagging task will continue to be used as our running example, so some of the codes we'll be investigating here have already been introduced in the previous section. We start by presenting a framework within which our learned models will be evaluated, and we then discuss three different training and inference paradigms that all apply to it. Our point of emphasis here is that our specified model is truly independent of both the learning and the inference algorithm and that this gives us the flexibility to engineer solutions to structured learning problems that scale well.

```
1.  main() {
2.     var data = parseAll() ;
3.     var c = train(data[0]) ;
4.     var total = 0 ;
5.     var correct = 0 ;
6.
7.     for (var sentence in data[1]) {
8.        var predictions = c(sentence[0]) ;
9.        for (var i from 1 to length predictions) {
10.          if (sentence[1][i] == predictions[i]) { correct += 1 ; }
11.          total += 1 ;
12.       }
13.    }
14.
15.    write "(" + correct + " / " + total + ") = " + correct / (1.0 * total) ;
16. }
```

Figure 7.6: Testing Framework: A simple program that trains and evaluates a sequence tagging model.

7.4.1 Testing Framework

Our testing framework is a CCMP program that parses annotated sentences from the Penn Treebank [Marcus et al., 1993] and assesses the performance of a learned model's part-of-speech tagging predictions on them. The source code for this program is listed in Figure 7.6. On line 2, a predefined quantity of training and testing sentences are parsed into lists using a function whose code is omitted. Then on line 3, the list of training sentences is passed to an as yet undefined function named `train`. The `train` function is assumed to return a prediction function that takes an array of strings representing the words in a sentence as input and returns an array of strings representing their part-of-speech tags as output. We can see that returned function on line 8, just before its predictions are compared to the labels found in the dataset.

Also provided by this testing framework is a function named `classifier` that helps other codes implement their `train` functions by constructing the final prediction function. The source for `classifier` is listed in Figure 7.7. Its first set of arguments are assumed to represent a sequence tagging model and a function that performs inference over that model on an extracted example. The results of the inference function come back in FGF syntax so that they are more useful to learning

```
1.  function classifier(m, infer) -> (example) {
2.     var predictions = childrenof infer(m, extract(m(example))) ;
3.     var result = list2Array(predictions) ;
4.     for (var i from 0 to length result) { result[i] = # result[i] ; }
5.     return result ;
6.  }
```

Figure 7.7: Testing Framework: A function that extracts the predictions made by an inference algorithm and returns them in an array.

algorithms, `classifier` must use low level CCMP operators to extract the model's predictions and return them in an array. The function `list2Array` called on line 3 is provided in CCMP's standard library.

Finally, we draw the reader's attention once more to Figures 7.2 and 7.3 which specify the models that will be learned by all three approaches discussed below. The first two approaches also use the Viterbi inference algorithm in Figure 7.4 to make predictions using their instances of those models. The final approach will define its own algorithm for that purpose.

7.4.2 Classical HMM

A classical, first-order Hidden Markov Model implementation should stay true to the acronym name of our `CPT` model by defining its learned parameters as conditional probabilities. The features generated by the FGFs on lines 4 and 5 of Figure 7.2 should correspond to an HMM's transition and emission tables respectively. In the first instantiation of our testing framework, we show that CCMP gives the programmer full control over his CCM's objective function by defining a `train` function that normalizes the model's learned parameters after the training data has been processed.

First, the loop beginning on line 6 serves to count each conjunctive feature in the dataset. We next recall that $P(A = a|B = b) = P(A = a \wedge B = b)/(\sum_{b'} P(A = a \wedge B = b'))$ for random variables A and B. So, to normalize the weights in the `CPT` model into conditional probabilities, all that remains is to sum the appropriate groups of counts and divide each count by the sum in which it participated. The `normalize` function achieves this using FGFs to select vector slices of our weight vector as well as the `<:=` operator which overwrites a vector slice with new values. Finally,

```
function train(data) {
   var cpt = CPT() ;
   cpt default -15 ;
   var tagModel = HMM(cpt) ;
   for (var sentence in data) {
      var x = extract+(tagModel(sentence[0])) ;
      tagModel <+= x(labelOuts(x, sentence[1])) ;
   }

   call normalize(cpt, <> oneBefore, <> currentTag) ;
   call normalize(cpt, <> currentTag, <in>) ;
   return classifier(tagModel, viterbi) ;
}

function normalize(cpt, ov, g) {
   var tags = cpt valuesof ov ;
   for (var t in tags) {
      var total = 0 ;
      var v = cpt[ov :: t /\ g] ;
      for (var o in keys v) {
         for (var i in keys v[o]) { total += v[o][i] ; }
      }
      cpt <:= v map divideAndLog(total) ;
   }
}

function divideAndLog(d) -> (n) { return log(n / d) ; }
```

Figure 7.8: The classical HMM fits into our testing framework by counting occurrences of conjunctive features and then normalizing those counts.

since a CCM's objective function is linear, we take the logs of the resulting probabilities so that maximizing our model's objective function corresponds to finding the most likely assignment.

7.4.3 Joint Discriminative Training

Of course, since CCMP gives complete control over learned parameters, there's no reason why they must represent probabilities as they did in Section 7.4.2. We now show that the discriminative, structured learning algorithm of [Collins, 2002] in which learned parameters have a much different semantics is perhaps even easier to implement than the classical HMM. This isn't surprising, since discriminative models are the focus in CCMP. The `train` function in Figure 7.9 simply loops over

```
1.  function train(data) {
2.     var tagModel = HMM(CPT()) ;
3.
4.     for (var sentence in data) {
5.        var x = extract+(tagModel(sentence[0])) ;
6.        var q = labelOuts(x, sentence[1]) ;
7.        tagModel <+= .25 * (x(q) - x(viterbi(tagModel, x))) ;
8.     }
9.
10.    return classifier(tagModel, viterbi) ;
11. }
```

Figure 7.9: The Collins' Perceptron structured learning algorithm fits into our testing framework by simply applying some simple linear algebra.

training sentences extracting structured examples and querying them. The query `q` computed on line 6 using a standard CCMP library function associates a label from the data with each output variable from the structured example. Then, the second query on line 7 is computed by Viterbi, and the two vectors resulting from the queries participate in some linear algebra that updates the model's weight vector.

7.4.4 A Greedy Discriminative Architecture

In our final example, we consider a discriminative architecture in which the `CPT` model is trained outside of the context of the `HMM` model that adopts it for its larger inference purpose. This example involves more code than the other two, but its inference algorithm is linear instead of quadratic in the number of possible tags for each output variable. The training and inference protocols are inspired by the SNoW learning architecture [Carlson et al., 1999] and the PoS tagger of [Roth and Zelenko, 1998], in which predictions for previous tags are used to inform the prediction for the current tag at inference-time. It may also be useful to refer to the discussion on multi-class classification in CCMs in Section 3.2.2.

We first present in Figure 7.10 the details of the `train` function which, as it trains `CPT`, treats the first output variable (`oneBefore`) as if it were an input feature supporting the multi-class classification being made by the second output variable (`currentTag`). In fact, the set of all features in the model

```
function train(data) {
   var m = CPT() ;

   for (var sentence in data) {
      for (var i from 0 to length sentence[0]) {
         var x = extract+(m(sentence[0, i])) ;
         var ov = outs x ;
         var vals = m valuesof ov[1] ;
         var tag = sentence[1, i + 1] ;
         var found = false ;
         var q = ov[0] :: sentence[1, i] ;

         for (var val in vals) {
            var v = x(q /\ ov[1] :: val) ;
            var f = val == tag ;
            var l = f ? 1 : 0 ;
            var p = m . v > 1 ? 1 : 0 ;
            if (l != p) { m <+= .125 * (l - p) * v ; }
            found ||= f ;
         }

         if (! found) { m <+= .125 * x(q /\ ov[1] :: tag) ; }
      }
   }

   return classifier(HMM(m), greedy) ;
}
```

Figure 7.10: A simple multi-class classifier that uses the previous word's tag as a feature when predicting the current tag also fits into our testing framework.

containing the predicate `currentTag :: ` s for a given string s are collectively treated as an LTU by `train`. The inference algorithm described below will do the same.

The `train` function is organized into three nested loops. The first loops over the sentences in the training data, and the second loops over the words in each sentence. An example is extracted for each word. Finally, the innermost loop iterates over the string values associated with the `currentTag` output variable in `CPT`'s label lexicon. For each such value s, we query the example for a vector of all features containing the predicate `currentTag :: ` s. We desire this vector's dot product with the model to be greater than an arbitrary threshold (set to 1 in Figure 7.10) if and only if s is the correct tag for the current word. If that proposition turns out to be false, we make a Perceptron

```
function greedy(m, x) {
   var ov = outs x ;
   var result = ov[0] :: "" ;
   var prev = "" ;

   for (var i from 1 to length ov) {
      var vals = m valuesof ov[i] ;
      var bestScore = -infinity ;
      var best = null ;
      var q = ov[i - 1] :: prev ;

      for (var v in vals) {
         var score = m . x(q /\ ov[i] :: v) ;
         if (score > bestScore) {
            bestScore = score ;
            best = v ;
         }
      }

      result /\= ov[i] :: best ;
      prev = best ;
   }

   return result ;
}
```

Figure 7.11: An inference algorithm that greedily fixes the prediction of an independently trained classifier on one word before using that prediction as a feature with which to help make a prediction on the next word.

style update with a learning rate of 0.125. Finally, a provision is made to make an appropriate update to the model when the label lexicon did not yet contain the true label of the current word. That label is considered to have a dot product of 0 which is below the threshold in that case.

We now have a multi-class classifier that predicts the tag of the current word given the tag of the previous word. The only question that remains is how this classifier's predictions should interact with each other in the context of a full sentence. Following [Roth and Zelenko, 1998], the inference algorithm in Figure 7.11 greedily fixes the classifier's prediction at each position in the sentence and uses it as an input feature for the next prediction.

```
model transitivity() -> (n) {
   out outs = boolean[n, n] ;
   for (var i from 0 to n) {
      for (var j from i + 1 to n) {
         for (var k from j + 1 to n) {
            constraint : outs[i][j] /\ outs[j][k] => outs[i][k] ;
            constraint : outs[i][j] /\ outs[i][k] => outs[j][k] ;
            constraint : outs[i][k] /\ outs[j][k] => outs[i][j] ;
         }
      }
   }
}
```

Figure 7.12: CCMP is capable of defining abstract properties over output variables independently of any learned model. Here, we see an encoding of transitivity which can be applied to any model involving a binary, Boolean classifier.

7.4.5 Discussion

The example codes presented in this section implement three kinds of models with different theoretical underpinnings. Perhaps because they are discussed nearly independently of each other in the literature, their implementations are always independent of each other as well. Each implementation makes its own assumptions about the type of problem being solved, or worse yet, the types of algorithms the user will wish to run.

The CCM formalism accommodates all three under a single formalism, highlighting the aspects that truly separate them from one another. In particular, it highlights *a*) how each model is structurally and operationally decomposed during both learning and inference and *b*) the fact that these decisions are orthogonal to each other. Thus, CCMP enables the programmer to design the shape of his model independently of how it will be trained and evaluated. Neither the choice to train jointly or as a set of independent classifiers nor the choice to optimize a global objective function or one that's more local needs to affect how the model is declared.

Furthermore, once models are defined in CCMP, their composition in terms of each other is simplified and their capacity for abstraction is enhanced. Simply import existing models into a new one establishing a common vocabulary of output variables and establish additional relationships (i.e. features and constraints) in terms of that vocabulary as necessary. Since relations are specified between output variables instead of models, abstract model properties can be defined independently of each other and independently of any learned model. For example, in Figure 7.12, we see a set of constraints defined in a model named “transitivity” assuming only a set of Boolean output variables. When applied to the same set of output variables as a binary relational classifier with Boolean output (e.g., the pairwise coreference classifier in Section 6.2.2), that classifier becomes transitive. We need not redesign this transitivity property in the future the next time such a classifier requires it. This type of abstraction was not possible in any previous LBP-like formalism.

Chapter 8

Conclusion

Simple learning based programs are becoming ubiquitous in today's technologies. They will inevitably grow in scope and complexity in the future, but to ensure a rapid pace of development, it is crucial to organize engineering efforts under a framework of modularity and reusability. Today's implementations of these programs fall short of this ideal in large part because there has not been enough cross-breeding between machine learning formalisms that are considered disparate and incompatible. This misguided perspective leads to implementations that are seemingly uncomposable by design. It would be much more productive to focus on the similarities of popular approaches rather than differences and to aim for implementations that reflect this similarity while directly supporting and encouraging composition in larger systems.

This thesis introduces Learning Based Programming, the study of programming language formalisms that directly support programs that learn their representations from data. The vision of LBP is to enable abstraction of both learning related implementation details and domain specific concepts that cannot be unambiguously defined without learned functions. Several important steps towards that goal have been taken by this work. First, we have detailed a set of design principles that an LBP programming language should have to ensure so that modern models are supported, and scalable solutions are feasible. Second, we presented two LBP languages and demonstrated their ability to represent at a high level systems based on both structured and unstructured ML techniques. These languages should be viewed as two data points from a continuum of representational possibilities which has only recently begun to be explored.

With Learning Based Java we intended to make machine learning accessible by hiding as many feature extraction and learning and inference algorithm details as possible from the programmer. This, we thought, would enable programmers to focus on their domain specific problems rather than worrying about how machine learning works. LBJ worked well in the simplest cases, but its ideals are flawed for a couple of key reasons. First, most people, especially researchers cannot be satisfied that their system works exactly as intended unless they are sure they understand every component of that system. Second, machine learning in general does not yet offer any general purpose solutions. Thus, whether or not a programmer understands these issues at the outset of designing a new system, they will eventually be forced to evaluate several different algorithms and to tune each to the peculiarities of their specific task if they wish to uncover the best performing solution. In other words, an LBP language that hides low level details too deeply will have limited usefulness in the long run.

The CCMP programming language addresses well the most important issues that LBJ failed to address. In particular, it supports the design of arbitrary CCMs, enabling structured learning and inference algorithms applied to structured models. Furthermore, it does so with full transparency to all low level details, so that any interested programmer can customize any aspect of algorithm behavior. Simultaneously, it also provides a new level of abstraction by explicitly representing output variables and separating them from the models that describe them. This is an important step for the development of modular, reusable model property specification, and we believe it is necessary learning based systems to scale well.

However, CCMP still lacks the ability to relate model design to algorithm design. For example, our implementation of the Viterbi algorithm in Figure 7.4 makes queries to a structured example over consecutive output variables. In doing so, CCMP facilitates the algorithm's discovery of the output variables themselves, but not the relationships between them. The Viterbi implementation must trust that relationships between consecutive output variables actually exist. From another prospective, the programmer who wishes to use Viterbi for sequence tagging must understand how to establish relationships in his model so that Viterbi can make use of them. It would be very useful if a compiler could catch these types of issues at compile-time and warn the programmer that the algorithm expects relationships that aren't present in the model. It might also be useful to recognize when relationships in the model aren't addressed by the algorithm.

Model analyses can potentially yield benefits for automatic algorithm selection as well. While the general problem of algorithm selection given a model structure is NP-hard, it would be possible to recognize pre-defined patterns in a model's design and map them to algorithms appropriately. This could even be part of an interactive model design process involving both model and data analyses. All of these provide fertile ground for interesting future work.

Appendix A

Cognitive Dimensions Questionnaire

We present here the full text of the questionnaire presented to the participants of the LBJ evaluation described in Chapter 5. See [Blackwell and Green, 2000] for more details.

Thinking About Notational Systems

This questionnaire collects your views about how easy it is to use some kind of notational system. Our definition of "notational systems" includes many different ways of storing and using information - books, different ways of using pencil and paper, libraries or filing systems, software programs, computers, and smaller electronic devices. The questionnaire includes a series of questions that encourage you to think about the ways you need to use one particular notational system, and whether it helps you to do the things you need.

A.1 Background Information

- What is the name of the system?
- How long have you been using it?
- Do you consider yourself proficient in its use?
- Have you used other similar systems? (If so, please name them)

A.2 Definitions

You might need to think carefully to answer the questions in the next sections, so we have provided some definitions and an example to get you started:

Product: The product is the ultimate reason why you are using the notational system - what things happen as an end result, or what things will be produced as a result of using the notational system. This event or object is called the product. Any product that needs a notation to describe it usually has some complex structure.

Notation: The notation is how you communicate with the system - you provide information in some special format to describe the end result that you want, and the notation provides information that you can read. Notations have a structure that corresponds in some way to the structure of the product they describe. They also have parts (components, aspects etc.) that correspond in some way to parts of the product.

Notations can include text, pictures, diagrams, tables, special symbols or various combinations of these. Some systems include multiple notations. These might be quite similar to each other - for example when using a typewriter, the text that it produces is just letters and characters, while the notation on the keys that you press tells you exactly how to get the result you want. In other cases, a system might include some notations that are hard for humans to produce or to read. For example when you use a telephone the notation on the buttons is a simple arrangement of digits, but the noises you hear aren't so easy to interpret (different dialing tones for each number, clicks, and ringing tones). A telephone with a display therefore provides a further notation that is easier for the human user to understand.

Sub-devices: Complex systems can include several specialized notations to help with a specific part of the job. Some of these might not normally be considered to be part of the system, for example when you stick a Post-It note on your computer screen to remind you what to write in a word processor document.

There are two kinds of these sub-devices.

- The Post-It note is an example of a helper device. Another example is when you make notes of telephone numbers on the back of an envelope: the complete system is the telephone plus the paper notes - if you didn't have some kind of helper device like the envelope, the telephone would be much less useful.

- A redefinition device changes the main notation in some way - such as defining a keyboard shortcut, a quick-dial code on a telephone, or a macro function. The redefinition device allows you to define these shortcuts, redefine them, delete them and so on.

Note that both helper devices and redefinition devices need their own notations that are separate from the main notation of the system. We therefore ask you to consider them separately in the rest of this questionnaire.

To review how we intend to use these terms, consider the example of a word processor. The product of using the word processor is the printed letter on paper. The notation is the way that the letter looks on the screen - on modern word processors it looks pretty similar to what gets printed out, but this wasn't always the case. If you want to find and replace a particular word throughout a document, you can call up a helper device, the search and replace function, usually with its own window. This window has its own special notation - the way that you have to write the text to be found and replaced, as well as buttons that you can click on to find whole words, or to find the word in upper and lower case etc.

A.3 Parts of the System

- What task or activity do you use the system for?
- What is the product of using the system?
- What is the main notation of the system?

When using the system, what proportion of your time (as a rough percentage) do you spend:

- Searching for information within the notation `[  ]%`
- Translating substantial amounts of information from some other source into the system `[  ]%`
- Adding small bits of information to a description that you have previously created `[  ]%`
- Reorganizing and restructuring descriptions that you have previously created `[  ]%`
- Playing around with new ideas in the notation, without being sure what will result `[  ]%`

Are there any helper devices? Please list them here, and fill out a separate copy of Section 5 for each one. There are several copies of Section 5 below.

Are there any redefinition devices? Please list them here, and fill out a separate copy of Section 5 for each one.

A.4 Questions About the Main Notation

A

- How easy is it to see or find the various parts of the notation while it is being created or changed? Why?
- What kind of things are more difficult to see or find?
- If you need to compare or combine different parts, can you see them at the same time? If not, why not?

B

- When you need to make changes to previous work, how easy is it to make the change? Why?
- Are there particular changes that are more difficult or especially difficult to make? Which ones?

C

- Does the notation a) let you say what you want reasonably briefly, or b) is it long-winded? Why?
- What sorts of things take more space to describe?

D

- What kind of things require the most mental effort with this notation?

- Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

E

- Do some kinds of mistake seem particularly common or easy to make? Which ones?
- Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

F

- How closely related is the notation to the result that you are describing? Why? (Note that in a sub-device, the result may be part of another notation, rather than the end product).
- Which parts seem to be a particularly strange way of doing or describing something?

G

- When reading the notation, is it easy to tell what each part is for in the overall scheme? Why?
- Are there some parts that are particularly difficult to interpret? Which ones?
- Are there parts that you really don't know what they mean, but you put them in just because it's always been that way? What are they?

H

- If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden?
- In what ways can it get worse when you are creating a particularly large description?
- Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?

I

- How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not?
- Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?
- Can you try out partially-completed versions of the product? If not, why not?

J

- Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this?
- What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

K

- When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?
- If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

L

- Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Please give examples.
- Are there places where some things ought to be similar, but the notation makes them different? What are they?

M

- Is it possible to make notes to yourself, or express information that is not really recognized as part of the notation?
- If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw?
- Do you ever add extra marks (or colors or format choices) to clarify, emphasize or repeat what is there already? [If yes: does this constitute a helper device? If so, please fill out another Section 5 describing it. Several copies of Section 5 are available below.]

N

- Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they?
- Does the system insist that you start by defining new terms before you can do anything else? What sort of things?
- If you wrote here, you have a redefinition device: please fill out a copy of Section 5 describing it. There are several copies below.

O

- Do you find yourself using this notation in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?
- After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they? Could it be improved specifically for your own requirements?

A.5 Questions About Sub-devices

Please fill out a copy of this section for each sub-device in the system.

This page is describing (place an 'x' in one box): a helper device `[ ]`, or a redefinition device `[ ]`.

- What is its name?
- What kind of notation is used in this sub-device?

When using this sub-device, what proportion of the time using it (as a rough percentage) do you spend:

- Searching for information `[   ]%`
- Translating substantial amounts of information from some other source into the system `[   ]%`
- Adding small bits of information to a description that you have previously created `[   ]%`
- Reorganizing and restructuring descriptions that you have previously created `[   ]%`
- Playing around with new ideas in the notation, without being sure what will result `[   ]%`

In what ways is the notation in this sub-device different from the main notation? Please place an 'x' in boxes where there are differences from the main notation, and write a few words explaining the difference.

- `[ ]` Is it easy to see different parts?
- `[ ]` Is it easy to make changes?
- `[ ]` Is the notation succinct or long-winded?
- `[ ]` Do some things require hard mental effort?
- `[ ]` Is it easy to make errors or slips?
- `[ ]` Is the notation closely related to the result?

- [ ] Is it easy to tell what each part is for?
- [ ] Are dependencies visible?
- [ ] Is it easy to stop and check your work so far?
- [ ] Is it possible to sketch things out?
- [ ] Can you work in any order you like?
- [ ] Are any similarities between different parts clear?
- [ ] Can you make informal notes to yourself?
- [ ] Can you define new terms or features?
- [ ] Do you use this notation in unusual ways?
- [ ] How could the design of the notation be improved?

References

[Bengtson and Roth, 2008] Bengtson, E. and Roth, D. (2008). Understanding the Value of Features for Coreference Resolution. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, pages 294–303.

[Blackwell and Green, 2000] Blackwell, A. and Green, T. (2000). A Cognitive Dimensions Questionnaire Optimised for Users. In Blackwell, A. and Bilotta, E., editors, *Proceedings of the Workshop of the Psychology of Programming Interest Group*, pages 137–152, Corigliano Calabro, Cosenza, Italy.

[Blum, 1992] Blum, A. (1992). Learning Boolean Functions in an Infinite Attribute Space. *Machine Learning*, 9(4):373–386.

[Burges, 1998] Burges, C. J. C. (1998). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2:121–167.

[Carlson et al., 1999] Carlson, A., Cumby, C., Rosen, J., and Roth, D. (1999). The SNoW Learning Architecture. Technical Report UIUCDCS-R-99-2101, UIUC Computer Science Department.

[Casella and George, 1992] Casella, G. and George, E. I. (1992). Explaining the Gibbs Sampler. *The American Statistician*, 46(3):167–174.

[Chandra and Chandra, 2005] Chandra, S. S. and Chandra, K. (2005). A Comparison of Java and C#. *Journal of Computing Sciences in Small Colleges*, 20:238–254.

[Chang et al., 2008] Chang, M., Ratinov, L., Rizzolo, N., and Roth, D. (2008). Learning and Inference with Constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1513–1518.

[Clarke, 2001] Clarke, S. (2001). Evaluating a New Programming Language. In Kadoda, G., editor, *Proceedings of the Workshop of the Psychology of Programming Interest Group*, pages 275–289, Bournemouth University, UK.

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., editors (2007). *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer.

[Collins, 2002] Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, volume 10, pages 1–8.

[Crammer and Singer, 2003] Crammer, K. and Singer, Y. (2003). Ultraconservative Online Algorithms for Multiclass Problems. *Journal of Machine Learning Research*, 3:951–991.

[Cumby and Roth, 2003] Cumby, C. and Roth, D. (2003). Feature Extraction Languages for Propositionalized Relational Learning. In *IJCAI Workshop on Learning Statistical Models from Relational Data*.

[Denis and Baldridge, 2007] Denis, P. and Baldridge, J. (2007). Joint Determination of Anaphoricity and Coreference Resolution Using Integer Programming. In *Proceedings of the Annual Meeting of the North American Association of Computational Linguistics (NAACL)*, pages 236–243, Rochester, New York. Association for Computational Linguistics.

[Denis and Baldridge, 2009] Denis, P. and Baldridge, J. (2009). Global Joint Models for Coreference Resolution and Named Entity Classification. In *Procesamiento del Lenguaje Natural (SEPLN)*, volume 42, pages 87–96.

[Finkel and Manning, 2008] Finkel, J. R. and Manning, C. D. (2008). Enforcing Transitivity in Coreference Resolution. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 45–48, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Fischer and Schumann, 2003] Fischer, B. and Schumann, J. (2003). AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. *Journal of Functional Programming*, 13(3):483–508.

[Freund and Schapire, 1999] Freund, Y. and Schapire, R. E. (1999). Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Getoor et al., 2000] Getoor, L., Friedman, N., Koller, D., and Pfeffer, A. (2000). Learning Probabilistic Relational Models. In Džeroski, S. and Lavrač, N., editors, *Relational Data Mining*, chapter 13, pages 307–333. Springer-Verlag New York, Inc., New York, NY, USA.

[Gilks et al., 1994] Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994). A Language and Program for Complex Bayesian Modelling. *The Statistician*, 43(1):169–177.

[Goodman et al., 2008] Goodman, N. D., Mansinghka, V. K., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: A Language for Generative Models. In *Proceedings of the Annual Conference in Uncertainty in Artificial Intelligence*, pages 220–229.

[Green, 1989] Green, T. R. G. (1989). Cognitive Dimensions of Notations. In *Proceedings of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers*, pages 443–460, New York, NY, USA. Cambridge University Press.

[Gupta, 2004] Gupta, D. (2004). What is a good first programming language? *Crossroads*, 10:7–7.

[Hadjerrouit, 1998] Hadjerrouit, S. (1998). Java as First Programming Language: A Critical Evaluation. *SIGCSE Bulletin*, 30:43–47.

[Hoffman and Kruskal, 1956] Hoffman, A. and Kruskal, J. (1956). Integral Boundary Points of Convex Polyhedra. In Kuhn, H. and Tucker, A., editors, *Annals of Mathematics Studies*, volume 38, pages 223–246. Princeton University Press, Princeton, NJ. Linear Inequalities and Related Systems.

[Holtz and Rasdorf, 1988] Holtz, N. M. and Rasdorf, W. J. (1988). An Evaluation of Programming Languages and Language Features for Engineering Software Development. *Engineering with Computers*, 3:183–199. 10.1007/BF01202140.

[Kildall, 1973] Kildall, G. A. (1973). A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL, pages 194–206, New York, NY, USA. ACM.

[Kingsbury and Palmer, 2002] Kingsbury, P. and Palmer, M. (2002). From Treebank to PropBank. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, Las Palmas, Spain. ELDA.

[Lafferty et al., 2001] Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Littlestone, 1988] Littlestone, N. (1988). Learning Quickly When Irrelevant Attributes Abound: A New Linear-threshold Algorithm. *Machine Learning*, 2(4):285–318.

[Luo, 2005] Luo, X. (2005). On Coreference Resolution Performance Metrics. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, HLT '05, pages 25–32, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Mannila and de Raadt, 2006] Mannila, L. and de Raadt, M. (2006). An Objective Comparison of Languages for Teaching Introductory Programming. In *Proceedings of the Baltic Sea Conference on Computing Education Research: Koli Calling*, Baltic Sea '06, pages 32–37, New York, NY, USA. ACM.

[Marcus et al., 1993] Marcus, M. P., Santorini, B., and Marcinkiewicz, M. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

[Martins et al., 2009] Martins, A., Smith, N., and Xing, E. (2009). Concise Integer Linear Programming Formulations for Dependency Parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 342–350, Suntec, Singapore. Association for Computational Linguistics.

[Mayfield and Rosé, 2011] Mayfield, E. and Rosé, C. P. (2011). Recognizing Authority in Dialogue with an Integer Linear Programming Constrained Model. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, HLT '11, pages 1018–1026, Stroudsburg, PA, USA. Association for Computational Linguistics.

[McCallum et al., 2009] McCallum, A., Schultz, K., and Singh, S. (2009). FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., and Culotta, A., editors, *The Conference on Advances in Neural Information Processing Systems (NIPS)*, volume 22, pages 1249–1257.

[McCarthy, 1960] McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3:184–195.

[Meseguer, 1992] Meseguer, J. (1992). Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96:73–155.

[Mikheev, 1997] Mikheev, A. (1997). Automatic Rule Induction for Unknown-word Guessing. *Computational Linguistics*, 23:405–423.

[Milch et al., 2007] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. (2007). BLOG: Probabilistic models with unknown objects. In Getoor, L. and Taskar, B., editors, *Introduction to Statistical Relational Learning*, chapter 13, pages 373–398. MIT Press, Cambridge, MA.

[Ng and Cardie, 2002] Ng, V. and Cardie, C. (2002). Improving Machine Learning Approaches to Coreference Resolution. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, ACL '02, pages 104–111, Stroudsburg, PA, USA. Association for Computational Linguistics.

[NIST, 2004] NIST (2004). The ace evaluation plan. www.nist.gov/speech/tests/ace/index.html.

[Parker et al., 2006] Parker, K. R., Ottaway, T. A., Chao, J. T., and Chang, J. (2006). A Formal Language Selection Process for Introductory Programming Courses. *Journal of Information Technology Education*, 5:133–151.

[Patel et al., 2008] Patel, K., Fogarty, J., Landay, J. A., and Harrison, B. (2008). Investigating Statistical Machine Learning as a Tool for Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 667–676, New York, NY, USA. ACM.

[Pfeffer, 2007] Pfeffer, A. (2007). The Design and Implementation of IBAL: A General Purpose Probabilistic Programming Language. In Getoor, L. and Taskar, B., editors, *Introduction to Statistical Relational Learning*, chapter 14, pages 399–432. MIT Press, Cambridge, MA.

[Plaisted and Greenbaum, 1986] Plaisted, D. A. and Greenbaum, S. (1986). A Structure-preserving Clause Form Translation. *Journal of Symbolic Computing*, 2:293–304.

[Punyakanok et al., 2008] Punyakanok, V., Roth, D., and Yih, W. (2008). The Importance of Syntactic Parsing and Inference in Semantic Role Labeling. *Computational Linguistics*, 34(2):257–287.

[Rabiner, 1989] Rabiner, L. R. (1989). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–285.

[Ratinov and Roth, 2009] Ratinov, L. and Roth, D. (2009). Design Challenges and Misconceptions in Named Entity Recognition. In *Proceedings of the Annual Conference on Computational Natural Language Learning (CoNLL)*, pages 147–155.

[Richardson and Domingos, 2006] Richardson, M. and Domingos, P. (2006). Markov Logic Networks. *Machine Learning Journal*, 62(1-2):107–136.

[Rizzolo and Roth, 2007] Rizzolo, N. and Roth, D. (2007). Modeling Discriminative Global Inference. In *Proceedings of the First International Conference on Semantic Computing (ICSC)*, pages 597–604, Irvine, California. IEEE.

[Rizzolo and Roth, 2010] Rizzolo, N. and Roth, D. (2010). Learning Based Java for Rapid Development of NLP Systems. In *Proceedings of the Language Resources and Evaluation Conference (LREC)*, Valletta, Malta. European Language Resources Association.

[Roast and Siddiqi, 1996] Roast, C. R. and Siddiqi, J. I. (1996). The Formal Examination of Cognitive Dimensions. In Blandford, A. and Thimbleby, H., editors, *HCI96 Industry Day and Adjunct Proceedings*, pages 150–156.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386–408. (Reprinted in *Neurocomputing* (MIT Press, 1988).).

[Roşu and Şerbănuţă, 2010] Roşu, G. and Şerbănuţă, T. F. (2010). An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434.

[Roth, 1999] Roth, D. (1999). Learning in Natural Language. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 898–904.

[Roth and Yih, 2005] Roth, D. and Yih, W. (2005). Integer Linear Programming Inference for Conditional Random Fields. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 737–744.

[Roth and Zelenko, 1998] Roth, D. and Zelenko, D. (1998). Part of Speech Tagging Using a Network of Linear Separators. In *COLING-ACL, The 17th International Conference on Computational Linguistics*, pages 1136–1142.

[Sato and Kameya, 1997] Sato, T. and Kameya, Y. (1997). PRISM: A Language for Symbolic-Statistical Modeling. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1330–1339.

[Sato et al., 2005] Sato, T., Kameya, Y., and Zhou, N.-F. (2005). Generative Modeling with Failure in PRISM. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 847–852, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Schmager et al., 2010] Schmager, F., Cameron, N., and Noble, J. (2010). GoHotDraw: Evaluating the Go Programming Language with Design Patterns. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 10:1–10:6, New York, NY, USA. ACM.

[Shaw et al., 1981] Shaw, M., Almes, G. T., Newcomer, J. M., Reid, B. K., and Wulf, W. A. (1981). A Comparison of Programming Languages for Software Engineering. *Software: Practice and Experience*, 11(1):1–52.

[Soon et al., 2001] Soon, W. M., Ng, H. T., and Lim, D. C. Y. (2001). A Machine Learning Approach to Coreference Resolution of Noun Phrases. *Computational Linguistics*, 27:521–544.

[Taskar, 2002] Taskar, B. (2002). Discriminative Probabilistic Models for Relational Data. In *Proceedings of the Annual Conference in Uncertainty in Artificial Intelligence*, pages 485–492.

[Thies and Amarasinghe, 2010] Thies, W. and Amarasinghe, S. (2010). An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376, New York, NY, USA. ACM.

[Thiffault et al., 2004] Thiffault, C., Bacchus, F., and Walsh, T. (2004). Solving Non-clausal Formulas with DPLL Search. In *Principles and Practices of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer.