A Uniform Method of Grammar Extraction and Its Applications

Fei Xia and **Martha Palmer** and **Aravind Joshi**
Department of Computer and Information Science
University of Pennsylvania
Philadelphia PA 19104, USA
{fxia,mpalmer,joshi}@linc.cis.upenn.edu

Abstract

Grammars are core elements of many NLP applications. In this paper, we present a system that automatically extracts lexicalized grammars from annotated corpora. The data produced by this system have been used in several tasks, such as training NLP tools (such as Supertaggers) and estimating the coverage of hand-crafted grammars. We report experimental results on two of those tasks and compare our approaches with related work.

1 Introduction

There are various grammar frameworks proposed for natural languages. We take Lexicalized Tree-adjoining Grammars (LTAGs) as representative of a class of lexicalized grammars. LTAGs (Joshi et al., 1975) are appealing for representing various phenomena in natural languages due to its linguistic and computational properties. In the last decade, LTAG has been used in several aspects of natural language understanding (e.g., parsing (Schabes, 1990; Srinivas, 1997), semantics (Joshi and Vijay-Shanker, 1999; Kallmeyer and Joshi, 1999), and discourse (Webber and Joshi, 1998)) and a number of NLP applications (e.g., machine translation (Palmer et al., 1998), information retrieval (Chandrasekar and Srinivas, 1997), and generation (Stone and Doran, 1997; McCoy et al., 1992). This paper describes a system that extracts LTAGs from annotated corpora (i.e., Treebanks).

There has been much work done on extracting Context-Free grammars (CFGs) (Shirai et al., 1995; Charniak, 1996; Krotov et al., 1998). However, extracting LTAGs is more complicated than extracting CFGs because of the differences between LTAGs and CFGs. First, the primitive elements of an LTAG are lexicalized tree structures (called *elementary trees*), not context-free rules (which can be seen as trees with depth one). Therefore, an LTAG extraction algorithm needs to examine a larger portion of a phrase structure to build an elementary tree. Second, the composition operations in LTAG are substitution (same as the one in a CFG) and adjunction. It is the operation of adjunction that distinguishes LTAG from all other formalisms. Third, unlike in CFGs, the parse trees (also known as *derived trees* in the LTAG) and the derivation trees (which describe how elementary trees are combined to form parse trees) are different in the LTAG formalism in the sense that a parse tree can be produced by several distinct derivation trees. Therefore, to provide training data for statistical LTAG parsers, an LTAG extraction algorithm should also build derivation trees.

For each phrase structure in a Treebank, our system creates a fully bracketed phrase structure, a set of elementary trees and a derivation tree. The data produced by our system have been used in several NLP tasks. We report experimental results on two of those applications and compare our approaches with related work.

2 LTAG formalism

The primitive elements of an LTAG are elementary trees (*etrees*). Each *etree* is associated with a lexical item (called the *anchor* of the tree) on its frontier. We choose LTAGs as our target grammars (i.e., the grammars to be extracted) because LTAGs possess many desirable properties, such as the Extended Domain of Locality, which allows the encapsulation of all arguments of the anchor associated with an *etree*. There are two types of *etrees*: initial trees and auxiliary trees. An auxiliary tree represents recursive structure and has a unique leaf node, called the *foot* node, which has the same syntactic category as the root node. Leaf nodes other than anchor nodes and foot nodes are *substitution* nodes. *Etrees* are combined by two operations: substitution and adjunction, as in Figure 1 and 2. The resulting structure of the combined *etrees* is called a *derived tree*. The combination process is expressed as a *derivation tree*.

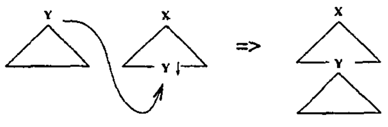


Figure 1: The substitution operation

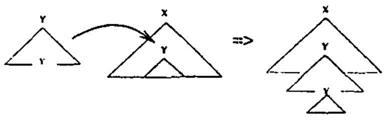


Figure 2: The adjunction operation

Figure 3 shows the *etrees*, the derived tree, and the derivation tree for the sentence *underwriters still draft policies*. Foot and substitution nodes are marked by $*$, and $\downarrow$, respectively. The dashed and solid lines in the derivation tree are for adjunction and substitution operations, respectively.

3 System Overview

We have built a system, called LexTract, for grammar extraction. The architecture of LexTract is shown in Figure 4 (the parts that will be discussed in this paper are in bold). The core of LexTract is an extraction algorithm that takes a Treebank sentence such as the one in Figure 5 and produces the trees (elementary trees, derived trees and derivation trees) such as the ones in Figure 3.

3.1 The Form of Target Grammars

Without further constraints, the *etrees* in the target grammar could be of various shapes.

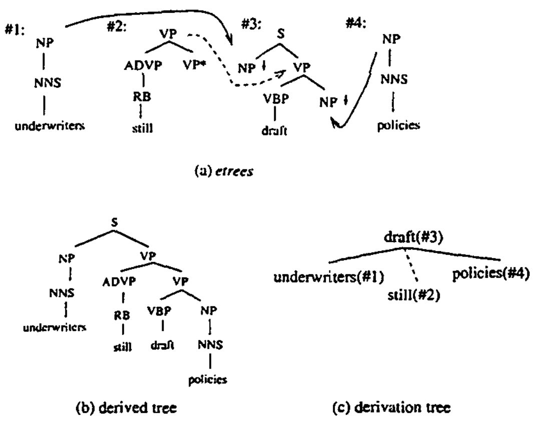


Figure 3: *Etrees*, derived tree and derivation tree for *underwriters still draft policies*

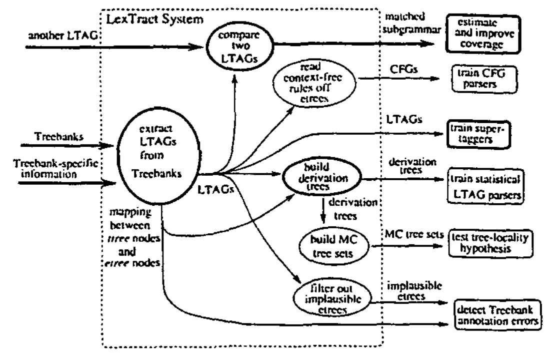

LexTract System
another LTAG
compare two LTAGs
matched subgrammar
estimate and improve coverage
read context-free rules off etrees
CFGs
train CFG parsers
LTAGs
train super-taggers
Treebanks
Treebank-specific information
extract LTAGs from Treebanks
LTAGs
build derivation trees
derivation trees
train statistical LTAG parsers
derivation trees
mapping between ttree nodes and etree nodes
build MC tree sets
MC tree sets
test tree-locality hypothesis
filter out implausible etrees
implausible etrees
detect Treebank annotation errors

Figure 4: Architecture of LexTract

```
((S (PP-LOC (IN at)
            (NP (NNP FNX)))
   (NP-SBJ-1 (NNS underwriters))
   (ADVP (RB still))
   (VP (VBP draft)
       (NP (NNS policies))
       (S-MNR
          (NP-SBJ (-NONE- *-1))
          (VP (VBG using)
             (NP
                (NP (NN fountain) (NNS pens))
                (CC and)
                (NP (VBG blotting) (NN papers))))))))
```

Figure 5: A Treebank example

Our system recognizes three types of relation (namely, predicate-argument, modification, and coordination relations) between the anchor of an *etree* and other nodes in the *etree*, and imposes the constraint that all the *etrees* to be extracted should fall into exactly one of the three patterns in Figure 6.

- The spine-etrees for predicate-argument relations. X^0 is the head of X^m and the anchor of the *etree*. The *etree* is formed by a spine $X^m \rightarrow X^{m-1} \rightarrow .. \rightarrow X^0$ and the arguments of X^0.
- The mod-etrees for modification relations. The root of the *etree* has two children, one is a foot node with the label W^q, and the other node X^m is a modifier

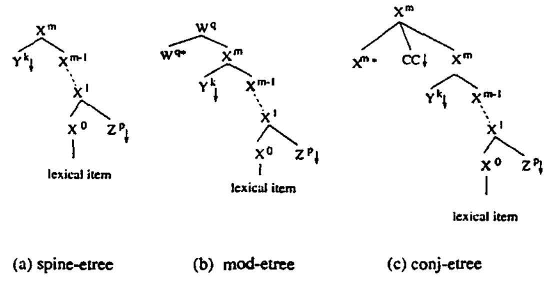


Figure 6: Three types of elementary trees in the target grammar

of the foot node. X^m is further expanded into a spine-etree whose head X^0 is the anchor of the whole mod-etree.

- The conj-etrees for coordination relations. In a conj-etree, the children of the root are two conjoined constituents and a node for a coordination conjunction. One conjoined constituent is marked as the foot node, and the other is expanded into a spine-etree whose head is the anchor of the whole tree.

Spine-etrees are initial trees, whereas mod-etrees and conj-etrees are auxiliary trees.

3.2 Treebank-specific information

The phrase structures in the Treebank (*ttrees* for short) are partially bracketed in the sense that arguments and modifiers are not structurally distinguished. In order to construct the *etrees*, which make such distinction, LexTract requires its user to provide additional information in the form of three tables: a Head Percolation Table, an Argument Table, and a Tagset Table.

A Head Percolation Table has previously been used in several statistical parsers (Magerman, 1995; Collins, 1997) to find heads of phrases. Our strategy for choosing heads is similar to the one in (Collins, 1997). An Argument Table informs LexTract what types of arguments a head can take. The Tagset Table specifies what function tags always mark arguments (adjuncts, heads, respectively). LexTract marks each sibling of a head as an argument if the sibling can be an argument of the head according to the Argument Table and none of the function tags of the sister indicates that it is an adjunct. For example, in Figure 5, the head of the root S is the verb *draft*, and the verb has two siblings: the noun phrase *policies* is marked as an argument of the verb because from the Argument Table we know that verbs in general can take an NP object; the clause is marked as a modifier of the verb because, although verbs in general can take a sentential argument, the Tagset Table informs LexTract that the function tag -MNR (*manner*) always marks a modifier.

3.3 Overview of the Extraction Algorithm

The extraction process has three steps: First, LexTract fully brackets each *ttree*; Second, LexTract decomposes the fully bracketed *ttree* into a set of *etrees*; Third, LexTract builds the derivation tree for the *ttree*.

```
((S (PP-LOC (IN at)
            (NP (NNP FNX)))
   (S  (NP-SBJ-1 (NNS underwriters))
      (VP  (ADVP (RB still))
           (VP  (VP  (VBP draft)
                     (NP (NNS policies)))
                (S-MNR
                  (NP-SBJ (-NONE- *-1))
                  (VP (VBG using)
                      (NP (NP (NN fountain)
                              (NP  (NNS pens)))
                          (CC and)
                          (NP (VBG blotting)
                              (NP  (NN papers)))))))) )))
```

Figure 7: The fully bracketed *ttree*

3.3.1 Fully bracketing *ttrees*

As just mentioned, the *ttrees* in the Treebank do not explicitly distinguish arguments and modifiers, whereas *etrees* do. To account for this difference, we first fully bracket the *ttrees* by adding intermediate nodes so that at each level, one of the following relations holds between the head and its siblings: (1) head-argument relation, (2) modification relation, and (3) coordination relation. LexTract achieves this by first choosing the head-child at each level and distinguishing arguments from adjuncts with the help of the three tables mentioned in Section 3.2, then adding intermediate nodes so that the modifiers and arguments of a head attach to different levels. Figure 7 shows the fully bracketed *ttree*. The nodes inserted by LexTract are in bold face.

3.3.2 Building *etrees*

In this step, LexTract removes recursive structures – which will become mod-etrees or conj-etrees – from the fully bracketed *ttrees* and builds spine-etrees for the non-recursive structures. Starting from the root of a fully bracketed *ttree*, LexTract first finds a unique path from the root to its head. It then checks each node *e* on the path. If a sibling of *e* in the *ttree* is marked as a modifier, LexTract marks *e* and *e*'s parent, and builds a mod-etree (or a conj-etree if *e* has another sibling which is a conjunction) with *e*'s parent as the root node, *e* as the foot node, and *e*'s siblings as the modifier. Next, LexTract creates a spine-etree with the remaining unmarked nodes on the path and their siblings. Finally, LexTract repeats this process for the nodes that are not on the path.

In Figure 8, which is the same as the one in Figure 7 except that some nodes are numbered and split into the top and bottom pairs,[1] the

[1]When a pair of *etrees* are combined during parsing,

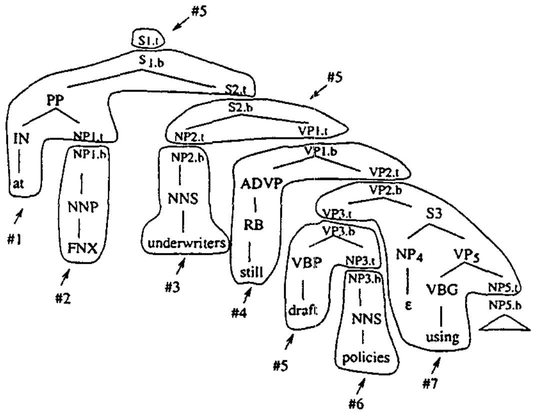


Figure 8: The extracted *etrees* can be seen as a decomposition of the fully bracketed *ttree*

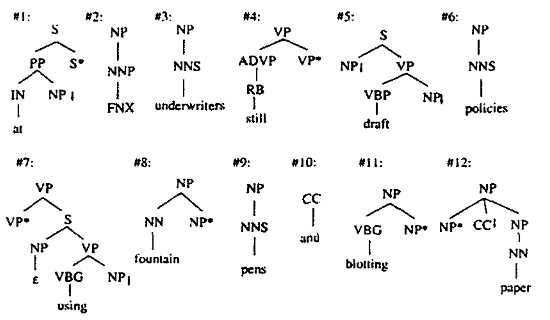


Figure 9: The extracted *etrees* from the fully bracketed *ttree*

path from the root S_1 to the head VBP is $S_1 \rightarrow S_2 \rightarrow VP_1 \rightarrow VP_2 \rightarrow VP_3 \rightarrow VBP$. Along the path the *PP* — *at FNX* — is a modifier of S_2; therefore, $S_1.b$, $S_2.t$, and the spine-etree rooted at *PP* form a mod-etree #1. Similarly, the ADVP *still* is a modifier of VP_2 and S_3 is a modifier of VP_3, and the corresponding structures form mod-etrees #4 and #7. On the path from the root to *VBP*, $S_1.t$ and $S_2.b$ are merged (and so are $VP_1.t$ and $VP_3.b$) to from the spine-etree #5. Repeating this process for other nodes will generate other trees such as trees #2, #3 and #6. The whole *ttree* yields twelve *etrees* as shown in Figure 9.

3.3.3 Building derivation trees

The fully bracketed *ttree* is in fact a derived tree of the sentence if the sentence is parsed with the *etrees* extracted by LexTract. In addition to these *etrees* and the derived tree, we also need derivation trees to train statistical LTAG parsers. Recall that, in general, given a derived tree, the derivation tree that can generate the derived tree may not be unique. Nevertheless, given the fully bracketed *ttree*, the *etrees*, and the positions of the *etrees* in the *ttree* (see Figure 8), the derivation tree becomes unique if we choose either one of the following:

- We adopt the traditional definition of derivation trees (which allows at most one adjunction at any node) and add an additional constraint which says that no adjunction operation is allowed at the foot node of any auxiliary tree.[2]
- We adopt the definition of derivation trees in (Schabes and Shieber, 1992) (which allows multiple adjunction at any node) and require all mod-etrees adjoin to the *etree* that they modify.

The user of LexTract can choose either option and inform LexTract about his choice by setting a parameter.[3] Figure 10 shows the derivation tree based on the second option.

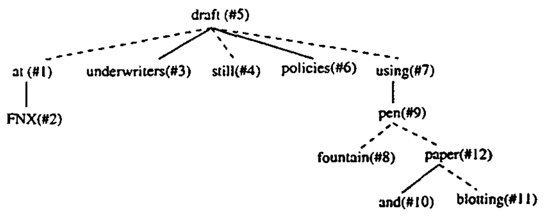


Figure 10: The derivation tree for the sentence

3.4 Uniqueness of decomposition

To summarize, LexTract is a language-independent grammar extraction system, which takes Treebank-specific information (see Section 3.2) and a *ttree* *T*, and creates

the root of one *etree* is merged with a node in the other *etree*. Splitting nodes into top and bottom pairs during the decomposition of the derived tree is the reverse process of merging nodes during parsing. For the sake of simplicity, we show the top and the bottom parts of a node only when the two parts will end up in different *etrees*.

[2]Without this additional constraint, the derivation tree sometimes is not unique. For example, in Figure 8, both #4 and #7 modify the *etree* #5. If adjunction were allowed at foot nodes, #4 could adjoin to #7 at $VP_{2.b}$, and #7 would adjoin to #5 at $VP_{3.b}$. An alternative is for #4 to adjoin to #5 at $VP_{3.b}$ and for #7 to adjoin to #4 at $VP_{2.t}$. The no-adjunction-at-foot-node constraint would rule out the latter alternative and make the derivation tree unique. Note that this constraint has been adopted by several hand-crafted grammars such as the XTAG grammar for English (XTAG-Group, 1998), because it eliminates this source of spurious ambiguity.

[3]This decision may affect parsing accuracy of an LTAG parser which uses the derivation trees for training, but it will not affect the results reported in this paper.

(1) a fully bracketed *ttree* T^*, (2) a set *Eset* of *etrees*, and (3) a derivation tree D for T^*. Furthermore, *Eset* is the only tree set that satisfies all the following conditions:

(C1) Decomposition: The tree set is a *decomposition* of T^*, that is, T^* would be generated if the trees in the set were combined via the substitution and adjunction operations.

(C2) LTAG formalism: Each tree in the set is a valid *etree*, according to the LTAG formalism. For instance, each tree should be lexicalized and the arguments of the anchor should be encapsulated in the same *etree*.

(C3) Target grammar: Each tree in the set falls into one of the three types as specified in Section 3.1.

(C4) Treebank-specific information: The head/argument/adjunct distinction in the trees is made according to the Treebank-specific information provided by the user as specified in Section 3.2.

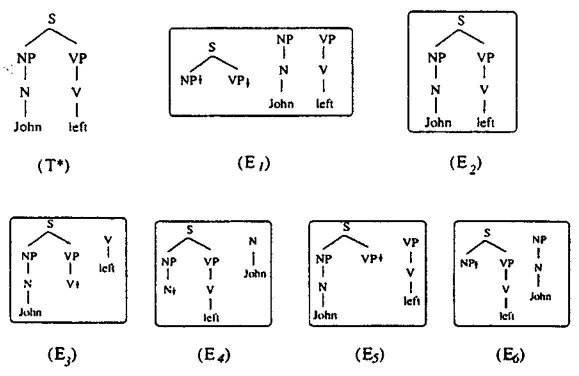


Figure 11: Tree sets for a fully bracketed *ttree*

This uniqueness of the tree set may be quite surprising at first sight, considering that the number of possible decompositions of T^* is $\Omega(2^n)$, where n is the number of nodes in T^*.[4] Instead of giving a proof of the uniqueness, we use an example to illustrate how the conditions ($C1$)—($C4$) rule out all the decompositions except the one produced by LexTract. In Figure 11, the *ttree* T^* has 5 nodes (i.e., S, NP, N, VP, and V). There are 32 distinct decompositions for T^*, 6 of which are shown in the same figure. Out of these 32 decompositions, only five (i.e., E_2 — E_6) are fully lexicalized — that is, each tree in these tree sets is anchored by a lexical item. The rest, including E_1, are not fully lexicalized, and are therefore ruled out by the condition ($C2$). For the remaining five *etree* sets, E_2 — E_4 are ruled out by the condition ($C3$), because each of these tree sets has one tree that violates one constraint which says that in a spine-etree an argument of the anchor should be a substitution node, rather than an internal node. For the remaining two, E_5 is ruled out by ($C4$) because according to the Head Table provided by the user, the head of the S node should be V, not N. Therefore, E_6, the tree set that is produced by LexTract, is the only *etree* set for T^* that satisfies (C1)—(C4).

3.5 The Experiments

We have ran LexTract on the one-million-word English Penn Treebank (Marcus et al., 1993) and got two Treebank grammars. The first one, G_1, uses the Treebank's tagset. The second Treebank grammar, G_2, uses a reduced tagset, where some tags in the Treebank tagset are merged into a single tag. For example, the tags for verbs, MD/VB/VBP/VBZ/VBN/VBD/VBG, are merged into a single tag V. The reduced tagset is basically the same as the tagset used in the XTAG grammar (XTAG-Group, 1998). G_2 is built so that we can compare it with the XTAG grammar, as will be discussed in the next section. We also ran the system on the 100-thousand-word Chinese Penn Treebank (Xia et al., 2000b) and on a 30-thousand-word Korean Penn Treebank. The sizes of extracted grammars are shown in Table 1. (For more discussion on the Chinese and the Korean Treebanks and the comparison between these Treebank grammars, see (Xia et al., 2000a)). The second column of the table lists the numbers of unique templates in each grammar, where *templates* are *etrees* with the lexical items removed.[5] The third column shows the numbers of unique

[4]Recall that the process of building *etrees* has two steps. First, LexTract treats each node as a pair of the top and bottom parts. The *ttree* is cut into pieces along the boundaries of the top and bottom parts of some nodes. The top and the bottom parts of each node belong to either two distinct pieces or one piece, as a result, there are 2^n distinct partitions. Second, some non-adjacent pieces in a partition can be glued together to form a bigger piece. Therefore, each partition will result in one or more decompositions of the *ttree*. In total, there are at least 2^n decompositions of the *ttree*.

[5]For instance, #3, #6 and #9 in Figure 9 are three different *etrees* but they share the same *template*. An *etree* can be seen as a (word, template) pair.

etrees. The average numbers of *etrees* for each word type in G_1 and G_2 are 2.67 and 2.38 respectively. Because frequent words often anchor many *etrees*, the numbers increase by more than 10 times when we consider word *token*, as shown in the fifth and sixth columns of the table. G_3 and G_4 are much smaller than G_1 and G_2 because the Chinese and the Korean Treebanks are much smaller than the English Treebank.

In addition to LTAGs, by reading context-free rules off the *etrees* of a Treebank LTAG, LexTract also produces CFGs. The numbers of unlexicalized context-free rules from G_1—G_4 are shown in the last column of Table 1. Comparing with other CFG extraction algorithms such as the one in (Krotov et al., 1998), the CFGs produced by LexTract have several good properties. For example, they allow unary rules and epsilon rules, they are more compact and the size of the grammar remains monotonic as the Treebank grows.

Figure 12 shows the log frequency of templates and the percentage of template tokens covered by template types in G_1.[6] In both cases, template types are sorted according to their frequencies and plotted on the X-axes. The figure indicates that a small portion of template types, which can be seen as the core of the grammar, cover majority of template tokens in the Treebank. For example, the first 100 (500, 1000 and 1500, resp.) templates cover 87.1% (96.6%, 98.4% and 99.0% resp.) of the tokens, whereas about half (3411) of the templates each occur only once, accounting for only 0.29% of template tokens in total.

4 Applications of LexTract

In addition to extract LTAGs and CFGs, LexTract has been used to perform the following tasks:

- We use the Treebank grammars produced by LexTract to evaluate the coverage of hand-crafted grammars.
- We use the (word, template) sequence produced by LexTract to re-train Srinivas' Supertaggers (Srinivas, 1997).
- The derivation trees created by LexTract are used to train a statistical LTAG parser (Sarkar, 2000). LexTract output has also been used to train an LR LTAG parser (Prolo, 2000).

[6]Similar results hold for G_2, G_3 and G_4.

- We have used LexTract to retrieve the data from Treebanks to test theoretical linguistic hypotheses such as the Tree-locality Hypothesis (Xia and Bleam, 2000).
- LexTract has a filter that checks the plausibility of extracted *etrees* by decomposing each *etree* into substructures and checking them. Implausible *etrees* are often caused by Treebank annotation errors. Because LexTract maintains the mappings between *etree* nodes and *ttree* nodes, it can detect certain types of annotation errors. We have used LexTract for the final cleanup of the Penn Chinese Treebank.

Due to space limitation, in this paper we will only discuss the first two tasks.

4.1 Evaluating the coverage of hand-crafted grammars

The XTAG grammar (XTAG-Group, 1998) is a hand-crafted large-scale grammar for English, which has been developed at University of Pennsylvania in the last decade. It has been used in many NLP applications such as generation (Stone and Doran, 1997). Evaluating the coverage of such a grammar is important for both its developers and its users.

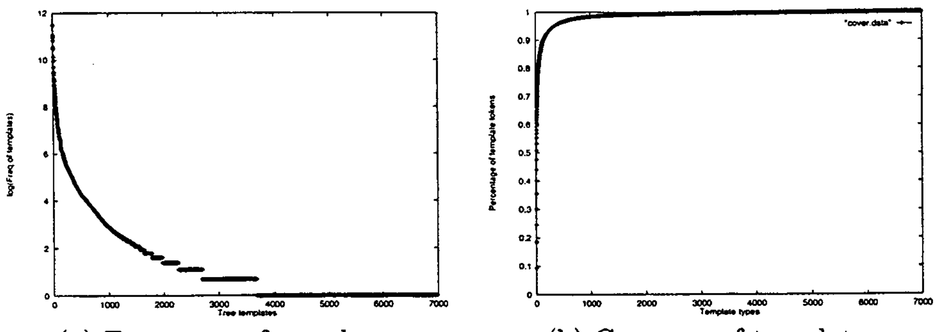
Previous evaluations (Doran et al., 1994; Srinivas et al., 1998) of the XTAG grammar use raw data (i.e., a set of sentences without syntactic bracketing). The data are first parsed by an LTAG parser and the coverage of the grammar is measured as the percentage of sentences in the data that get at least one parse, which is not necessarily the correct parse. For more discussion on this approach, see (Prasad and Sarkar, 2000).

We propose a new evaluation method that takes advantage of Treebanks and LexTract. The idea is as follows: given a Treebank T and a hand-crafted grammar G_h, the coverage of G_h on T can be measured by the overlap of G_h and a Treebank grammar G_t that is produced by LexTract from T. In this case, we will estimate the coverage of the XTAG grammar on the English Penn Treebank (PTB) using the Treebank grammar G_2.

There are obvious differences between these two grammars. For example, feature structures and multi-anchor *etrees* are present only in the XTAG grammar, whereas frequency information is available only in G_2. When we match templates in two grammars, we disre-

	template types	*etree* types	word types	*etree* types per word type	*etree* types per word token	CFG rules (unlexicalized)
Eng G_1	6926	131,397	49,206	2.67	34.68	1524
Eng G_2	2920	117,356	49,206	2.38	27.70	675
Ch G_3	1140	21,125	10,772	1.96	9.13	515
Kor G_4	634	9,787	6,747	1.45	2.76	177

Table 1: Grammars extracted from three Treebanks



(a) Frequency of templates (b) Coverage of templates

Figure 12: Template types and template tokens in G_1

gard the type of information that is present only in one grammar. As a result, the mapping between two grammars is not one-to-one.

	matched templates	unmatched templates	total
XTAG	497	507	1004
G_2	215	2705	2920
frequency	82.1%	17.9%	100%

Table 2: Matched templates in two grammars

Table 2 shows that 497 templates in the XTAG grammar and 215 templates in G_2 match, and the latter accounts for 82.1% of the template tokens in the PTB. The remaining 17.9% template tokens in the PTB do not match any template in the XTAG grammar because of one of the following reasons:

(T1) Incorrect templates in G_2: These templates result from Treebank annotation errors, and therefore, are not in XTAG.

(T2) Coordination in XTAG: the templates for coordinations in XTAG are generated on the fly while parsing (Sarkar and Joshi, 1996), and are not part of the 1004 templates. Therefore, the *conj-etrees* in G_2, which account for 3.4% of the template tokens in the Treebank, do not match any templates in XTAG.

(T3) Alternative analyses: XTAG and PTB sometimes choose different analyses for the same phenomenon. For example, the two grammars treat reduced relative clauses differently. As a result, the templates used to handle those phenomena in these two grammars do not *match* according to our definition.

(T4) Constructions not covered by XTAG: Some of such constructions are the unlike coordination phrase (UCP), parenthetical (PRN), and ellipsis.

For (T1)—(T3), the XTAG grammar can handle the corresponding constructions although the templates used in two grammars look very different. To find out what constructions are not covered by XTAG, we manually classify 289 of the most frequent unmatched templates in G_2 according to the reason why they are absent from XTAG. These 289 templates account for 93.9% of all the unmatched template tokens in the Treebank. The results are shown in Table 3, where the percentage is with respect to all the tokens in the Treebank. From the table, it is clear that the most common reason for mis-matches is (T3). Combining the results in Table 2 and 3, we conclude that 97.2% of template tokens in the Treebank are covered by XTAG, while another 1.7% are not. For the remaining 1.1% template tokens, we do not know whether or not they are covered by XTAG because we have not checked the remaining 2416 unmatched templates in G_2.[7]

To summarize, we have just showed that,

[7]The number 97.2% is the sum of two numbers: the first one is the percentage of matched template tokens (82.1% from Table 2). The second number is the percentage of template tokens which fall under (T1)—(T3), i.e., 16.8%-1.7%=15.1% from Table 3.

	T1	T2	T3	T4	total
type	51	52	93	93	289
freq	1.1%	3.4%	10.6%	1.7%	16.8%

Table 3: Classifications of 289 unmatched templates

by comparing templates in the XTAG grammar with the Treebank grammar produced by LexTract, we estimate that the XTAG grammar covers 97.2% of template tokens in the English Treebank. Comparing with previous evaluation approach, this method has several advantages. First, the whole process is semi-automatic and requires little human effort. Second, the coverage can be calculated at either sentence level or *etree* level, which is more fine-grained. Third, the method provides a list of *etrees* that can be added to the grammar to improve its coverage. Fourth, there is no need to parse the whole corpus, which could have been very time-consuming.

4.2 Training Supertaggers

A Supertagger (Joshi and Srinivas, 1994; Srinivas, 1997) assigns an *etree* template to each word in a sentence. The templates are also called *Supertags* because they include more information than Part-of-Speech tags. Srinivas implemented the first Supertagger, and he also built a Lightweight Dependency Analyzer that assembles the Supertags of words to create an almost-parse for the sentence. Supertaggers have been found useful for several applications, such as information retrieval (Chandrasekar and Srinivas, 1997).

To use a Treebank to train a Supertagger, the phrase structures in the Treebank have to be converted into (word, Supertag) sequences first. Producing such sequences is exactly one of LexTract's main functions, as shown previously in Section 3.3.2 and Figure 9.

Besides LexTract, there are two other attempts in converting the English Penn Treebank to train a Supertagger. Srinivas (1997) uses heuristics to map structural information in the Treebank into Supertags. His method is different from LexTract in that the set of Supertags in his method is chosen from the *pre-existing* XTAG grammar *before* the conversion starts, whereas LexTract extracts the Supertag set from Treebanks. His conversion program is also designed for this particular Supertag set, and it is not very easy to port it to another Supertag set. A third difference is that the Supertags in his converted data do not always fit together, due to the discrepancy between the XTAG grammar and the Treebank annotation and the fact that the XTAG grammar does not cover all the templates in the Treebank (see Section 4.1). In other words, even if the Supertagger is 100% accurate, it is possible that the correct parse for a sentence can not be produced by combining those Supertags in the sentence.

Another work in converting Treebanks into LTAGs is described in (Chen and Vijay-Shanker, 2000). The method is similar to ours in that both work use Head Percolation Tables to find the head and both distinguish adjuncts from modifiers using syntactic tags and functional tags. Nevertheless, there are several differences: only LexTract explicitly creates fully bracketed *ttrees*, which are identical to the derived trees for the sentences. As a result, building *etrees* can be seen as a task of decomposing the fully bracketed *ttrees*. The mapping between the nodes in fully bracketed *ttrees* and *etrees* makes LexTract a useful tool for Treebank annotation and error detection. The two approaches also differ in how they distinguish arguments from adjuncts and how they handle coordinations.

Table 4 lists the tagging accuracy of the same trigram Supertagger (Srinivas, 1997) trained and tested on the same original PTB data.[8] The difference in tagging accuracy is caused by different conversion algorithms that convert the original PTB data into the (word, template) sequences, which are fed to the Supertagger. The results of Chen & Vijay-Shanker's method come from their paper (Chen and Vijay-Shanker, 2000). They built eight grammars. We just list two of them which seem to be most relevant: C_4 uses a reduced tagset while C_3 uses the PTB tagset. As for Srinivas' results, we did not use the results reported in (Srinivas, 1997) and (Chen et al., 1999) because they are based on different training and testing data.[9] Instead, we re-ran

[8] All use Section 2-21 of the PTB for training, and Section 22 or 23 for testing. We choose those sections because several state-of-the-art parsers (Collins, 1997; Ratnaparkhi, 1998; Charniak, 1997) are trained on Section 2-21 and tested on Section 23. We include the results for Section 22 because (Chen and Vijay-Shanker, 2000) is tested on that section. For Srinivas' and our grammars, the first line is the results tested on Section 23, and the second line is the one for Section 22. Chen & Vijay-Shanker's results are for Section 22 only.

[9] He used Section 0-24 minus Section 20 for training and the Section 20 for testing.

his Supertagger using his data on the sections that we have chosen.[10] We have calculated two baselines for each set of data. The first one tags each word in testing data with the most common Supertag w.r.t the word in the training data. For an unknown word, just use its most common Supertag. For the second baseline, we use a trigram POS tagger to tag the words first, and then for each word we use the most common Supertag w.r.t. the (word, POS tag) pair.

	templates	base1	base2	acc
Srinivas'	483	72.59	74.24	85.78
		72.14	73.74	85.53
our G_2	2920	71.45	74.14	84.41
		70.54	73.41	83.60
our G_1	6926	69.70	71.82	82.21
		68.79	70.90	81.88
Chen's	2366 —	-	-	77.8 —
(sect 22)	— 8996			— 78.9
C_4	4911	-	-	78.90
C_3	8623	-	-	78.00

Table 4: Supertagging results based on three different conversion algorithms

A few observations are in order. First, the baselines for Supertagging are lower than the one for POS tagging, which is 91%, indicating Supertagging is harder than POS tagging. Second, the second baseline is slightly better than the first baseline, indicating using POS tags may improve the Supertagging accuracy.[11] Third, the Supertagging accuracy using G_2 is 1.3–1.9% lower than the one using Srinivas' data. This is not surprising since the size of G_2 is 6 times that of Srinivas' grammar. Notice that G_1 is twice the size of G_2 and the accuracy using G_1 is 2% lower. Fourth, higher Supertagging accuracy does not necessarily means the quality of converted data are better since the underlying grammars differ a lot with respect to the size and the coverage. A better measure will be the parsing accuracy (i.e., the converted data should be fed to a common LTAG parser and the evaluations should be based on parsing results). We are currently working on that. Nevertheless, the experiments show that the (word, template) sequences produced by LexTract are useful for training Supertaggers. Our results are slightly lower than the ones trained on Srinivas' data, but our conversion algorithm has several appealing properties: LexTract does not use pre-existing Supertag set; LexTract is language-independent; the (word, Supertag) sequence produced by LexTract fit together.

5 Conclusion

We have presented a system for grammar extraction that produces an LTAG from a Treebank. The output produced by the system has been used in many NLP tasks, two of which are discussed in the paper. In the first task, by comparing the XTAG grammar with a Treebank grammar produced by LexTract, we estimate that the XTAG grammar covers 97.2% of template tokens in the English Treebank. We plan to use the Treebank grammar to improve the coverage of the XTAG grammar. We have also found constructions that are covered in the XTAG grammar but do not appear in the Treebank. In the second task, LexTract converts the Treebank into a format that can be used to train Supertaggers, and the Supertagging accuracy is compatible to, if not better than, the ones based on other conversion algorithms. For future work, we plan to use derivation trees to train LTAG parsers directly and use LexTract to add semantic information to the Penn Treebank.

References

R. Chandrasekar and B. Srinivas. 1997. Gleaning information from the Web: Using Syntax to Filter out Irrelevant Information. In *Proc. of*

[10]Noticeably, the results we report on Srinivas' data, 85.78% on Section 23 and 85.53% on Section 22, are lower than 92.2% reported in (Srinivas, 1997) and 91.37% in (Chen et al., 1999). There are several reasons for the difference. First, the size of training data in our report is smaller than the one for his previous work; Second, we treat punctuation marks as normal words during evaluation because, like other words, punctuation marks can anchor *etrees*, whereas he treats the Supertags for punctuation marks as always correct. Third, he used some equivalent classes during evaluations. If a word is mis-tagged as x, while the correct Supertag is y, he considers that not to be an error if x and y appear in the same equivalent class. We suspect that the reason that those Supertagging errors are disregarded is that those errors might not affect parsing results when the Supertags are combined. For example, both adjectives and nouns can modify other nouns. The two templates (i.e. Supertags) representing these modification relations look the same except for the POS tags of the anchors. If a word which should be tagged with one Supertag is mistagged with the other Supertag, it is likely that the wrong Supertag can still fit with other Supertags in the sentence and produce the right parse. We did not use these equivalent classes in this experiment because we are not aware of a systematic way to find all the cases in which Supertagging errors do not affect the final parsing results.

[11]The baselines and results on Section 23 for (Chen and Vijay-Shanker, 2000) are not available to us.

AAAI 1997 Spring Symposium on NLP on the World Wide Web.

Eugene Charniak. 1996. Treebank Grammars. In *Proc. of AAAI-1996.*

Eugene Charniak. 1997. Statistical Parsing with a Context-Free Grammar and Word Statistics. In *Proc. of AAAI-1997.*

John Chen and K. Vijay-Shanker. 2000. Automated Extraction of TAGs from the Penn Treebank. In *6th International Workshop on Parsing Technologies (IWPT-2000), Italy.*

John Chen, Srinivas Bangalore, and K. Vijay-Shanker. 1999. New Models for Improving Supertag Disambiguation. In *Proc. of EACL-1999.*

Mike Collins. 1997. Three Generative, Lexicalised Models for Statistical Parsing. In *Proc. of the 35th ACL.*

C. Doran, D. Egedi, B. A. Hockey, B. Srinivas, and M. Zaidel. 1994. XTAG System - A Wide Coverage Grammar for English. In *Proc. of COLING-1994*, Kyoto, Japan.

Aravind Joshi and B. Srinivas. 1994. Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing. In *Proc. of COLING-1994.*

Aravind Joshi and K. Vijay-Shanker. 1999. Compositional Semantics with LTAG: How Much Underspecification Is Necessary? In *Proc. of 3nd International Workshop on Computational Semantics.*

Aravind K. Joshi, L. Levy, and M. Takahashi. 1975. Tree Adjunct Grammars. *Journal of Computer and System Sciences.*

Laura Kallmeyer and Aravind Joshi. 1999. Underspecified Semantics with LTAG.

Alexander Krotov, Mark Hepple, Robert Gaizauskas, and Yorick Wilks. 1998. Compacting the Penn Treebank Grammar. In *Proc. of ACL-COLING.*

David M. Magerman. 1995. Statistical Decision-Tree Models for Parsing. In *Proc. of the 33rd ACL.*

M. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Lingustics.*

K. F. McCoy, K. Vijay-Shanker, and G. Yang. 1992. A Functional Approach to Generation with TAG. In *Proc. of the 30th ACL.*

Martha Palmer, Owen Rambow, and Alexis Nasr. 1998. Rapid Prototyping of Domain-Specific Machine Translation System. In *Proc. of AMTA-1998*, Langhorne, PA.

Rashmi Prasad and Anoop Sarkar. 2000. Comparing Test-Suite Based Evaluation and Corpus-Based Evaluation of a Wide-Coverage Grammar for English. In *Proc. of LREC satellite workshop Using Evaluation within HLT Programs: Results and Trends*, Athen, Greece.

Carlos A. Prolo. 2000. An Efficient LR Parser Generator for TAGs. In *6th International Workshop on Parsing Technologies (IWPT 2000), Italy.*

Adwait Ratnaparkhi. 1998. *Maximum Entropy Models for Natural Language Ambiguity Resolution.* Ph.D. thesis, University of Pennsylvania.

Anoop Sarkar and Aravind Joshi. 1996. Coordination in Tree Adjoining Grammars: Formalization and Implementation. In *Proc. of the 18th COLING*, Copenhagen, Denmark.

Anoop Sarkar. 2000. Practical Experiments in Parsing using Tree Adjoining Grammars. In *Proc. of 5th International Workshop on TAG and Related Frameworks (TAG+5).*

The XTAG-Group. 1998. A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS 98-18, University of Pennsylvania.

Yves Schabes and Stuart Shieber. 1992. An Alternative Conception of Tree-Adjoining Derivation. In *Proc. of the 20th Meeting of the Association for Computational Linguistics.*

Yves Schabes. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars.* Ph.D. thesis, University of Pennsylvania.

Kiyoaki Shirai, Takenobu Tokunaga, and Hozumi Tanaka. 1995. Automatic Extraction of Japanese Grammar from a Bracketed Corpus. In *Proc. of Natural Language Processing Pacific Rim Symposium (NLPRS-1995).*

B. Srinivas, Anoop Sarkar, Christine Doran, and Beth Ann Hockey. 1998. Grammar and Parser Evaluation in the XTAG Project. In *Workshop on Evaluation of Parsing Systems*, Granada, Spain.

B. Srinivas. 1997. *Complexity of Lexical Descriptions and Its Relevance to Partial Parsing.* Ph.D. thesis, University of Pennsylvania.

Matthew Stone and Christine Doran. 1997. Sentence Planning as Description Using Tree Adjoining Grammar. In *Proc. of the 35th ACL.*

Bonnie Webber and Aravind Joshi. 1998. Anchoring a Lexicalized Tree Adjoining Grammar for Discourse. In *Proc. of ACL-COLING Workshop on Discourse Relations and Discourse Markers.*

Fei Xia and Tonia Bleam. 2000. A Corpus-Based Evaluation of Syntactic Locality in TAGs. In *Proc. of 5th International Workshop on TAG and Related Frameworks (TAG+5).*

Fei Xia, Chunghye Han, Martha Palmer, and Aravind Joshi. 2000a. Comparing Lexicalized Treebank Grammars Extracted from Chinese, Korean, and English Corpora. In *Proc. of the 2nd Chinese Language Processing Workshop*, Hong Kong, China.

Fei Xia, Martha Palmer, Nianwen Xue, Mary Ellen Okurowski, John Kovarik, Shizhe Huang, Tony Kroch, and Mitch Marcus. 2000b. Developing Guidelines and Ensuring Consistency for Chinese Text Annotation. In *Proc. of the 2nd International Conference on Language Resources and Evaluation (LREC-2000)*, Athens, Greece.