

# On the Relationship Between Lexical Semantics and Syntax for the Inference of Context-Free Grammars

Tim Oates, Tom Armstrong, Justin Harris and Mark Nejman

Department of Computer Science and Electrical Engineering  
University of Maryland Baltimore County, Baltimore, MD 21250  
{oates,arm1,jharri11,mnejmal}@umbc.edu

## Abstract

Context-free grammars cannot be identified in the limit from positive examples (Gold 1967), yet natural language grammars are more powerful than context-free grammars and humans learn them with remarkable ease from positive examples (Marcus 1993). Identifiability results for formal languages ignore a potentially powerful source of information available to learners of natural languages, namely, *meanings*. This paper explores the learnability of syntax (i.e. context-free grammars) given positive examples and knowledge of lexical semantics, and the learnability of lexical semantics given knowledge of syntax. The long-term goal is to develop an approach to learning both syntax and semantics that bootstraps itself, using limited knowledge about syntax to infer additional knowledge about semantics, and limited knowledge about semantics to infer additional knowledge about syntax.

## Introduction

Learning formal languages from positive examples is a hard problem. If the language to be learned is finite and every string in the language is guaranteed to be presented at least once, the learner can memorize the strings that it sees. However, if the language can contain infinitely many strings and the learner is given a finite amount of time, then simple memorization will not work. The learner must generalize from the examples it sees. This is precisely the problem facing children learning their native language. For many important classes of languages, including regular and context-free, there are infinitely many languages that include any given set of positive examples. The challenge facing the learner is to avoid overgeneralization, to avoid choosing a language that contains the examples seen thus far and is a superset of the target language.

There is a handful of methods that are typically employed to avoid overgeneralization and establish learnability. If the learner knows that a given string is not in the target language, any overly general languages that include the string can be ruled out. A large number of learnability results reported in the literature depend on negative examples (Angluin 1987; Gold 1967; Oncina & Garcia 1992). In the absence of negative examples, learnability can follow from restrictions on

the class of languages from which the target is drawn (Angluin 1982; Oates, Desai, & Bhat 2002; Koshiba, Mäkinen, & Takada 2000) or on the method for selecting examples (Denis, D'Halluin, & Gilleron 1996; Li & Vitányi 1991). Establishing learnability for context-free languages is more difficult than for regular languages. A number of results on the learnability of context-free languages from positive examples require each string to be paired with its unlabeled derivation tree (Carrasco, Oncina, & Calera 1998; Oates, Desai, & Bhat 2002; Sakakibara 1992). Such a pair is called a *positive structural example*. An unlabeled derivation tree is a parse tree in which the non-terminal labels on interior nodes are not present.

The work reported in this paper is part of a larger effort aimed at understanding what is required to allow a robot to learn fragments of natural languages given qualitatively the same inputs available to children - utterances and sensory information about the physical context in which the utterances are heard. Children rarely receive negative examples (i.e. syntactically incorrect utterances marked as such), pay little attention when they do, and have only a few very weak proxies for negative examples (e.g. failure of a caregiver to respond to an utterance as expected) (Marcus 1993). Therefore, we focus on learning exclusively from positive examples, and on context-free languages because they are sufficiently expressive to represent large fragments of natural languages.

The key to our approach is to recognize that, in addition to hearing utterances, children have sensory access to the world around them. In particular, we assume that the lexicon contains word/meaning pairs, where the meanings have been established, for example, by some associative learning algorithm (Oates 2001; Roy 1999). The purpose of natural language communication is to share meanings. Therefore, the goal of the learner is to efficiently arrive at a compact representation of the language that makes it possible to determine which strings are in the language and what their meanings are.

According to Frege's principle of compositionality (Frege 1879), the meanings of phrases and sentences must be a function solely of the meanings of the words involved. Suppose rules for syntactic composition (i.e. productions in the grammar) are in a one-to-one correspondence with rules for semantic composition. For example, if the grammar for En-

glish has a syntactic production of the form  $S \rightarrow NP VP$ , then there must be a corresponding semantic rule that says how to combine the meaning of an NP with the meaning of a VP to get the meaning of an S. Therefore, each syntactic parse tree has a structurally equivalent semantic parse tree. More importantly for this paper, each semantic parse tree, which specifies how the meanings of words and phrases are combined to yield the meaning of a sentence, has a structurally equivalent syntactic parse tree. We use lexical semantics to generate possible semantic parse trees, i.e. those that yield a coherent semantics for the sentence, and use these trees as input to an algorithm for learning context-free grammars from positive structural examples.

This paper explores the utility of lexical semantics with respect to inferring context-free languages from positive examples consisting solely of strings. Although unlabeled derivation trees are used by the learning algorithm, they are not part of its input. Rather, they are derived from lexical semantics. The main contributions are (1) the specification of a class of context-free grammars that can be learned from positive string examples and lexical semantics, and (2) a result on the learnability of lexical semantics given a context-free grammar. Our ultimate goal is to combine these two results into a system that uses lexical knowledge obtained via associative learning to infer some knowledge of syntax, and uses knowledge of syntax to infer additional lexical knowledge. Over time, such a system could bootstrap itself to increasingly complete knowledge of both syntax and semantics.

## Background

This section reviews one common algorithm for inferring context-free languages from positive structural examples and the  $\lambda$ -calculus, which is used to represent lexical and compositional semantics.

### Inferring Grammars from Positive Structural Examples

A *context-free grammar* (CFG) is a four-tuple  $\{N, \Sigma, P, S\}$  where  $N$  is a finite set of non-terminals,  $\Sigma$  is a finite set of terminals,  $P$  is a finite set of productions, and  $S \in N$  is the start symbol.  $N$  and  $\Sigma$  are disjoint. A CFG is in Chomsky Normal Form (CNF) if all productions are of the form  $X \rightarrow YZ$  or  $X \rightarrow \sigma$ , for  $X, Y, Z \in N$  and  $\sigma \in \Sigma$ . For the remainder of this paper, the word *grammar* refers to a CFG in CNF.

Let  $L(G)$  denote the language of grammar  $G$ . An unlabeled derivation tree (UDT) for  $s \in L(G)$  is the derivation tree for  $s$  with the non-terminal labels on the interior nodes removed. UDTs can be represented as parenthesized strings. For example,  $(a(bc))$  and  $((ab)c)$  are possible UDTs for string  $abc$ . In the first UDT, the string  $bc$  is generated by some non-terminal, say,  $X$ , and the string  $aX$  is generated by the start symbol.

A grammar is *reversible* if it is reset-free and invertible. A grammar is *reset-free* if  $A \rightarrow XY$  and  $B \rightarrow XY$  in  $P$  implies  $A = B$ . A grammar is *invertible* if  $X \rightarrow AY$  and  $X \rightarrow BY$  in  $P$  implies  $A = B$ . Sakakibara's algorithm (Sakakibara 1992) takes a set of positive structural examples

Table 1: A simple lexicon with semantics and semantic types.

Lexeme	$\lambda$ -calculus expression	Semantic type
Nathaniel	NATHANIEL	e
Isabel	ISABEL	e
loves	$\lambda x \lambda y . \text{LOVES}(y,x)$	$\langle e, \langle e,t \rangle \rangle$
rests	$\lambda x . \text{RESTS}(x)$	$\langle e,t \rangle$

and returns the reversible grammar that has the smallest language of all reversible grammars that contain the examples. The algorithm labels the root of each UDT with the start symbol and assigns unique labels to all interior nodes. The union of the productions in the newly labeled trees serve as the initial grammar. Since the generating grammar is known to be reversible, pairs of non-terminals that violate the reset-free and invertible properties are merged until no such pair exists, at which point the algorithm terminates. Let  $GI(E)$  be the output of this algorithm on input  $E$ .

### Montague Grammar and the $\lambda$ -Calculus

The first formal treatment of natural language semantics is typically attributed to Richard Montague. In three of his last papers, Montague introduced an intensional logic for representing the semantics of natural language strings (Dowty, Wall, & Peters 1981; Partee & Hendriks 1996). The semantics of words are represented as expressions in the  $\lambda$ -calculus which are (typically) functions. These functions take arguments and return values, both of which are also expressions in the  $\lambda$ -calculus. The semantics of strings are computed bottom-up by applying the  $\lambda$ -calculus expressions of lower-level constituents to one another. Table 1 shows a small lexicon in the Montague framework that will serve as a running example throughout the remainder of this section.

The  $\lambda$ -calculus is useful for exploring function descriptions and function application. It is a model suited for functional programming used in some languages (e.g. Lisp). Function descriptions are created by performing *lambda abstractions*. In lambda abstractions a variable argument is given scope and bound to a lambda operator. These anonymous functions can then be applied to other expressions.

Three rules are used to reduce  $\lambda$ -calculus expressions:  $\alpha$ -,  $\beta$ - and  $\eta$ - reductions. A  $\lambda$ -calculus expression, when fully reduced, will be a first-order logic expression. Let ' $[x/y]z$ ' denote replacing instances of  $x$  with  $y$  in expression  $z$ . For any two arbitrary expression, a lambda operator may be bound to the same named variable in both. Performing an  $\alpha$ -reduction on one of the expressions results in a renaming of the variable to a unique name. When an expression is functionally applied to an argument expression it is called  $\beta$ -reduction. The first lambda-operator and variable are stripped off and the argument is substituted for each instance of the variable. The final operation removes lambda-operators which are bound to variables no longer found in the body of the expression.

For our purposes, and generally, the processes of abstraction and  $\beta$ -reduction are the most useful tools. Abstractions

(i.e. functions) provide the representation for lexical semantics, and  $\beta$ -reductions are used to compute compositional semantics.

The following example shows the derivation of the meaning (an expression in first-order logic) of the UDT (*Nathaniel (loves Isabel)*). Note that there is ambiguity for any given pair inside a set of parentheses as to which is the function and which is the argument. For now, we assume the order of application is given and it is as follows: *loves* applies to *Isabel* and *loves Isabel* applies to *Nathaniel*.

1. (Nathaniel (loves Isabel)) = (NATHANIEL ( $\lambda x \lambda y .$  LOVES( $y,x$ ) ISABEL))
2. (NATHANIEL ( $\lambda x \lambda y .$  LOVES( $y,x$ ) ISABEL))  $\Rightarrow_{\beta}$  (NATHANIEL  $\lambda y .$  loves( $y,ISABEL$ ))
3. (NATHANIEL  $\lambda y .$  LOVES( $y,ISABEL$ ))  $\Rightarrow_{\beta}$  loves(NATHANIEL,ISABEL)

From ordered function application we arrive at the standard English meaning of the string.

### Semantic Types

Every  $\lambda$ -calculus expression has a semantic type, and these types constrain how  $\lambda$ -calculus expressions compose with one another. Types are either basic or complex. Basic types are defined as  $e$ , entities (e.g. Nathaniel, Isabel), or  $t$ , truth values (e.g. true, false). Complex types, which represent types of functions, are defined recursively: if  $\alpha$  and  $\beta$  are types, then  $\langle \alpha, \beta \rangle$  is a type. Semantic types dictate the direction of function application (i.e. when applying a function of type  $\langle \gamma, \delta \rangle$ , the argument to the function must have type  $\gamma$ ). Take from Table 1 the types for *Isabel* and *rests*,  $e$  and  $\langle e,t \rangle$  respectively. The order of application must be from *rests* to *Isabel* to properly compose.

Each expression in the  $\lambda$ -calculus, being a function description, has a parameter and return value. The semantic type of a function can be represented by  $\langle \alpha, \beta \rangle$  where  $\alpha$  and  $\beta$  are the semantic types of the parameter and return value respectively. There exists a many-to-one mapping between  $\lambda$ -calculus expressions and semantic types (e.g. *Nathaniel* and *Isabel* are both of type  $e$ ).

### Learning Syntax Using Lexical Semantics

In this section we formally explore the utility of UDTs derived from semantic parse trees with respect to grammatical inference.

### Lexical Semantics Reducing Numbers of Parse Trees

How many semantically valid UDTs could there be for a string? For a string of length  $n$ , the number of possible UDTs is equal to the number of possible binary bracketings of that string. Catalan, an 18th century Belgian mathematician, posed a problem for finding the number of ways to compose factors in calculating products (Weisstein 2003). This problem is equivalent to finding the number of binary bracketings. Formally,  $C(n)$ , the  $n^{th}$  Catalan number, is  $\frac{(2n)!}{(n!(n+1)!)}$ . For strings of length  $n$ , the number of binary bracketings is equal to the  $n - 1^{th}$  Catalan number.

Table 2: A simple lexicon with semantic types.

Lexeme	Semantic type
the	$\langle \langle e,t \rangle, \langle \langle e,t \rangle, t \rangle \rangle$
dog	$\langle e,t \rangle$
cat	$\langle e,t \rangle$
hates	$\langle \langle \langle e,t \rangle, t \rangle, \langle e,t \rangle \rangle$

The number of possible parse trees for a string grows exponentially in the length of the string. From a learnability perspective, dealing with large numbers of trees for a single string is not easily handled. Given lexical types for terminal nodes, performing type checking on possible trees will rule out a subset of them as compositionally invalid. Type checking succeeds if after type composition over a tree for a string the root node has semantic type  $t$ . Bracketed strings of this form are called *semantically valid*.

Take, for example, the string  $s = \textit{the cat hates the dog}$  generated by some grammar. Given the length of  $s$  is 5,  $C(4) = 14$ . Of the 14 possible parses, 3 are valid in terms of semantic composition: ((the cat) (hates (the dog))), ((the cat) hates) (the dog)) and (((the cat) hates) the) dog). As shown above, a first order logic expression for the string can then be constructed following the application orderings for each of the three trees. In one case the expression denotes *the cat is the hater of the dog* and in others *the dog is the hater of the cat*. The following is a type-checking evaluation for one possible parse tree.

Take (the cat) in type form as  $\langle \langle e,t \rangle, \langle \langle e,t \rangle, t \rangle \rangle \rightarrow \langle e,t \rangle$  from Table 2. Performing the application in the order given (the cat) has type  $\langle \langle e,t \rangle, t \rangle$ . The type for (the dog) is clearly equivalent. The next bracket combines (hates (the dog)), thus the type is  $\langle \langle \langle e,t \rangle, t \rangle, \langle e,t \rangle \rangle \rightarrow \langle \langle e,t \rangle, t \rangle$  resulting in a type of  $\langle e,t \rangle$ . The final combination is between the types of (the cat) and (hates (the dog)) or  $\langle \langle \langle e,t \rangle, t \rangle \rightarrow \langle e,t \rangle$ . The root results in a  $t$  for this bracketing of the string.

### Computing Semantically Valid Parses

For string  $s$ , type  $\alpha$ , and grammar  $G$ , let  $T(s)$ ,  $T_{\alpha}(s)$ , and  $T_G(s)$  denote sets of unlabeled derivation trees.  $T_G(s)$  is the set of trees that are possible given grammar  $G$ .  $T(s)$  is the union of  $T_G(s)$  for all grammars that generate  $s$ . Assuming that lexical semantics are given,  $T_{\alpha}(s)$  is the restriction of  $T(s)$  to those trees whose root have type  $\alpha$ . To use semantic parse trees to learn syntax, it must be possible to compute  $T_{\alpha}(s)$  efficiently. We show that this is the case in what follows, starting from an efficient algorithm for computing  $|T(s)|$ .

The recurrence below can be used to compute  $|T(s)|$  bottom up.  $M$  is a zero-indexed  $n \times n$  array, where  $|s| = n$ , and  $M[i, j]$  is  $|T(s[i \dots j])|$ , i.e. the number of possible UDTs for the substring of  $s$  ranging from position  $i$  to position  $j$ . Therefore,  $|T(s)| = M[0, n - 1]$ .

$$M[i, i] = 1$$

$$M[i, j] = \sum_{l=2}^n \sum_{p=0}^{n-l} \sum_{q=0}^{l-2} M[p, p+q] * M[p+q+1, p+l-1]$$

Assuming the grammar is in CNF, the only way to generate a terminal is via a production of the form  $X \rightarrow \sigma$ . Therefore, there is only one unlabeled derivation tree for each substring of  $s$  of length 1 and  $M[i, i] = 1$  for all  $i$ . All other entries are computed bottom-up via the second line in the recurrence. The first sum is over substring lengths,  $l$ , ranging from 2 to  $n$ . The second sum is over all starting positions,  $p$ , in  $s$  of strings of length  $l$ . Together,  $l$  and  $p$  identify a substring of  $s$ , namely,  $s[p \dots p+l-1]$  that must have been generated by a production of the form  $X \rightarrow YZ$ . The third sum is over all ways in which  $Y$  and  $Z$  can divide up that substring. For any such division, the number of trees for the substring is the product of the number of trees for the part  $Y$  generates and the number of trees for the part that  $Z$  generates.

Given lexical semantics, it is possible to compute  $T_\alpha(s)$  using the recurrence below. Rather than  $M$  being an  $n \times n$  array, it is an  $n \times n \times m$  array where  $m$  is the number of distinct types that can label interior nodes in a semantic parse tree. Let  $\delta(expr)$  take on value 1 if  $expr$  is true, and 0 otherwise. Let  $\text{typeof}(expr)$  return the type (an integer between 0 and  $m-1$ ) of  $expr$ . Let @ denote function application.

$$\begin{aligned} M[i, i, t] &= \delta(\text{typeof}(s[i \dots i]) = t) \\ M[i, j, t] &= \sum_{t_1, t_2} \sum_{l=2}^n \sum_{p=0}^{n-l} \sum_{q=0}^{l-2} M[p, p+q, t_1] * \\ &\quad M[p+q+1, p+l-1, t_2] * \\ &\quad \delta(\text{typeof}(t_1 @ t_2) = t \vee \\ &\quad \text{typeof}(t_2 @ t_1) = t) \end{aligned}$$

There is still only one unlabeled derivation tree for each substring of length 1, but the type of that tree is the type of the lambda expression for the meaning of the terminal. The outer sum in the second line of the recurrence is over all  $m^2$  pairs of types. Given that  $s[p \dots p+l-1]$  will be split at offset  $q$  (as established by the inner three sums), that split will yield a valid semantic parse only if the types of the two halves are compatible, i.e. if one can be applied to the other. If  $t_1$  is the type of the left half and  $t_2$  is the type of the right half, and applying  $t_1$  to  $t_2$  or applying  $t_2$  to  $t_1$  yields type  $t$ , then  $M[i, j, t]$  is updated.

The recurrence above can be computed in  $O(m^2 n^3)$  time. Note that  $m$  will typically be a small constant because interior nodes can only be labeled by lexical types and types out of which lexical types are composed. Also, it is trivial to augment the computation so that  $M$  can be used to extract all semantically valid parses of a given type. This is done by keeping a list of split points,  $q$ , for each  $(i, j, t)$  that yield the desired type (i.e.  $t$ ). This increases the storage required by a multiplicative factor of  $n$ .

### Using Semantically Valid Parse Trees to Learn Syntax

Our goal is to use lexical semantics and positive string examples to obtain positive structural examples that can be used

to learn syntax. Clearly, if for every string in  $L(G)$  there is a single semantically valid parse, then learning is straightforward. Later in this section we define a class of grammars for which this is the case. What happens, though, if there are multiple semantically valid parses for some string(s) in  $L(G)$ ?

Let  $E = \{s_1, s_2, \dots\}$  be a set of strings. Let  $T(E) = \cup_{s_i \in E} T(s_i)$ , and let  $T_G(E)$  and  $T_\alpha(E)$  be defined similarly. Now consider grammar  $G$  for which  $L(G) = \{xcd, xab, ycd, zab\}$  and  $T_G(L(G)) = \{((xc)d), ((xa)b), (y(cd)), (z(ab))\}$ . Suppose the type of  $a$  and  $c$  is  $t$ , and the type of all other lexical items is  $< t, t >$ . Then the semantically valid parses of the strings in  $L(G)$  are as follows:

$$T_t(L(G)) = \{((xc)d), ((xa)b), (y(cd)), (z(ab)), (x(cd)), (x(ab)), ((yc)d), ((za)b)\}$$

For a reset-free grammar to generate these UDTs, the non-terminal that generates  $cd$  must also generate  $ab$  due to the trees  $(x(ab))$  and  $(x(cd))$ . Coupled with the fact that  $(y(cd))$  and  $(z(ab))$  are in  $L$ , the grammar must also generate  $(y(ab))$  and  $(z(cd))$ , neither of which are in  $L(G)$ . That is, learning a grammar based on all semantically valid parse trees can, in some cases, lead to overgeneralization.

Even so, given a sample of strings,  $E$ , from the language of some grammar,  $G$ , we can establish a few useful properties of the grammar learned from the structural examples contained in  $T_t(E)$ . For example, the language of that grammar will always be a subset of the language of the grammar learned from  $E$  with the true UDTs. Somewhat more formally:

$$L(GI(T_G(E))) \subseteq L(GI(T_t(E)))$$

To see that this is true, first note that the output of  $GI$  is independent of the order in which merges occur. It then suffices to show that the above holds for a single merge order. Suppose the merges performed by  $GI$  on  $T_t(E)$  are precisely those performed on  $T_G(E)$  up to the point where  $GI(T_G(E))$  would terminate. If  $GI(T_t(E))$  were to stop at this point, the language of the resulting grammar would be precisely  $L(GI(T_G(E)))$  because it would contain all of the productions in  $GI(T_G(E))$  plus productions that produce all and only the trees in  $T_t(E) - T_G(E)$ , and these trees correspond to strings in  $E$  that are in  $L(GI(T_G(E)))$ . Additional merging required for  $GI(T_t(E))$  to terminate can only add strings to the resulting language, so  $L(GI(T_G(E))) \subseteq L(GI(T_t(E)))$ .

### Lexically Unambiguous Parses

The previous section established that there exist grammars for which the language inferred using all semantically valid parses of the training examples is a superset of the language inferred using the true semantic parses of the training examples. This section defines a class of grammars for which every string in the language of such a grammar has a single semantically valid parse. For grammars in this class, lexical semantics ensure learnability.

For a given grammar  $G$ , let  $\text{right}(X)$  be the set of terminals and non-terminals that can occur in the rightmost position of any string derivable in one or more steps from  $X$ .

Let  $left(X)$  be defined analogously. Let  $canapply(A, B)$  return true iff either  $A$  is of the type  $\langle \alpha, \beta \rangle$  and  $B$  is of the type  $\alpha$  (i.e.  $A$  can be applied to  $B$ ) or  $A$  is of the type  $\gamma$  and  $B$  is of the type  $\langle \gamma, \delta \rangle$  (i.e.  $B$  can be applied to  $A$ ).

**Theorem 1** *If for every production of the form  $X \rightarrow YZ$  in grammar  $G$  neither condition 1 nor condition 2 below holds, then there is exactly one semantically valid parse of every string in the language generated by  $G$ .*

1.  $\exists A \in right(Y) \cup \{Y\} \wedge \exists B \in left(Z)$  s.t.  $canapply(A, B)$
2.  $\exists A \in right(Y) \wedge \exists B \in left(Z) \cup \{Z\}$  s.t.  $canapply(A, B)$

**Proof:** The proof will be by induction on the height of the derivation tree that generates the string. Because  $G$  is in CNF, all productions are of the form  $X \rightarrow YZ$  or  $X \rightarrow \sigma$ . Each non-terminal is generated by a tree of height one, and there is only one way to semantically parse any given non-terminal, so strings derived by a tree of height one have a single semantically valid parse. This is the base case. The inductive assumption is that all strings generated by a tree of height no more than  $h$  have a single semantically valid parse.

Suppose some string has a derivation tree of height  $h + 1$  and the string has multiple semantically valid parses. Let  $X$  be the non-terminal that roots the derivation tree, and let  $Y$  and  $Z$  be the non-terminals that root its left and right subtrees (i.e. the derivation tree was produced by expanding  $X$  to  $YZ$  via production  $X \rightarrow YZ$ ). The subtrees rooted by  $Y$  and  $Z$  can have height at most  $h$ , so by the inductive assumption the strings they generate have a single semantically valid parse.

Therefore, the only way for the string generated by  $X$  to have more than one semantically valid parse is for some non-terminal on the right edge of the tree generated by  $Y$  to have a type that can be applied to the type of some non-terminal on the left edge of the tree generated by  $Z$  (or vice versa). However, conditions 1 and 2 above explicitly disallow this, so there can be no string generated by a tree of height  $h + 1$  with multiple semantically valid parses. Inductively, the theorem holds for strings generated by derivation trees of any height, i.e. all strings.  $\square$

The theorem above says that there is a class of grammars for which all strings have a single semantically valid parse. Grammars in this class can be learned by any algorithm that learns CFGs from unlabeled derivation trees.

## Semantic Type Inference

Until now we have assumed that the learning algorithm was given lexical semantics for all terminals. Now let us consider the case where we are given the UDTs and order of application for internal nodes for a sample of strings in a grammar. From this information we will demonstrate how all semantic types can be inferred using a minimal subset of semantic types.

Let  $X \rightarrow Y Z$  be a production for which the meaning of  $X$  is obtained by applying the meaning of  $Y$  to the meaning of  $Z$ . Because the production obeys type constraints, if  $X$  is

of type  $\langle \alpha, \beta \rangle$  then  $Y$  must be of type  $\alpha$  and  $Z$  must be of type  $\beta$ . Types can be inferred for a given production in CNF in two ways. If the type of  $Y$  is known, the types of  $X$  and  $Z$  can be inferred. If the types of  $X$  and  $Z$  are known, the type of  $Y$  can be inferred. We assume that the start symbol has the same type for all strings. Consider the following grammar assuming left to right function application for all productions (e.g. assuming NP applies to VP and not vice versa):

$$\begin{array}{l} S \rightarrow NP VP \\ NP \rightarrow DET N \\ VP \rightarrow TV NP \\ VP \rightarrow IV \end{array}$$

The start symbol,  $S$ , has semantic type of  $t$ . For this grammar, if we know the types of the elements of  $\{N \cup \Sigma\}$  that are never used as functors in any production, we can infer the types of all other elements of  $\{N \cup \Sigma\}$ . For the above grammar the known types are of  $S, VP, N$  and  $IV$ . Knowing  $S$  and  $VP$  allows inference of  $NP$ , which then allows inference of  $DET$  using  $N$ , and inference of  $TV$  using  $VP$  and  $NP$ .

We say that non-terminal  $Y$  *never applies* if it never occurs in a production  $X \rightarrow YZ$  or  $X \rightarrow ZY$  for which the meaning of  $Y$  is applied to the meaning of  $Z$  to obtain the meaning of  $X$ . The following theorem defines a subset of types that need to be known to ensure that all types are inferable.

**Theorem 2** *For any context-free grammar  $G$  in Chomsky Normal Form, given the types of all non-terminals that never apply, the types of all other non-terminals can be inferred.*

**Proof:** Let  $G^*$  be the directed graph created from grammar  $G$  as follows. For each production of the form  $X \rightarrow YZ$  add nodes labeled  $X, Y,$  and  $Z$  to  $G^*$ . If  $Y$  applies to  $Z$ , add edges from node  $Y$  to both node  $X$  and  $Z$ . If  $Z$  applies to  $Y$ , add edges from node  $Z$  to both node  $X$  and  $Y$ . Each node is marked as either known or unknown, depending on whether the type of the associated non-terminal is known. Nodes with out-degree zero correspond to non-terminals that never apply, and are therefore assumed to be known. All other nodes are initially marked unknown.

If all of the neighbors of a node are marked known, then its type can be inferred and the node marked known. This inference/markings step can be repeated until no new nodes are marked known. Because the number of unknown nodes decreases by at least one on each step, or the algorithm terminates, there can be no more than  $O(|N|)$  iterations, each of which might need to do  $O(|P|)$  work.

Suppose there exists a grammar  $G$  for which all nodes in  $G^*$  are not marked known when the algorithm terminates. Let  $X$  be the non-terminal associated with this node. If the type of  $X$  cannot be inferred, then the type of at least one of its neighbors, call it  $Y$ , cannot be inferred. That is, there is path of length 1 from  $X$  to a node whose type cannot be inferred. Likewise, because the type of  $Y$  cannot be inferred, the type of at least one of its neighbors cannot be inferred, and this node lies on a path of length 2 from  $X$ . Inductively, there must be a node at the end of a path of length  $n$  for all

$n \geq 0$  from  $X$  whose type cannot be inferred. However, because a non-terminal's type cannot be defined (even partially) in terms of itself, all paths must terminate in a node with out-degree zero in  $O(|N|)$  steps, and the types of all such nodes are known. This is a contradiction. Therefore, the theorem holds.  $\square$

The importance of theorem 2 is that words with types that never apply are typically those that refer to perceptually concrete aspects of the environment, such as nouns. That is, it is possible that an embedded learner might associatively learn the types of these words (Oates 2001; Roy 1999), and then use knowledge of syntax to infer the semantic types of all other words in the lexicon.

## Conclusion

This paper represents the first results of our inquiry into the relationship between meanings and learnability for context-free grammars. Theorem 1 established that there exists a class of grammars whose syntax can be learned from positive string examples and lexical semantics. Theorem 2 established that it is possible to infer lexical semantics given a grammar's syntax and a small number of lexical types.

The work most similar to ours reported in the literature is that of Tellier and her colleagues (Tellier 1998; Dudau-Sofronie, Tellier, & Tommasi 2001) who are also interested in the role of lexical semantics in grammatical inference. They propose an algorithm for inferring rigid categorial grammars given strings and their meanings, though the algorithm has exponential complexity and it is unclear (i.e. there is no proof) whether it converges to the target grammar or some grammar containing the target.

Future work will proceed in a number of directions. Ultimately, we want to develop algorithms that will iteratively use incomplete lexical knowledge to infer grammar fragments, and then use these fragments to infer more lexical knowledge. The goal is to have a learner that can provably converge on the correct lexicon and grammar by bootstrapping from some small amount of knowledge about lexical semantics obtained via associative learning.

An important result that we're currently working toward is an algorithm for inferring syntax when there are multiple semantically valid parse trees for one or more strings. One possible approach is to compute the corresponding semantics of each parse and use the fact that the learner is embedded in an environment to determine which parse is correct, i.e. which one refers to the current state of the world. An alternative approach might involve noticing when a merge would include strings in the grammar that are not observed.

## References

- Angluin, D. 1982. Inference of reversible languages. *Journal of the Association for Computing Machinery* 29:741–765.
- Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75:87–106.
- Carrasco, R. C.; Oncina, J.; and Calera, J. 1998. Stochastic inference of regular tree languages. *Lecture Notes in Computer Science* 1433:187–198.
- Denis, F.; D'Halluin, C.; and Gilleron, R. 1996. PAC learning with simple examples. In *Symposium on Theoretical Aspects of Computer Science*, 231–242.
- Dowty, D. R.; Wall, R. E.; and Peters, S. 1981. *Introduction to Montague Semantics*. Dordrecht: Reidel.
- Dudau-Sofronie, D.; Tellier, I.; and Tommasi, M. 2001. From logic to grammars via types. In Popelínský, L., and Nepil, M., eds., *Proceedings of the 3rd Workshop on Learning Language in Logic*, 35–46.
- Frege, F. L. G. 1879. Begriffsschrift. In van Heijenoort, J., ed., *Frege and Godel: Two Fundamental Texts in Mathematical Logic (1970)*. Cambridge, MA: Harvard University Press.
- Gold, E. M. 1967. Language identification in the limit. *Information and Control* 10:447–474.
- Koshiba, T.; Makinen, E.; and Takada, Y. 2000. Inferring pure context-free languages from positive data. *Acta Cybernetica* 14(3):469–477.
- Li, M., and Vitanyi, P. M. B. 1991. Learning simple concepts under simple distributions. *SIAM Journal of Computing* 20(5):911–935.
- Marcus, G. F. 1993. Negative evidence in language acquisition. *Cognition* 46(1):53–85.
- Oates, T.; Desai, D.; and Bhat, V. 2002. Learning k-reversible context-free grammars from positive structural examples. In *Proceedings of the Nineteenth International Conference on Machine Learning*.
- Oates, T. 2001. *Grounding Knowledge in Sensors: Unsupervised Learning for Language and Planning*. Ph.D. Dissertation, The University of Massachusetts, Amherst.
- Oncina, J., and Garcia, P. 1992. Inferring regular languages in polynomial updated time. In de la Blanca, N. P.; Sanfeliu, A.; and Vidal, E., eds., *Pattern Recognition and Image Analysis*. World Scientific.
- Partee, B., and Hendriks, H. 1996. Montague grammar. In van Benthem, J., and ter Meulen, A., eds., *Handbook of Logic and Language*. Amsterdam: Elsevier Science and The MIT Press. 5–91.
- Roy, D. 1999. *Learning Words from Sights and Sounds: a Computational Model*. Ph.D. Dissertation, MIT.
- Sakakibara, Y. 1992. Efficient learning of context-free grammars from positive structural examples. *Information and Computation* 97:23–60.
- Tellier, I. 1998. Meaning helps learning syntax. *Lecture Notes in Computer Science* 1433.
- Weisstein, E. W. 2003. Catalan numbers. In *Eric Weisstein's World of Mathematics*.