2

From Treebanks to Tree-Adjoining Grammars

FEI XIA AND MARTHA PALMER

2.1 Introduction

Grammars are valuable resources for natural language processing. A large-scale grammar may incorporate a vast amount of information on morphology, syntax, and semantics. Traditionally, grammars are built manually. Hand-crafted grammars often contain rich information, but they require tremendous human effort to build and maintain. As large-scale treebanks become available in the last decade, there has been much work on extracting grammars automatically from treebanks. Such grammars are called *treebank grammars.*[1]

Many of the previous work on grammar extraction such as (Shirai et al., 1995; Charniak, 1996; Krotov et al., 1998) generate context-free grammars (CFGs). In this chapter, we present a system, LexTract, which generates both CFGs and lexicalized tree-adjoining grammars (LTAGs).

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*), rather than context-free rules. Therefore, an LTAG extraction algorithm needs to examine a larger portion of a phrase structure to build an elementary tree. Second, because the adjoining operation in LTAG allows an elementary tree to be inserted within another elementary tree, an elementary tree is often formed by *gluing together* several disconnected parts of a phrase structure. Third, unlike in CFGs, parse trees (also known as *derived trees* in the LTAG formalism) and derivation trees (which describe how elementary trees are combined to form parse trees) are distinct 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 in addition to elementary trees.

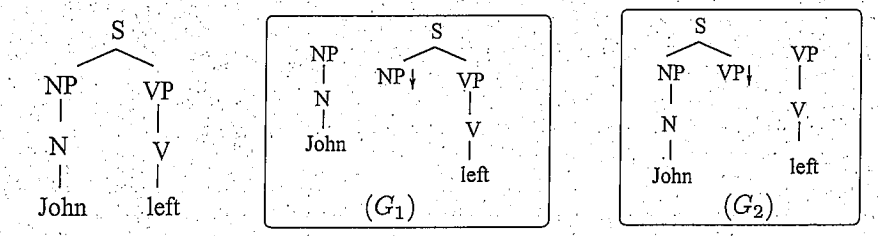
There were two main considerations when we designed LexTract: First, given a parse tree, the number of distinct LTAG grammars that produce this parse tree can be exponential with respect to the number of leaf nodes in the parse tree, and most of those grammars are not linguistically plausible. In order to extract only linguistically plausible grammars, we make certain assumptions about how three major relations (predicate-argument relations, modification relations, and coordinated relations) should be handled in LTAG grammars. The assumptions are based on well-established linguistic notions such as the notion of *head*. Second, in order to make LexTract a good grammar extraction tool that can be applied to various treebanks for different languages, we put all the language-dependent or treebank-dependent information in three tables (that is, the head percolation table, argument table, and tagset table). Users can easily modify these tables to reflect their own preferences. Given the assumptions and the tables, the process of extracting grammars is totally deterministic and we extract exactly one grammar for any given treebank. We have run the system on three publicly available treebanks, and the system output has been used in various NLP tasks.

As the LTAG formalism is a general framework and its usage is not restricted to natural languages, the formalism itself does not impose any constraint that is based solely on the properties of natural languages. Because the grammars that LexTract aims to extract are for natural languages only, we impose additional constraints on the treebank grammars to reflect the properties of natural languages. In section 2.2, we introduce these constraints and describe the grammar that we intend to extract. In section 2.3, we describe the extraction algorithm and compare it with related work. In section 2.4, we report experimental results on some tasks that use extracted grammars.

2.2 The Target Grammars

Given a parse tree, the LTAG grammars that can generate the parse tree are not unique. For instance, a simple parse tree such as the one in figure 2.1a can be produced by either grammar G_1 in figure 2.1b or grammar G_2 in figure 2.1c. While both grammars are LTAGs, people with a linguistic background would prefer G_1 over G_2 because it is more plausible to have the verb (rather than the noun) anchor a clause. The question is how we can equip LexTract with such linguistic knowledge so that it will produce G_1 rather than G_2.

Recall that the LTAG formalism is a general framework. Besides natural languages, the formalism can be used to generate formal languages such as $\{a^n b^n c^n\}$. Because its usage is not restricted to natural languages, the formalism itself does not impose constraints that are based solely on the properties of natural languages. As the grammars that LexTract aims to extract are for natural languages only, we would like to impose constraints on the target grammars (i.e., the grammars built by LexTract) to reflect the properties of natural languages. These extra constraints



(a) a parse tree (b) grammar G_1 (c) grammar G_2

FIGURE 2.1 A parse tree and two LTAGs that can generate the parse tree

are based on well-defined linguistic notions such as the notion of *head*. As a result, the target grammars form only a subset of all possible LTAG grammars. For the example in figure 2.1, LexTract will produce only G_1, not G_2.

In this section, let us first review a few important syntactic notions and show how they are represented in linguistic theories and LTAG grammars. The notions are a head and its projections, arguments and modifiers. We shall also define three prototypes and require that each elementary tree in the target grammars fall into one of the prototypes.[2]

2.2.1 Several important syntactic notions

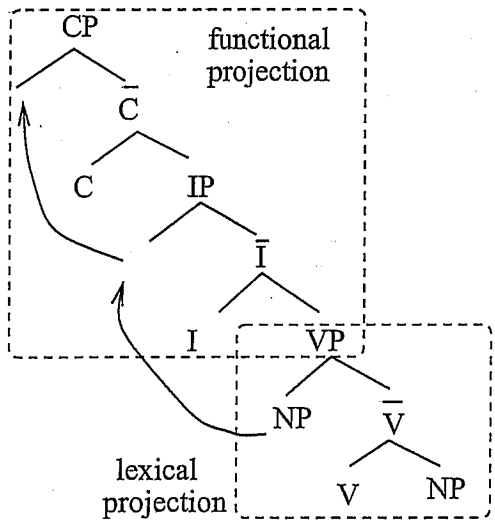
An important concept in many contemporary linguistic theories such as X-bar theory (Jackendoff, 1977) and GB theory (Chomsky, 1981) is the notion of *head*. A *head* determines the main properties of the phrase that it belongs to, and it may project to various levels. We call the chain formed by a head and its projections *a projection chain*. In X-bar theory (see figure 2.2a), a head X projects to $\bar{X}$, which further projects to XP. The X in this paradigm can be any part of speech such as a verb, where the XP is a phrase such as a verb phrase. GB theory divides heads into two types: lexical heads and functional heads. In figure 2.2b, V (for verb) is a lexical head, whereas C (for complementizer) and I (for inflection) are functional heads. The projections of lexical and functional heads are called *lexical* and *functional* projections, respectively. The solid arrows in the figure show the syntactic movement from a lower position to a higher position.

A head may have several arguments, and it and its projections can be modified by other components. For instance, a verb can project to a verb phrase, and it may have one or more arguments, and a verb phrase can be modified by preposition phrases, adverbial phrases, and so on.

2.2.2 Prototypes of elementary trees

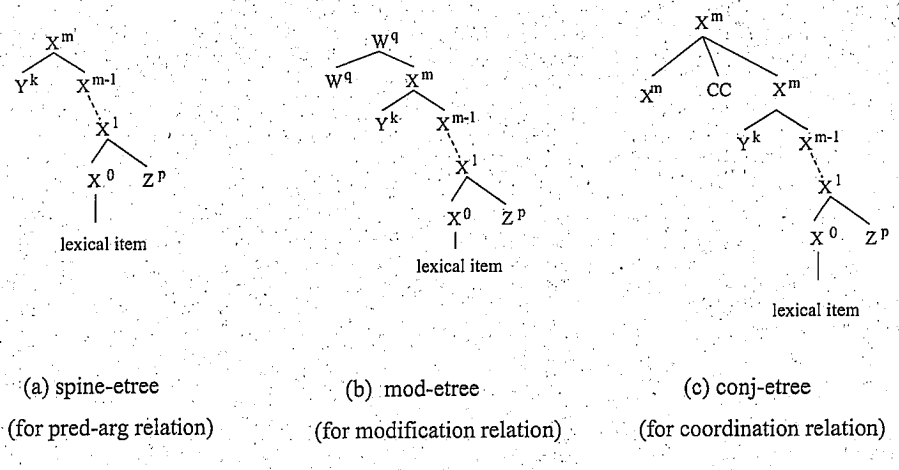
Recall that LTAG is a general framework, and therefore it does not have to follow a particular linguistic theory such as X-bar theory. However, the notions of head,

(1) XP -> YP $\overline{X}$
(2) $\overline{X}$ -> $\overline{X}$ WP
(3) $\overline{X}$ -> X YP



(a) rules in X-bar theory (b) a phrase structure in GB-theory

FIGURE 2.2 The notions of *head* in X-bar theory and GB-theory



(a) spine-etree (for pred-arg relation) (b) mod-etree (for modification relation) (c) conj-etree (for coordination relation)

FIGURE 2.3 The three forms of elementary trees in the target grammar

projection, argument, and modifier are widely accepted in the LTAG community, and people often follow some conventions when manually crafting LTAG grammars for natural languages. For instance, they often use initial trees to express predicate-argument relations: the anchor of an initial tree is the head of the root node, and all the arguments of the head are included in the same initial tree. In contrast, auxiliary trees are used to express modification relations, where the root node and the foot node have the label of the modified element, and the modifier is a sibling of the foot node. We formalize these conventions and define three types of elementary tree (*etree* for short) according to the relations between the anchor of the elementary tree and other nodes in the tree, as shown in figure 2.3:

- *Spine-etrees* for predicate-argument relations: A spine-etree is formed by a head X^0, its projections $X^1,...,X^m$, and its arguments. We call the path from X^0 to the root X^m a *projection chain*. The head X^0 is also the anchor of the tree, and its arguments are leaf nodes attached at various levels.
- *Mod-etrees* for modification relations: The root of a mod-etree has two children: one child has the same label (W^q) as the root, while the other child, X^m, is a modifier of W^q. The X^m child is further expanded into a spine-etree whose head X^0 is the anchor of the whole mod-etree.
- *Conj-etrees* for coordination relations: In a conj-etree, the children of the root are two conjoined constituents and one conjunction.[3] One conjoined constituent is expanded into a spine-etree whose head is the anchor of the whole tree. Structurally, a conj-etree is the same as a mod-etree except that the root has one extra conjunction child.

The similarity between the forms in figure 2.3 and rules in X-bar theory is obvious: a spine-etree is a tree that combines the first and the third types of rules in X-bar theory (see figure 2.2a); Similarly, a mod-etree incorporates all three types

of rules. A spine-etree is also very similar to the basic structure in GB-theory, as in figure 2.2b.

Some explanations about the prototypes are in order. First, each node in these prototypes may have zero or more children; when it has more than one child, the order among these children is not specified in the prototypes. For instance, the prototypes allow arguments Y^k and Z^p to appear to the left or to the right of X^0. Second, in the LTAG formalism elementary trees are divided into two types: initial trees and auxiliary trees. In this section, we define three forms of elementary trees. These two classifications are based on different criteria. The former classification is based on the existence of a foot node in the tree. Our classification is based on the relation between the anchor of the tree and other nodes in the tree. In general, spine-etrees are initial trees, mod-etrees and conj-etrees are auxiliary trees; however, there are exceptions to this generalization.[4] Third, the notions of *head* and *anchor* do not always coincide: the anchor of a spine-etree is the head of the root node, whereas the anchor of a mod-etree (or conj-etree) is the head of the modifier phrase, but not the head of the root node.

Now that we have defined the prototypes, we require each elementary tree produced by LexTract to fall into one of three prototypes. For a little abuse of notation, we also use the terms *spine-etree*, *mod-etree*, and *conj-etree* to refer to the corresponding templates.

2.3 The Extraction Algorithm

The core of LexTract is an extraction algorithm that takes a phrase structure in a treebank and produces an LTAG grammar. Extracting LTAGs is more complicated than extracting CFGs because of the differences between LTAGs and CFGs. First, the primitive elements of an LTAG are elementary trees, rather than context-free rules. Therefore, an LTAG extraction algorithm needs to examine a larger portion of a phrase structure to build an elementary tree. Second, because the adjoining operation in LTAG allows an elementary tree to be inserted within another elementary tree, an elementary tree is often formed by *gluing together* several disconnected parts of a phrase structure.

Our extraction algorithm has three steps: first, we convert a treebank tree (*ttree* for short) into a derived tree in the LTAG formalism. Figure 2.4 is a *ttree* example that we shall use throughout the section. The labels come from the English Penn Treebank (Marcus et al., 1993). There is a major difference between a *ttree* and a derived tree: in a *ttree*, arguments and modifiers are not always explicitly marked and structurally distinguished, and they can be siblings of one another. In contrast, the target grammars that we just defined distinguish heads, argument, and adjuncts, and arguments and adjuncts are never siblings in an LTAG derived tree. We convert a *ttree* into a derived tree by inserting more internal nodes so that arguments and adjuncts are attached at different levels.

In the second step of the algorithm, the newly created derived tree is decomposed into a set of elementary trees (a.k.a. *etrees*). In the third step, we create derivation

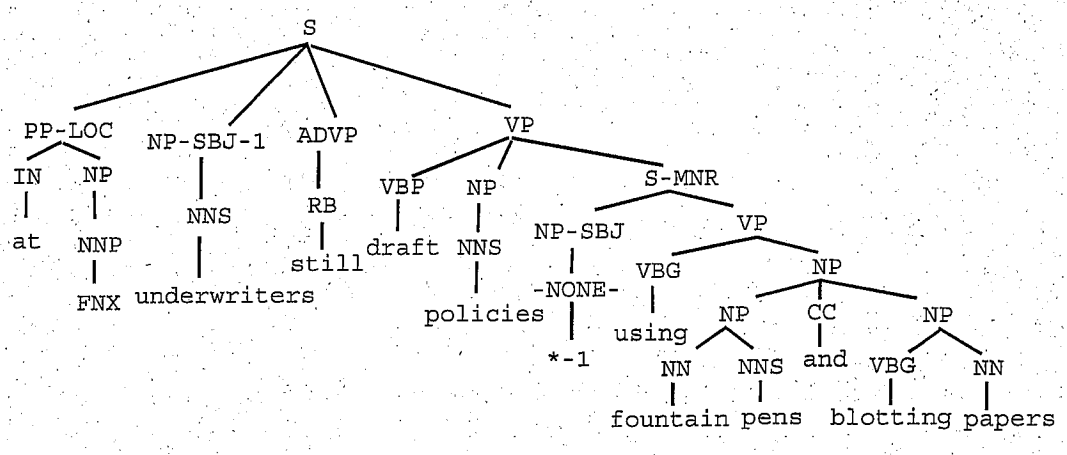


FIGURE 2.4 A treebank tree

trees, which show how *etrees* are combined to form the derived tree. The derivation trees are used to train statistical LTAG parsers.

We start the section with descriptions of three tables as part of the input to LexTract; we then describe the three steps of the extraction algorithm; finally we briefly discuss the uniqueness of the system output.

2.3.1 Distinguishing head, argument, and modifiers using three tables

In a *ttree*, the head of a phrase is not explicitly marked. Similarly, arguments and adjuncts are not structurally distinguished. In order to construct the *etrees*, which make this distinction, LexTract requires its user to provide some information about the treebank in the form of three tables: a Head Percolation Table, an Argument Table, and a Tagset Table. The Head Percolation Table is used to find the head of a phrase, whereas the Argument Table and the Tagset Table are used to make argument/modifier distinction.

In a Head Percolation Table, an entry is of the form (x *direct* $y_1/y_2/.../y_n$), where x and y_i are syntactic labels, *direct* is either $LEFT$ or $RIGHT$, and $\{y_i\}$ is the set of possible tags of x's head child. A *head child* of a node x in a *ttree* is the child of x which dominates the head of x. For instance, in figure 2.4, the head of the root node S is the VBP node, so the head child of S is the VP node.

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 theirs except that the order of the tags in the set $\{y_i\}$ does not matter in our algorithm and we do not use special rules to choose the head of noun phrases. To be more specific, to choose the head child of a node whose tag is X, we check the tags of the node's children from left to right (or vice versa according to

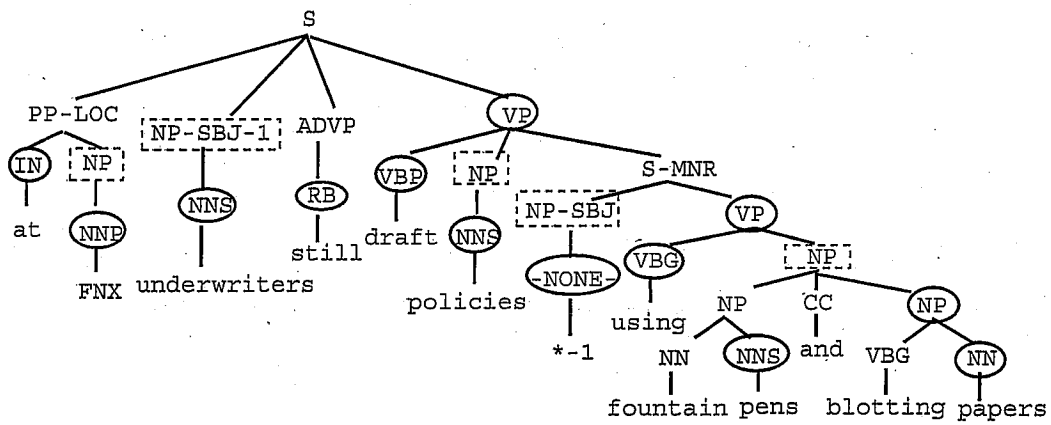


FIGURE 2.5 The *ttree* after head/argument/adjunct are distinguished,: head children are circled, and arguments are marked with dotted boxes.

direct) and find the first child whose tag is in $\{y_i\}$. Given that table, we mark the head child of every node in figure 2.4, as shown in figure 2.5.

An Argument Table specifies the number and the types of arguments that a head can take. The entry in an argument table is of the form *(head_tag, left_arg_num, right_arg_num,* $y_1/y_2/.../y_n$*)*: *head_tag* is the syntactic tag of a head, $\{y_i\}$ is the set of possible tags for the head's arguments, and *left_arg_num* (*right_arg_num*, respectively) is the maximal number of arguments to the left (right, respectively) of the head. For example, the entry *(IN, 0, 1, NP/S/SBAR)* says that a preposition (*IN*) does not have left arguments, and it has at most one right argument whose label is *NP*, *S*, or *SBAR*.

The Tagset Table provides types and attributes of the tags in the treebank's tagset. A few function tags (e.g., *SBJ* for subject) always mark arguments. Similarly, some function tags such as *TMP* for a temporal phrase always mark modifiers. Such information is specified in the Tagset Table. With the Argument Table and the Tagset Table, 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 sibling indicates that it is an adjunct. For example, in figure 2.4, the head of the root *S* is the verb *draft*, and the verb has two siblings: LexTract marks the noun phrase *policies* as an argument of the verb because from the Argument Table we know that verbs in general can take an NP object; it marks the clause *using fountain pens and blotting papers* 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.

All three tables can be created by hand. As each table contains only dozens of entries, it should take a person no more than a couple of hours to create these tables if he understands the basic notions of heads, arguments, and adjuncts, and

is familiar with the tagset of the treebank. The three sets of tables that we created for English, Chinese, and Korean Treebanks can be found in Xia (2001).

2.3.2 Step 1: Converting *ttrees* into derived trees

To extract *etrees* from a *ttree*, LexTract first converts the *ttree* into a derived tree by adding intermediate nodes to the *ttree* so that, at each level of the new *ttree*, exactly one of the following holds:

(Head-argument relation) there are one or more nodes: one is the head, the rest are its arguments;

(Modification relation) there are exactly two nodes: one node is modified by the other;

(Coordination relation) there are three nodes: two nodes are coordinated by a conjunction.

LexTract achieves this by first choosing the head-child at each level and distinguishing arguments from adjuncts as mentioned in section 2.3.1, then adding intermediate nodes so that the modifiers and arguments of a head attach to different levels. Figure 2.6 shows the new *ttree* after adding new nodes to the *ttree* in figure 2.4. The inserted nodes are in bold. It shall become clear after the next section that this new *ttree* is indeed a derived tree for the sentence if the sentence is parsed with the extracted *etrees* produced by LexTract.

2.3.3 Step 2: Building *etrees*

In this stage, each node X in the derived tree is split into two parts: the top part $X.t$ and the bottom part $X.b$. The reason for the splitting is as follows. When two *etrees* are combined during LTAG parsing, the root of one *etree* is merged with a node in the other *etree*. The resulting structure of the combined *etrees* is a derived tree. Therefore, a node in a derived tree actually has two parts (*top* and *bottom*), which could come from different *etrees*. Extracting *etrees* from a derived tree can be seen as the reverse process of parsing. Therefore, during the extraction process, each node in the derived tree is split into the top and bottom parts.

In this step, LexTract decomposes the derived tree into a set of *etrees*: LexTract removes recursive structures (which will become mod-etrees or conj-etrees) from the derived tree, and builds spine-etrees for the remaining nonrecursive structures. To be more specific, starting from the root of a derived tree, LexTract first finds the path from the root to its head. It then checks each node *hc* on the path. If a sibling *s* of *hc* in the *ttree* is marked as an adjunct, the algorithm factors out from the *ttree* the recursive structure that includes *hc.t*, *s.t*, and the bottom part of *hc*'s parent *p*. The recursive structure becomes part of a mod-etree (or a conj-etree if *hc* has another sibling that is a conjunction), in which *p.b* is the root node, *hc.t* is the foot node, and *s.t* is a sister of the foot node. Next, LexTract creates a spine-etree with the remaining nodes on the path and their siblings. It repeats the process for the subtrees whose roots are not on the path.

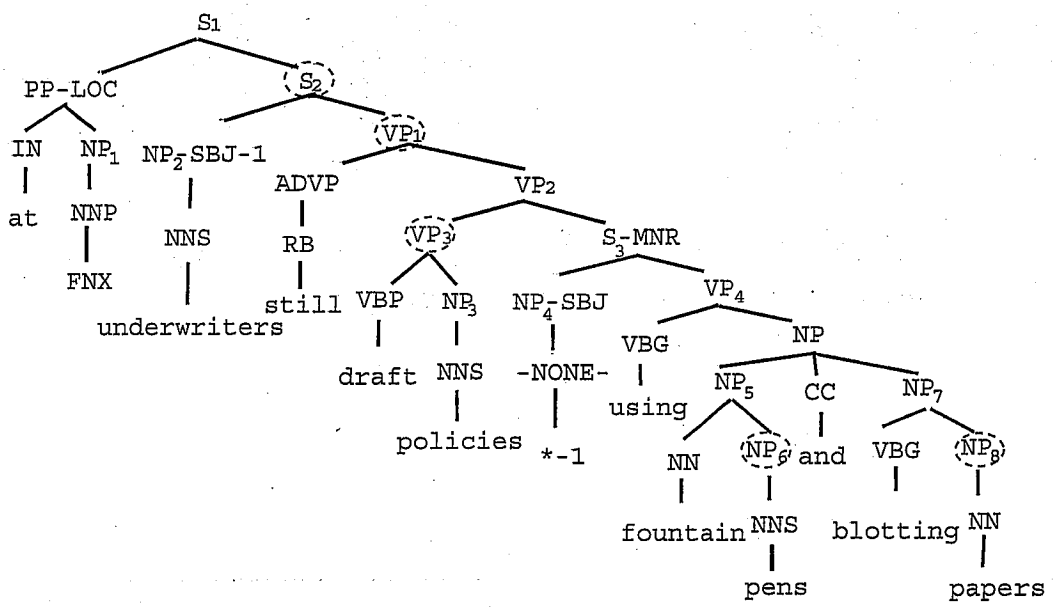


FIGURE 2.6 The new, expanded *ttree*: the five nodes in dotted circles are inserted by LexTract.

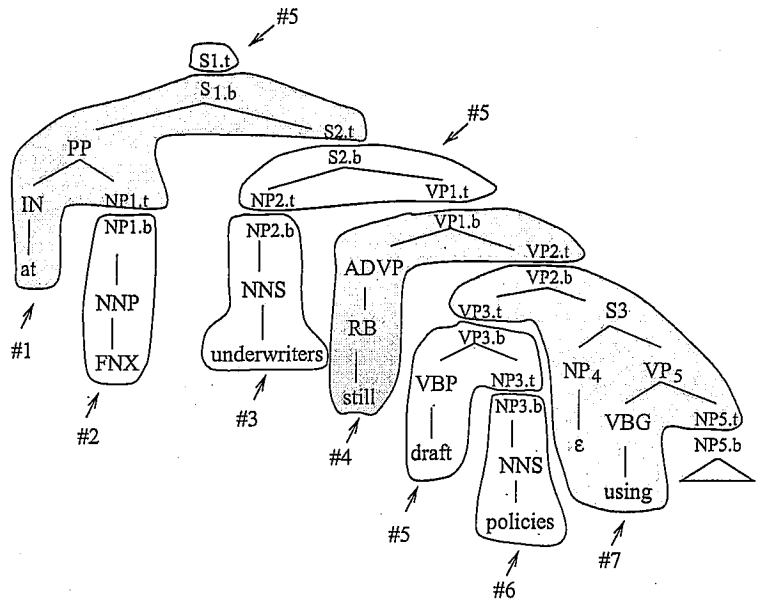


FIGURE 2.7 The extracted *etrees* can be seen as a decomposition of the new, expanded *ttree*.

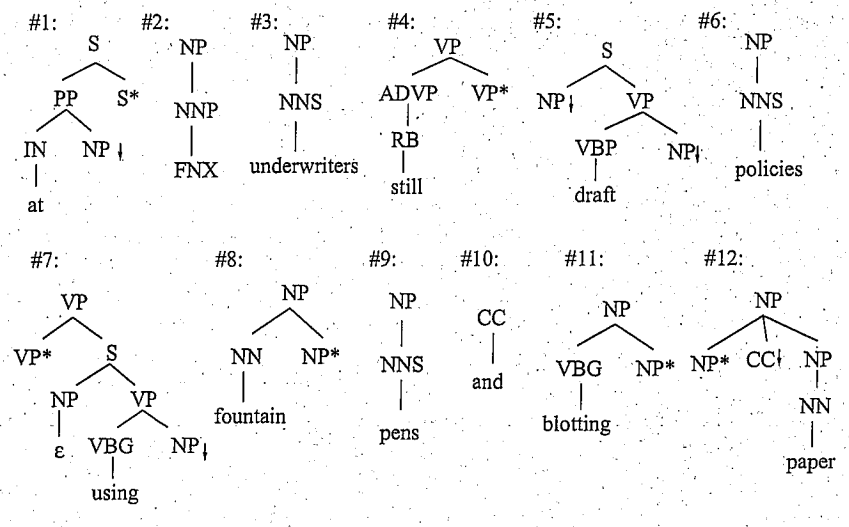


FIGURE 2.8 The extracted *etrees* from the expanded *ttree*

To see how the algorithm works, let us look at an example. Figure 2.7 shows the same derived tree as the one in figure 2.6 except that some nodes are numbered and split into the top and bottom parts. 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*. The 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*" has been marked as a modifier of S_2 in the previous stage; 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 form 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 2.8. Notice that the tree structures that form an *etree* are often not adjacent in the derived tree. For instance, the spine-etree #5 in figure 2.7 is separated by three mod-etrees (#1, #4, and #7) in the derived tree.

2.3.4 Step 3: Creating derivation trees

For the purpose of grammar development, a set of *etrees* may be sufficient. However, to train a statistical LTAG parser, derivation trees, which store the history of how *etrees* are combined to form derived trees, are required. Recall that, unlike in CFG, the derived trees and derivation trees in the LTAG formalism are different in the sense that a derived tree can be produced by several distinct derivation trees. There are two slightly different definitions of derivation trees in the LTAG literature. The first definition adopts the no-multi-adjunction constraint, whereas the second one allows multiple adjunctions at the same nodes under certain conditions (Schabes

and Shieber, 1992). The no-multi-adjunction constraint says that, when *etrees* are combined, at most one adjunction is allowed at any node in any *etree*. As a result, if a phrase XP in an *etree* E_h has several adjuncts (each adjunct belongs to a mod-etree), according to the first definition, the mod-etrees with these adjuncts form a chain in the derivation tree, with one mod-etree-adjoining to E_h and the rest adjoining to one another; whereas according to the second definition, these mod-etrees are allowed, but not required, to all adjoin to E_h. Figures 2.9 and 2.10 show two derivation trees, both combining the *etrees* in figure 2.8 to form the derived tree in figure 2.7. Note that mod-etrees #4 and #7 both modify #5 at the *VP* node, and they form a chain in figure 2.9, whereas they are siblings in figure 2.10.

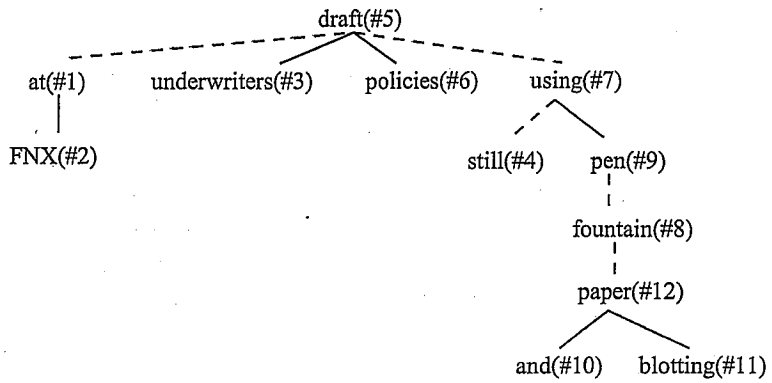


FIGURE 2.9 The derivation tree *with* the no-multi-adjunction constraint

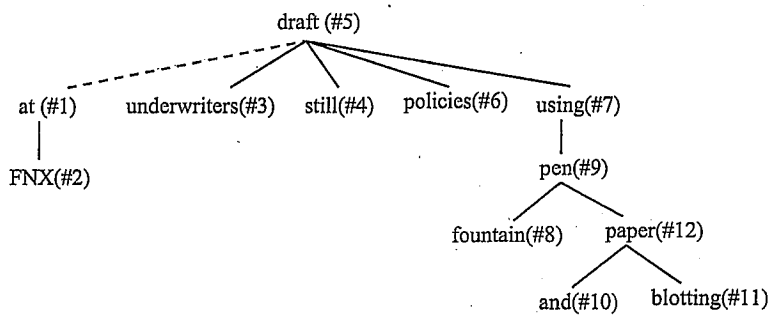


FIGURE 2.10 The derivation tree *without* the no-multi-adjunction constraint

In general, given a derived tree T and a set $ESet$ of *etrees*, there may be more than one derivation tree that generates T by combining *etrees* in $ESet$. This is because, when a phrase has several adjuncts, the corresponding *etrees* could form a chain in the derivation tree and the order of these etrees on the chain is not fixed.

For instance, switching the order of trees #4 and #7 in figure 2.9 (i.e., making #4 the parent of #7 and the child of #5) will yield a different derivation tree, which generates the same derived tree. Such differences in the derivation trees arguably do not imply any ambiguity in the meaning of the sentence. Therefore, we can impose either of the following constraints to make the derivation tree unique given a derived tree T and a set $ESet$ of *etrees*:

- If we adopt the first definition of derivation trees (which allows at most one adjunction at any node), we add an additional constraint which says that no adjunction operation is allowed at the foot node of any auxiliary tree. This no-adjunction-at-foot-node constraint makes the derivation tree unique by forcing the ordering of mod-etrees on the chain. This constraint has been adopted by several hand-crafted grammars such as the XTAG grammar for English (XTAG-Group, 1998) in order to eliminate this source of spurious ambiguity.
- If we use the second definition of derivation trees (which allows multiple adjunction at any node), we require all mod-etrees to adjoin to the *etree* that they modify. Because of this requirement, mod-etrees that modify the same *etree* are always siblings in the derivation tree.

The user of LexTract can choose either option and inform LexTract about one's choice by setting a parameter.[5] Once the choice is made, LexTract builds the derivation tree in two steps. First, for each *etree* in $ESet$, it finds the etree $\hat{e}$ which e substitutes/adjoins into; $\hat{e}$ will be the parent of e in the derivation tree. Second, it builds a derivation tree from those (e, $\hat{e}$) pairs. The algorithm can be found in chapter 5 of Xia (2001).

2.3.5 Uniqueness of decomposition

So far, we have discussed the extraction algorithm used by LexTract. The algorithm takes three tables with language-specific information and a *ttree* T, and creates (1) a derived tree T^*, (2) a set $ESet$ of *etrees*, and (3) a derivation tree D for T^*. The derivation tree D is unique given T and $ESet$ once we choose one of two options. Furthermore, in general, $ESet$ is the only tree set that satisfies all the following conditions:

(C1) **Decomposition:** The tree set is a *decomposition* of T^*; that is, T^* can be generated by combining the trees in the set via the substitution and adjoining operations.

(C2) **LTAG formalism:** Each tree in the set is an elementary tree according to the LTAG formalism. For instance, each tree is lexicalized and in an auxiliary tree the foot node and the root node have the same label.

(C3) **Target grammar:** Each tree in the set falls into one of the three types as specified in section 2.2.2.

(C4) **Language-specific information:** The head/argument/adjunct distinction in the trees is made according to the language-specific information provided by the user.

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 (including POS tags such as *N*, but excluding lexical items such as *John*) in T^*.[6] 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 2.11, the derived tree T^* has five nodes (i.e., *S, NP, N, VP,* and *V*). There are thirty-two distinct decompositions for T^*, six of which are shown in the same figure. Out of these thirty-two 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 the constraint that in a spine-etree an argument of the anchor should be a substitution node, rather than an internal node.[7] For the remaining two, E_5 is ruled out by ($C4$) because, according to the head percolation table provided by the user, the head-child of the *S* node should be the *VP* node, rather than the *NP* node. Therefore, E_6, the tree set that is produced by LexTract, is the only *etree* set for T^* that satisfies (C1)–(C4).

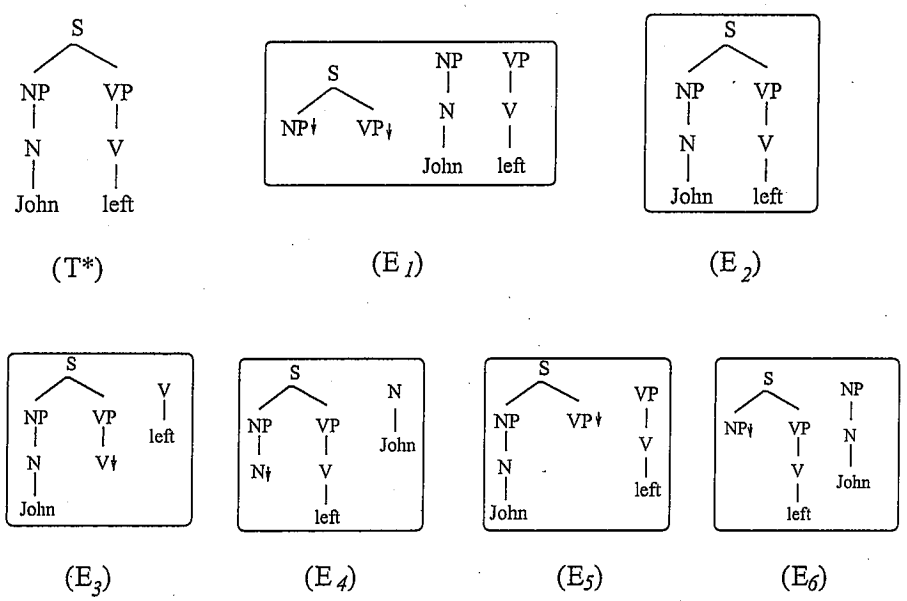
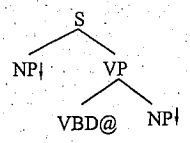


FIGURE 2.11 Six tree sets for the derived tree T^*: E_1 is ruled out by $C2$, $E_2 - E_4$ are ruled out by $C3$, and E_5 is ruled out by $C4$; E_6 satisfies all four conditions and is the one produced by LexTract.

2.3.6 Comparison with other work

LexTract is designed to extract LTAGs, but, as shown in figure 2.12, simply reading context-free rules off the templates in an extracted LTAG yields a context-free grammar. In this section, we compare LexTract with other extraction algorithms for CFGs and LTAGs proposed in the literature.



(a) a template

(1) S -> NP VP
(2) VP -> VBD NP

(b) CFG rules derived from (a)

FIGURE 2.12 The context-free rules derived from a template

CFG extraction algorithms

Many systems that use treebank context-free grammars simply read context-free rules off the phrase structures in treebanks. Because the phrase structures in the source treebanks are partially flat (i.e., arguments and adjuncts can be siblings), the resulting grammars are very large. There have been several research efforts addressing this issue. Due to space limitations, we compare LexTract with only one of these efforts, which is an algorithm that reduces the size of the derived grammar by eliminating redundant rules (Krotov et al., 1998). A rule is *redundant* if it can be "parsed" (in the familiar sense of context-free parsing) using other rules of the grammar. The algorithm checks each rule in the grammar in turn and removes the redundant rules from the grammar. For example, in a grammar that has the following three rules, the algorithm would remove Rule (1) because Rule (1) can be parsed by Rules (2) and (3):

Rule (1): $VP \rightarrow VB\ NP\ PP$
Rule (2): $VP \rightarrow VB\ NP$
Rule (3): $NP \rightarrow NP\ PP$

The rules that remain when all rules have been checked constitute the compacted grammar. The compact grammar for the PTB has 1,122 context-free rules, and the recall and precision of a CFG parser with the compact grammar are 30.93% and 19.18% respectively, in contrast to 70.78% and 77.66% of the same parser with the full grammar, which has 15,421 context-free rules.

Krotov's method differs dramatically from LexTract in several ways. First, it does not use the notion of *head* and it does not distinguish adjuncts from arguments. In the previous example, because of the existence of Rules (2) and (3), Rule (1) is

considered redundant and gets removed even though the *PP* in Rule (1) can be an argument of a verb such as *put*. Second, the compacting process may result in different grammars depending on the order in which the rules in the full grammar are checked. To maintain order-independence, their algorithm removed all unary and epsilon rules by collapsing them with the sister nodes. Because of frequent occurrences of empty categories and unary rules in the treebank, we suspect that this practice will make the resulting grammars less intuitive, and it might also contribute to the low parsing accuracy when the compact grammar was used. Third, the growth of their grammar is nonmonotonic in that, as the corpus grows, the size of the grammar may actually decrease because the new rules in the grammar may cause the existing rules to become redundant and get eliminated. Although the *size* of the compact grammar might approach a limit eventually in their experiment, it is not clear how stable the grammar really is, considering the existence of annotation errors in the treebank. For example, it is possible that a few bad rules (e.g., $\{X \rightarrow X\ ZP\}$, where *ZP* can be any syntactic label) can ruin the whole grammar because they make many good rules become redundant and get eliminated. They mentioned in their paper that they developed a linguistic compaction algorithm that could retain redundant but linguistically valid rules, and they gave the sizes of two grammars built by this new algorithm. Unfortunately, the description is too sketchy for us to determine exactly how that algorithm works.

In contrast, LexTract uses the notion of *head* and it distinguishes arguments from adjuncts. For instance, LexTract determines whether the *PP* in Rule (1) is an argument or an adjunct according to the Argument Table and the Tagset Table. If it is an argument, LexTract will keep the rule; if it is an adjunct, LexTract will replace this rule with Rule (2) and another rule $VP \rightarrow VP\ PP$. The redundant rules that Krotov's method would remove are not produced by LexTract because a context-free rule produced by LexTract never has both arguments and adjuncts as siblings. Second, the CFG produced by LexTract is order-independent, and it allows unary rules and epsilon rules. In addition, the growth of the grammar is monotonic, and the existence of bad rules would not affect the good rules. As for the number of context-free rules, the CFG built by LexTract from the PTB has 1,524 rules (see section 2.4.1), whereas in Krotov's approach, the compact grammar has 1,122 rules and the two linguistically compact grammars have 4,820 and 6,417 rules, respectively.[8]

LTAG extraction algorithms

We first published the extraction algorithm used by LexTract in Xia (1999). The algorithm was later revised and the new version and a few applications of LexTract were discussed in Xia et al. (2000a) and other papers. Besides LexTract, there are two systems that extract LTAGs from treebanks: they are (Neumann, 1998) and (Chen and Vijay-Shanker, 2000).

Neumann's lexicalized tree grammars: Neumann (1998) describes an extraction algorithm and tests it on the PTB and a German Treebank called

NEGRA (Skut et al., 1997). There are several similarities between his approach and LexTract. First, both approaches adopt notions of *head* and use a head percolation table to identify the head-child at each level. Second, both decompose the *ttrees* from the top downward such that the subtrees rooted by nonhead children are cut off and the cutting point is marked for substitution. The main difference between the two is that Neumann's system does not distinguish arguments from adjuncts, and therefore it does not factor out the majority of recursive structures with adjuncts. As a result, only 7.97% of the templates in his grammar are auxiliary trees, and the size of his grammar is much larger than ours: his system extracts 11,979 templates from three sections of the PTB (i.e., Sections 02–04), whereas LexTract extracts 6926 templates from the whole corpus (i.e., Sections 00–24). It is also not clear from his paper how he treats conjunctions, empty categories and coindexation; therefore, we cannot compare these two approaches on these issues.

Chen and Vijay-Shanker's approach: Chen and Vijay-Shanker's method (Chen and Vijay-Shanker, 2000) is similar to LexTract in that both use a head percolation table to find the head and both distinguish arguments from adjuncts. Nevertheless, there are several differences.

One major difference is the overall architecture. When we designed LexTract, we explicitly defined three prototypes of elementary trees in the target grammars. The prototypes are language independent and every *etree* built by LexTract falls into one of three prototypes. Given a treebank and three tables containing language-specific information, for each phrase structure (*ttree*) in the treebank, LexTract first explicitly inserts internal nodes to the *ttree* to form a LTAG derived tree. It then decomposes the derived tree into a set of *etrees*.

The bidirectional mapping between the nodes in this derived tree and the *etrees* makes LexTract a useful tool for treebank annotation and error detection (see section 2.4.5). LexTract also explicitly builds derivation trees. Chen and Vijay-Shanker's system does not explicitly define the prototypes of elementary trees, and it does not build derivation trees. Also it does not convert a *ttree* into an LTAG derived tree; therefore, there are no one-to-one mappings between the nodes in a *ttree* and the nodes in the extracted *etrees*. The two systems also differ in their algorithms for making argument/adjunct distinctions, their treatments for coordination, punctuation marks, and so forth.

Another way to compare these systems is to evaluate the performances of a common NLP tool that is trained by the data produced by the systems. One of such tools is Srinivas's Supertagger. In section 2.4.3, we shall report the performances of the Supertagger with the data produced by these two systems.

2.4 Applications of LexTract

In the previous section, we introduced a grammar extraction tool LexTract, which takes treebanks and three tables as input and produces grammars and

derivation trees. In this section, we discuss some applications of LexTract and report experimental results. These applications roughly fall into four types:

- The treebank grammars built by LexTract are useful for grammar development and comparison (sections 2.4.1 and 2.4.2).
- The lexicon and derivation trees derived from treebanks can be used to train statistical tools such as Supertaggers and parsers (sections 2.4.3 and 2.4.4).
- The bidirectional mappings between *ttree* nodes and *etree* nodes makes LexTract a useful tool for treebank annotation (section 2.4.5).
- LexTract can retrieve the data from treebanks to test theoretical linguistic hypotheses such as the tree-locality hypothesis (Xia and Bleam, 2000).

In this section, we shall briefly discuss the first three types.[9]

All the experimental results reported in this section were conducted by us, except for the parsing results of an LTAG statistical parser in section 2.4.4, which was produced by Anoop Sarkar.[10]

2.4.1 Treebank grammars as stand-alone grammars

The treebank grammars extracted by LexTract can be used as stand-alone grammars for languages that do not have wide-coverage grammars.

We ran LexTract on the English Penn Treebank (PTB) and extracted two treebank grammars. The first one, G_1, uses PTB's tagset. The second treebank grammar, G_2, uses a reduced tagset, where some tags in the PTB tagset are merged into a single tag, as shown in Table 2.1. The reduced tagset is basically the same as the one used in the XTAG grammar (XTAG-Group, 1998), which is a large-scale hand-crafted grammar that has been under development at the University of Pennsylvania since the early 1990s. We built G_2 with this reduced tagset for two reasons. First, we use G_2 to estimate the coverage of the XTAG grammar (see section 2.4.2). Second, G_2 is much smaller than G_1 and presumably the sparse data problem is less severe when G_2 is used. For some applications such as Supertagging and testing the tree-locality hypothesis, G_2 is as good as, if not better than, G_1.

The sizes of the two grammars are in Table 2.2. The first two columns show the number of templates and elementary trees. Recall that a template is an elementary tree without the anchor word. There are 49,206 unique words in the PTB, and the third column lists the average number of elementary trees that a word anchors. The last column of the table shows the number of context-free rules when we simply read context-free rules off the templates in an extracted LTAG.

In G_1 as well as G_2, a few templates occur very often while others occur rarely in the corpus. Among 6,926 templates in G_1, 96 templates each occur more than a thousand times, and they account for 86.91% of the template tokens in the PTB. In contrast, 3,276 templates occur only once, and together they account for only 0.27% of the template tokens in the PTB. In figure 2.13, we plot the frequency of the templates as a function of the rank of the templates on doubly logarithmic axes.

TABLE 2.1 Some tags in the English Penn Treebank tagset are merged into a single tag. The reduced tagset is used in G_2 and is similar to the tagset used in the XTAG grammar.

	tags in the PTB and G_1	tags in XTAG and G_2
adjectives	JJ/JJR/JJS	A
adverbs	RB/RBR/RBS/WRB	Ad
determiners	DT/PDT/WDT/PRP$/WP$	D
nouns	CD/NN/NNS/NNP/NNPS/PRP WP/EX/$/#	N
verbs	MD/VB/VBP/VBZ/VBN V VBD/VBG/TO	V
clauses	S/SQ/SBAR/SBARQ/SINV	S
noun phrases	NAC/NP/NX/QP/WHNP	NP
adjective phrases	ADJP/WHADJP	AP
adverbial phrases	ADVP/WHADVP	AdvP
preposition phrases	PP/WHPP	PP

TABLE 2.2 Two LTAG grammars extracted from the English Penn Treebank

	template types	*etree* types	# of *etrees* per word	context-free rules
LTAG G_1	6,926	131,397	2.67	1,524
LTAG G_2	2,920	117,356	2.38	675

The curve is close to a straight line, indicating that the relationship between the rank and frequency of templates satisfies a general version of Zipf's law.[11]

Once LexTract extracts grammars from treebanks, a natural question that comes to mind is: how complete is the grammar? To answer the question, we plot the number of templates as a function of the percentage of the corpus used to generate the templates, as in figure 2.14. To reduce the effect of the original ordering of the *ttrees* in the treebank, we randomly shuffle the *ttrees* in the treebank before running LexTract. We repeat the process ten times, and calculate the minimal, maximal, and average numbers of the templates generated by a certain percentage of the corpus. The figure shows that the curves for the minimal, maximal, and average template numbers are almost identical. Furthermore, in all three curves the number of templates does not converge as the size of the treebank grows, implying that there could be many new templates in new data.

As the number of templates does not coverage as the size of the treebank grows, the next question is whether these low frequency templates are linguistically plausible. To answer this question, we randomly selected 100 templates from the 3,276 templates in G_1 that occur only once in the corpus. After manually examining

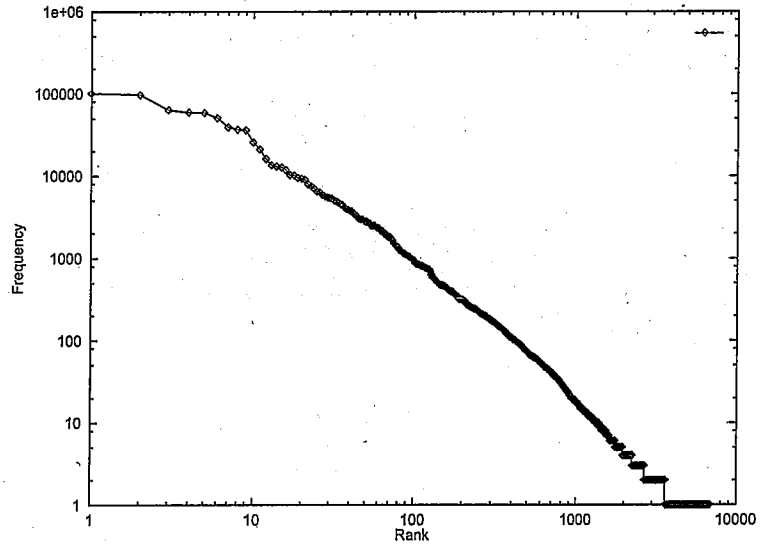


FIGURE 2.13 The relation between the rank and frequency of *etree* templates: X-axis and Y-axis show the rank and frequency of templates respectively, and both are on log scales.

them, we found that 41 templates resulted from annotation errors and two from missing entries in the language-specific tables that we made for the PTB; the remaining 57 were linguistically plausible. This experiment shows that, although the PTB is pretty large, G_1 is still missing many plausible templates for English.[12]

So far, the discussion has been based on templates, rather than on *etrees*. For parsing purposes, a more important question is: how often do the unseen *etrees* occur in new data? Recall that an *etree* is equivalent to a (word, template) pair. If an *etree* is unseen, the word can be unseen (*uw*) or seen(*sw*), and the template can be unseen (*ut*) or seen (*st*). Therefore there are four kinds of unseen pairs, where (sw, st) means both words and templates have appeared in the training data, but not the pair. Table 2.3 shows that in G_1 only 7.85% of the pairs in Section 23 of the PTB are not seen in Sections 2 to 21.[13] Of all the unseen (word, template) pairs in G_1, only 4.20% (0.31%+0.02% divided by 7.85%) are caused by unseen templates, and the remaining 95.80% are caused by unseen words or unseen combinations. This implies that the presence of unseen templates is unlikely to have a significant impact on Supertagging or parsing. In addition, most unseen (word, template) pairs are (sw, st) pairs, indicating that some type of smoothing over sets of templates (e.g., the notion of tree families in the XTAG grammar) could be helpful for improving parsing accuracy. In the table, we also list the percentages of unseen (word, POS tag) pairs in the same data for comparison. This table shows two differences between POS tags and templates. First, the number of POS tags is much smaller, and there are no unseen POS tags; consequently, the percentages for (sw, ut) and (uw, ut) are zero. Second, the percentage of unknown (word, POS tag) pairs where both words and

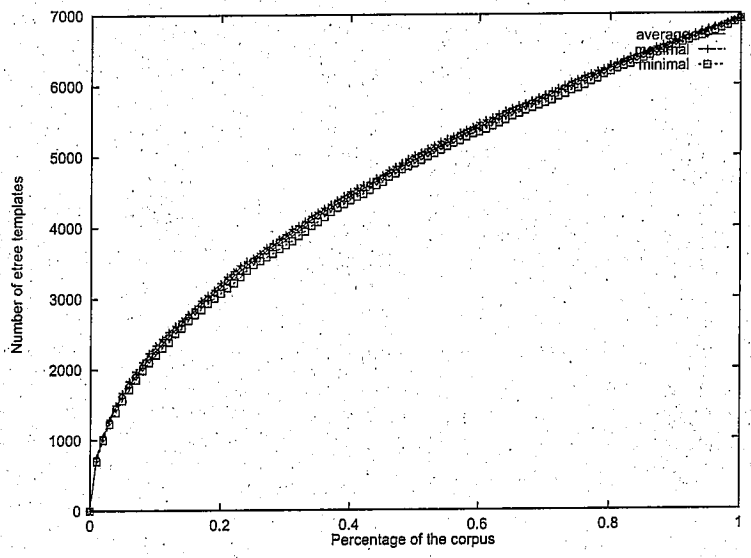


FIGURE 2.14 The growth of templates in G_1: X-axis shows the percentage of the corpus used for extraction (e.g., 0.2 means 20%), Y-axis shows the number of extracted templates.

POS tags are known is much lower than that for (word, template) pairs. Because of these differences, the baseline for POS tagging is much higher than the one for template tagging (i.e., Supertagging), as shall be discussed in section 2.4.3.

TABLE 2.3 The types of unseen (word, template) pairs in Section 23 of the English Penn Treebank

	# of tags	(sw, st)	(uw, st)	(sw, ut)	(uw, ut)	total
POS tags	48	0.44%	2.47%	0	0	2.91%
LTAG G_1	6,926	5.09%	2.43%	0.31%	0.02%	7.85%
LTAG G_2	2,920	4.20%	2.45%	0.10%	0.01%	6.76%

In addition to the English treebank, we also extracted grammars from the Chinese Penn Treebanks and the Korean Penn Treebanks.[14] The results are shown in Table 2.4. An interesting question is what kind of elementary trees and subtrees are shared among grammars for different languages. We have conducted some preliminary experiments and the results were reported in Xia et al. (2001).

An interesting question is how similar or different those treebank grammars are. In one of our previous experiments, we looked at each language pair, and counted the numbers of templates that are shared between the two corresponding treebank grammars. Then we classified the templates that appear in one grammar but not

TABLE 2.4 Grammars extracted from the English, Chinese, and Korean Penn Treebanks

	corpus size (in words)	word types	template types	*etree* types	context-free rules
English	1M	49,206	6,926	131,397	1,524
Chinese	100K	10,772	1,140	21,125	515
Korean	50K	10,035	632	13,941	152

the other. Some of the mismatches are due to spurious reasons (such as annotation errors or different annotation style), whereas the rest are due to the true differences between the two languages (for instance, some syntactic constructions appear only in one of the two languages). Our preliminary results were reported in Xia et al. (2001). The exercise helps us better understand the similarities and differences between languages with respect to their grammars. Another potential benefit of this exercise is that it produces the links between the templates in the grammars, which could be a valuable resources for transfer-based machine translation systems.

2.4.2 Treebank grammars combined with hand-crafted grammars

If a language already has a hand-crafted grammar such as the XTAG English grammar, we can use a treebank grammar to evaluate and improve the coverage of this hand-crafted grammar.

Previous evaluations (Doran et al., 2000; Srinivas et al., 1998) of hand-crafted grammars 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 can be parsed.[15] For more discussion on these evaluations, see Prasad and Sarkar (2000).

Now with the treebank grammar produced by LexTract, we can estimate the coverage of a hand-crafted grammar by measuring the overlap of the hand-crafted grammar and the treebank grammar. The main idea is as follows: given a treebank T and a hand-crafted grammar G_h, let G_t be the set of templates extracted from T by LexTract. The coverage of G_h on T can be measured as the percentage of template tokens in T that are covered by the *intersection* of G_t and G_h. One complication is that the treebank and G_h may choose different analyses for certain syntactic constructions; that is, although some constructions are covered by both grammars, the corresponding templates in these grammars would look very different. To address this problem, some human effort is required. In this section, we report the main results of our experiment; the details can be found in Xia and Palmer (2000).

In our experiment, we chose G_2 as our treebank grammar (see Table 2.2 in section 2.4.1) and the XTAG grammar as the hand-crafted grammar. The former has 2,920 templates, and the latter has 1,004 templates. We first calculate how many templates in the XTAG grammar *match* some template in G_2. We define two

types of matching: *t-match* and *s-match*. We say that two templates *t-match* (*t* for *template*) if they are identical barring the types of information present only in one grammar – such as feature structures in the XTAG grammars and the frequency information in G_2. The definition of *t-match* is pretty strict because it does not tolerate minor differences between two grammars such as the number of projections of a head. A more lenient measure is called *s-match*. Two templates are said to *s-match* if they are decomposed into the same set of subtemplates. A subtemplate can be a subcategorization frame, a modification pair, or a head projection chain. Note that neither type of matching is one-to-one.[16]

Table 2.5 lists the numbers of matched templates in the two grammars. The last row lists the percentage of the template tokens in the PTB that match some templates in XTAG. For instance, the first column says 173 templates in XTAG *t-match* 81 templates in G_2, and these 81 templates account for 78.6% of the template tokens in the PTB. There are 17.9% of the template tokens in the PTB that do not match any template in the XTAG grammar. There are several reasons why a template appears in G_2 but not in the XTAG grammar:

TABLE 2.5 The numbers of templates in the XTAG grammar that match some templates in G_2 and their frequencies in the English Penn Treebank

	t-match	s-match	matched subtotal	unmatched subtotal	total
# of templates in XTAG	173	324	497	507	1,004
# of templates in G_2	81	134	215	2,705	2,920
% of template tokens	78.6%	3.5%	82.1%	17.9%	100%

(T1) incorrect templates in G_2: These templates result from treebank annotation errors, and therefore they are not in XTAG.

(T2) coordination in XTAG: the templates for coordination in XTAG are generated on-the-fly during parsing (Sarkar and Joshi, 1996), and are not part of the 1,004 templates. Therefore, the conj-templates in G_2, which account for about 3.4% of the template tokens in the PTB, do not match any templates in the XTAG grammar.

(T3) alternative analyses: XTAG and G_2 sometimes choose different analyses for the same phenomena. As a result, the templates used to handle these phenomena do not *match* according to our definition.

(T4) constructions not covered by XTAG: Some constructions, such as the unlike coordination phrase (*UCP*), parenthetical (*PRN*), and ellipsis, are not currently covered by the XTAG grammar.

For the first three types, the XTAG grammar can handle the corresponding constructions although the templates used in two grammars look different and do not *match* according to our definition. To find out what constructions are not covered by XTAG, we manually classified the 289 most frequent unmatched templates in

G_2 according to (T1)–(T4) as previously defined. These 289 templates account for 16.8% of all the template tokens in the treebank. The results are shown in Table 2.6, where the percentage is with respect to all the template tokens in the treebank. From the table, it is clear that most unmatched template tokens are due to (T3); that is, alternative analyses adopted by the two grammars. Combining the results in Tables 2.5 and 2.6, we conclude that 97.2% of template tokens in the PTB can be handled by the current XTAG grammar,[17] while another 1.7% cannot. There are 2416 unmatched templates in G_2 that we have not checked manually, which account for the remaining 1.1% of template tokens in the PTB.

TABLE 2.6 Classification of 289 templates that are in G_2, but not in the XTAG grammar

	T1	T2	T3	T4	total
# of template types	51	52	93	93	289
% of template tokens	1.1%	3.4%	10.6%	1.7%	16.8%

Instead of just calculating the percentage of template tokens in the PTB that match templates in the XTAG grammar, we can also calculate the percentage of sentences in the PTB that can be parsed by the XTAG grammar. This can be done by first running LexTract to build a derivation tree for each sentence in the PTB as discussed in section 2.3.4. Each node in a derivation tree is a (word, template) pair as in figures 2.9 and 2.10. A sentence is covered by the XTAG grammar if both of the following conditions hold:

- For each (word, template) pair (w, t) in the derivation tree, there exists a template t' in the XTAG grammar such that t' t-matches or s-matches t and (w, t') is in the lexicon of the XTAG grammar.
- The new derivation tree made up of (w, t') could fit together.[18]

The average length of the sentences in the PTB is 23.8 words. For most of the sentences, their derivation trees contain at least one (word, template) pair that is not in the XTAG grammar; therefore, the percentage of sentences in the PTB that satisfies both conditions is very low. Nevertheless, this experiment provides a list of (word, template) pairs that could be added to the XTAG grammar to improve its coverage.

To summarize this section, we have presented a method for evaluating the coverage of a hand-crafted grammar – the XTAG grammar – on a treebank. First, we used LexTract to automatically extract a treebank grammar. Second, we matched the templates in the two grammars. Third, we manually classified unmatched templates in the treebank grammar to decide how many of them were due to missing constructions in the hand-crafted grammar. Some of the unmatched templates can be added to the hand-crafted grammar to improve its coverage. Our experiments showed that the XTAG grammar covers at least 97.2% of the template tokens in the English Penn Treebank. This method has several advantages. First, the whole process is semiautomatic and does not require much human effort. Second,

the method provides a list of templates and (word, template) pairs that can be added to the grammar to improve its coverage. Third, there is no need to use the grammar to parse the whole corpus and manually check whether the correct parses are produced, which can be very time-consuming. Fourth, the coverage of the grammar can be estimated at either the template level or the sentence level.

2.4.3 Lexicons as training data for supertaggers

A Supertagger (Joshi and Srinivas, 1994; Srinivas, 1997) assigns an *etree* template to each word in a sentence. The templates are called *Supertags* because they include more information than part-of-speech (POS) tags. In this section, we use these two terms (i.e., template and Supertag) interchangeably, and *a word has x Supertags* means that the word can anchor x distinct *templates*. In general, a word has many more Supertags than POS tags because a word appearing in different elementary trees will have different Supertags even if the POS tag of the word remains the same. For example, a preposition has different Supertags when the PP headed by the preposition modifies a VP, an NP, or a clause. In the PTB, on average, a word *type* has 2.67 Supertags, and a word *token* has 34.68 Supertags.[19] In contrast, on average a word *type* has 1.17 POS tags, whereas a word *token* has 2.29 POS tags.[20]

Srinivas implemented an n-gram Supertagger and he also built a Lightweight Dependency Analyzer (LDA) that assembles a Supertag sequence to create an almost-parse for a sentence. A Supertagger can also be used as a preprocessor (just like a POS tagger) to speed up parsing, because after the Supertagging stage an LTAG parser needs to consider only one or a few templates (in case of n-best Supertagging) for each word in the sentence, instead of every template that the word can anchor. Besides parsing, Srinivas (1997) has shown in his thesis that Supertaggers are useful for other applications, such as information retrieval, information extraction, language modeling, and simplification.

One difficulty in using Supertaggers is the lack of training and testing data. To use a treebank for that purpose, the phrase structures in the treebank have to be converted into (word, Supertag) sequences first. As discussed in section 2.3.3, LexTract builds a set of elementary trees from a parse tree. As an elementary tree is a (word, Supertag) pair, LexTract can easily produce a (word, Supertag) sequence for each sentence in the treebank. Besides LexTract, there have been two other attempts at converting the English Penn Treebank to (word, Supertag) sequences in order to train a Supertagger. One is Chen and Vijay-Shanker's method (Chen and Vijay-Shanker, 2000), which has been discussed in section 2.3.6. The other was reported in Srinivas (1997), in which the author first selected a subset of templates from the XTAG grammar, then used heuristics to map structural information in the treebank into the subset of templates. Srinivas's approach differs from LexTract and Chen and Vijay-Shanker's method in that it uses a preexisting Supertag set, rather than extracting a Supertag set directly from the treebank. As a result, it is not guaranteed that the Supertag sequences in Srinivas's converted data would 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 template tokens in the treebank.

In our experiment, we use the data converted by these three methods to train and test the same Supertagger (i.e., Srinivas's n-gram Supertagger). Except for the conversion methods, everything else is identical, including the Supertagger, the evaluation tool, and the original PTB data. The results are given in Table 2.7. Following the convention of recent parsing work, we use Sections 02–21 for training, and Section 23 for testing. We also include the results for Section 22 because (Chen and Vijay-Shanker, 2000) is tested on that section and its results on Section 23 are not available. The results of Chen and Vijay-Shanker's method come from their paper (Chen and Vijay-Shanker, 2000). They built eight grammars. We list two of them that seem to be most relevant: C_4 uses a reduced tagset while C_3 uses the PTB tagset. As for Srinivas's results, we had to rerun his Supertagger using his data on the sections that we have chosen, because his previous results were trained and tested on different sections of the PTB.[21]

TABLE 2.7 Supertagging results based on three different conversion algorithms, everything else such as the original data and Supertagger are identical. For comparison, two sets of baselines are provided: in the first set (base1), a word is tagged with most common Supertag for that word; in the second set (base2), a word is first POS tagged using a trigram POS tagger, then tagged with the most common Supertag for that (word, POS tag) pair.

	# of templates	section	base1	base2	tagging accuracy
Srinivas's grammar	483	23	72.59	74.24	85.78
		22	72.14	73.74	85.53
our G_2	2920	23	71.45	74.14	84.41
		22	70.54	73.41	83.60
our G_1	6926	23	69.70	71.82	82.21
		22	68.79	70.90	81.88
Chen and Vijay-Shanker's	2366 – 8996	22	-	-	77.8 – 78.9
C_4	4911	22	-	-	78.90
C_3	8623	22	-	-	78.00

We calculated two sets of baselines. For the first set, we tagged each word in the testing data with the most common Supertag for that word in the training data. For an unknown word, the most common Supertag was used. For the second set of baselines, we used a trigram POS tagger to tag the words first, and then for each word we used the most common Supertag for that (word, POS tag) pair. The table shows that the first set of baselines for Supertagging were around 70%, which are much lower than the 91% baseline for the POS tagging task if the same method is used. This implies that Supertagging is a much harder task than POS tagging. The

second set of baselines were slightly better than the first set of baselines, indicating that POS tags could help to improve the Supertagging accuracy.

A word of caution is in order. Because the sets of Supertags used by these conversion methods differ a lot with respect to the size, coverage, and so on, we cannot use Supertagging accuracy to compare the quality of grammar extraction tools that produce the training data for the Supertagger. For instance, the accuracy using G_1 is about 2% lower than the one using G_2, although both are produced by LexTract. Furthermore, higher Supertagging accuracy does not necessarily imply higher parsing accuracy when a tool (such as an LDA) is used to assemble a Supertag sequence to create a parse tree. We conducted this experiment only to show that the (word, template) sequences produced by LexTract are useful for training Supertaggers.

2.4.4 Derivation trees as training data for statistical LTAG parsers

In the previous section, we have shown that the (word, template) sequences produced by LexTract can be used to train a Supertagger. The output of a Supertagger can then be fed to an LDA or a parser to produce parse trees. A problem with this approach is that the Supertagging errors can hurt parsing performance.

Another way of using LexTract for parsing is to train an LTAG parser directly, without using a Supertagger as a preprocessor. There have been two LTAG parsers that use LexTract's output as training data. One is a head-corner LTAG statistical parser built by Anoop Sarkar. To reduce the amount of labeled data needed to train his parser, Sarkar adopts a cotraining method, which uses a small amount of labeled data, a large amount of unlabeled data, and a tag dictionary. *Labeled* data are sentences annotated with phrase structures; *unlabeled* data are sentences stripped of all annotations; and a tag dictionary is a set of (word, template) pairs. In his experiment, the labeled data are Sections 02–06 of the PTB, the unlabeled data are Sections 07–21 stripped of all annotation, and the tag dictionary includes all the (word, sequence) pairs from Sections 02–21. When tested on Section 23 of the PTB, the labeled bracketing precision and recall are 80.02% and 79.64%, respectively. Considering that the labeled data used by the parser are only about 25% of the training data used by other parsers, we believe that the results are very promising. The details of the generative model, the co-training method, and the experiment can be found in Sarkar (2001). The second parser that uses LexTract to convert the treebank data to the training data for LTAG parsers is a LR parser developed by Carlos Prolo. The details can be found in Prolo (2000).

2.4.5 LexTract as a tool for error detection in treebank annotation

Recall that given a treebank tree T, LexTract inserts additional internal nodes into it to form a new tree T^* and decomposes T^* into a set $ESet$ of elementary trees.

Because *ESet* is a decomposition of T^*, there is a mapping between the nodes in T^* and the nodes in *ESet*.[22] If there are annotation errors in T, those errors will be passed into T^*, and then to some *etrees* in *ESet*; as a result, those corresponding *etrees* are likely to be linguistically implausible.[23] For the reversed direction, if an *etree* is linguistically implausible, it implies that the corresponding nodes in the T^* and T are not annotated correctly. Based on this relation, we use LexTract to detect annotation errors in a treebank.

The algorithm for error detection has three steps: first, we run LexTract on the whole treebank to generate a grammar G; second, we check each template in G, decide whether it is plausible and mark it accordingly; third, for each *ttree* T_i in the treebank, we run LexTract and generate a grammar G_i. Obviously, G_i is a subset of G. If G_i includes any implausible *etree* as marked in G, then we modify T_i to T_i' so that the *etrees* generated by T_i' are all plausible. It is possible that the new *ttree* T_i' yields some new *etrees* which are not in G. In that case, we mark the plausibility of such *etrees* and add them to G. In this algorithm, human effort is required with respect to two aspects: checking the plausibility of *etrees* and modifying *ttrees*.[24]

We used LexTract for the final cleanup of the Chinese Penn Treebank. The treebank contained about 100,000 words after word segmentation, and the average sentence length was 23.8 words. Before LexTract was used for the final cleanup, the treebank had been manually checked at least twice and the annotation accuracy was already above 95%. Details on the treebank can be found in Xia et al. (2000b) and at the website *http://www.ldc.upenn.edu/ctb*; the treebank is available to the public via LDC.

Before the final cleanup, the treebank grammar G_o contained 1245 *etree* templates. It took a linguistics expert about ten hours to manually examine all the templates in G_o to determine whether they were plausible. After that, it took another person (who was one of the two annotators in the Chinese Treebank project) about twenty hours to run LexTract and correct treebank annotation. After the cleanup, 169 templates in the old grammar disappeared, and 38 templates were added to the new grammar; so the new grammar has 1,114 *etree* templates. We also automatically counted the number of word tokens in the treebank that anchored distinct templates before and after the cleanup. We found 579 word tokens (which account for 0.58% of the total number of word tokens in the treebank) whose templates had changed after the cleanup. The differences may not be huge, but considering the accuracy before running LexTract was already above 95%, the results of the final cleanup were satisfactory. These errors found by LexTract can be classified as follows:

- Formatting errors in *ttrees* such as unbalanced brackets and illegal tags: when a *ttree* is not properly formatted, LexTract will give a warning and exit without further processing of the *ttree*.
- Mismatched syntactic labels (including POS tags, phrase labels and empty category tags): Except for careless typos (e.g., using the tag *LC* (localizer) rather than *CL* (classifier) for a classifier in the Chinese Treebank), mismatched

syntactic labels are often due to incompatible labels at several levels. For example, in Chinese, a coordinating conjunction (CC) such as *tong* is also a preposition (IN); therefore, the sentence "*John tong/and_with Mary zou/leave le/ASP*" means either "*John and Mary left*" or "*John left with Mary*", and both structures in figure 2.15a and 2.15b are correct.[25] However, the structure in figure 2.15c is incorrect because the POS tag *CC* and the phrase label *PP* do not match, resulting in an implausible *etree* in figure 2.15d. This type of error is relatively common because POS tagging and bracketing were done at separate annotation stages by different annotators.

```
(S (NP-SBJ (NP (NNP John))
           (CC tong)
           (NP (NNP Mary)))
   (VP (VB zou)
       (AS le)))
```

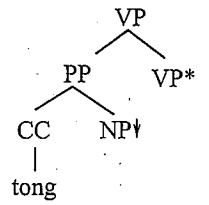
(a) tong as a conjunction

```
(S (NP-SBJ (NNP John))
   (VP (PP (IN tong)
           (NP (NNP Mary)))
       (VB zou)
       (AS le)))
```

(b) tong as a preposition

```
(S (NP-SBJ (NNP John))
   (VP (PP (CC tong)
           (NP (NNP Mary)))
       (VB zou)
       (AS le)))
```

(c) the incompatible labels



(d) the resulting *etree*

FIGURE 2.15 An error caused by incompatible labels: (a) and (b) are two correct structures; (c) has two mismatched labels (*PP* and *CC*), both marked in boldface; (d) is the elementary tree resulted from the annotation errors in (c).

- Wrong or missing function tags: LexTract uses syntactic labels and function tags to distinguish arguments from adjuncts. Wrong or missing function tags may cause an argument to be mistaken as an adjunct by LexTract or vice versa. For example, the Chinese Treebank annotation guidelines require that the subject of a verb should always have the function tag *-SBJ*, but sometimes annotators forgot to do that. In figure 2.16, the structure in (c) is identical to the one in (a) except that the subject in (c) is missing such a function tag; as a result,

LexTract treats the subject in (c) as an adjunct, and creates an implausible *etree* in (d) rather than the plausible *etree* in (b).

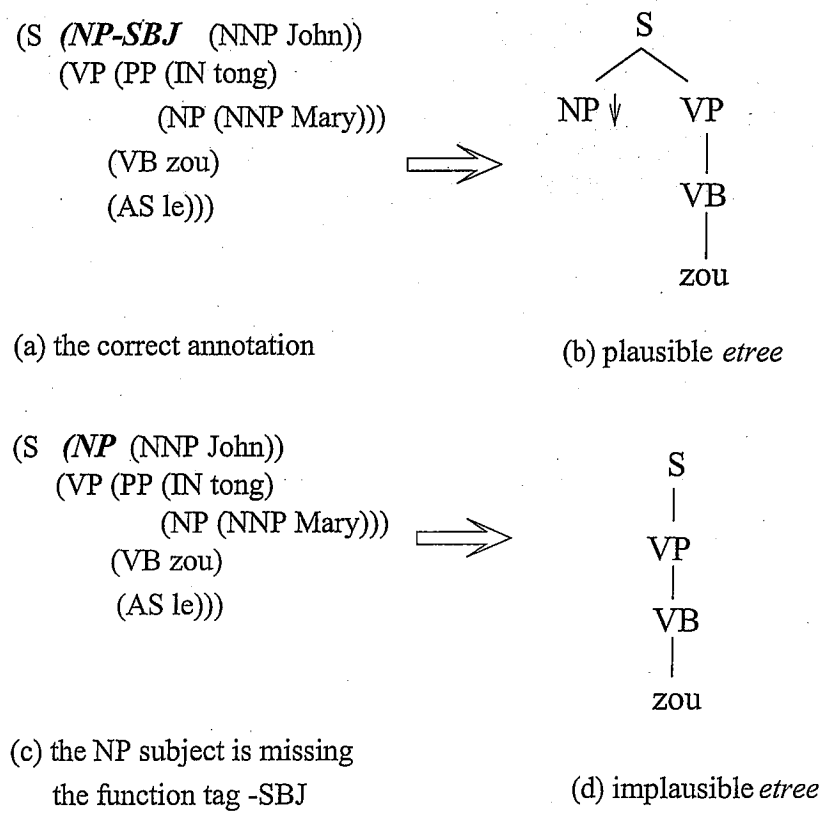


FIGURE 2.16 An error caused by a missing function tag: (a) is the correct structure; (b) is one of the *etrees* extracted from (a); (c) misses the function tag *SBJ*; (d) is an extracted *etree* from (c), and it is implausible because it does not contain the subject of the verb.

- Missing *ttree* nodes: One reason for this type of error in the Chinese Penn Treebank is that annotators forgot to mark dropped arguments. In figure 2.17, the dropped argument should be marked as an empty category **pro**, as in (a). Failing to do that, as in (c), would result in an implausible *etree* in (d).
- Extra *ttree* nodes: This type of error is rare and mostly caused by careless typos or misunderstanding of the annotation guidelines.

Two observations are in order. First, the main function of LexTract is extracting LTAGs and building derivation trees to train LTAG parsers and Supertaggers. Error detection is only a byproduct of the system. Consequently, there are errors that LexTract cannot detect; namely, the errors that do not result in implausible *etrees*.

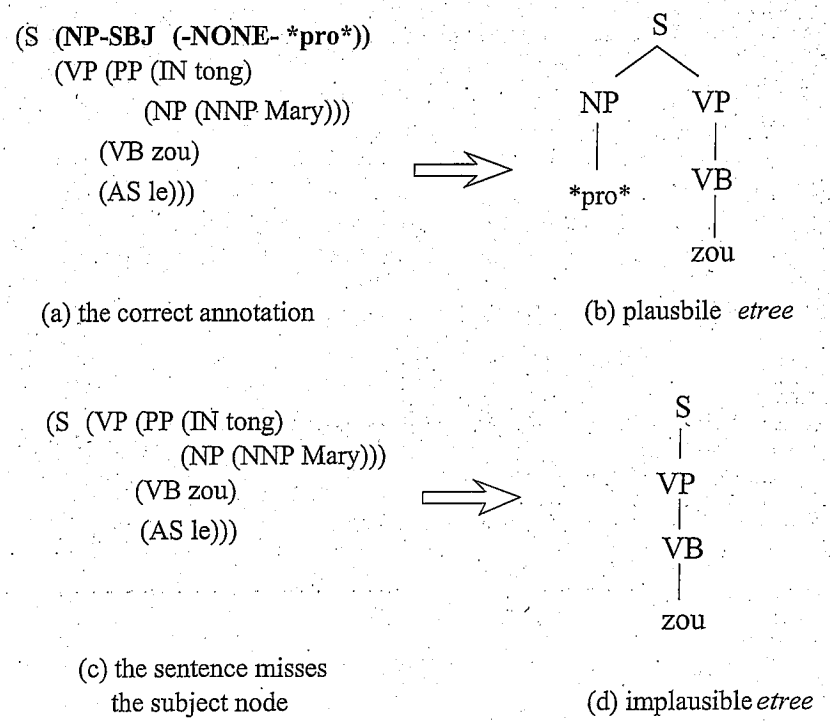


FIGURE 2.17 An error caused by a missing subject node: (a) is the correct structure; (b) is one of the *etrees* extracted from (a); (c) misses the *NP-SBJ* node and its child; (d) is an extracted *etree* from (c), and it is implausible because it does not contain the subject of the verb.

For example, in English, a *PP* can modify either an *NP* or a *VP*. Given a particular context, in general, only one attachment makes sense. If the treebank chooses the wrong attachment, LexTract cannot detect that error.

The second observation is that using templates can detect more annotation errors than using context-free rules. For example, in English either the subject or the object of a verb can undergo wh-movement and leave a trace in its position, as shown in figure 2.18a and 2.18b. But the subject and the object cannot be moved at the same time, as in figure 2.18c. That is, the first two templates are plausible but the third template is not. However, all three templates consist of the same set of context-free rules as in figure 2.18d, and all the context-free rules are plausible. Thus, the annotation errors that result in the implausible template in figure 2.18c can be detected only if we use templates, rather than context-free rules.

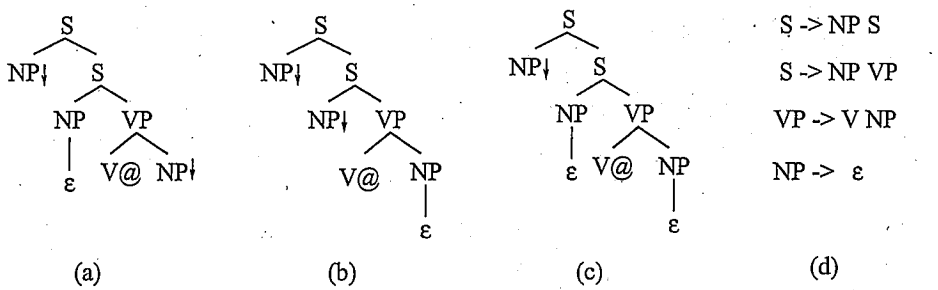


FIGURE 2.18 Three templates and corresponding context-free rules: the first two templates are plausible, while the third one is not. The context-free rules derived from these three templates are identical and each rule is plausible.

2.5 Summary

We outlined a system named LexTract, which takes a treebank and language-specific information, and produces grammars (LTAGs and CFGs) and derivation trees. LexTract has several advantageous properties. First, it takes very little human effort to build three tables (i.e., the Tagset Table, the Head Percolation Table, and the Argument Table). Once the tables are ready, LexTract can extract grammars from treebanks in a short time. Because LexTract does not include any language-dependent code, it can be applied to various treebanks for different languages. Second, LexTract builds a unique derivation tree for each sentence in the treebank, which can be used to train statistical LTAG parsers directly. Third, LexTract allows its users to have some control over the kind of treebank grammar to be extracted. For example, by changing the entries in the Head Percolation Table, the Argument Table, and the Tagset Table, users can get different treebank grammars and they can then choose the ones that best fit their goals. Fourth, the grammar produced by LexTract is guaranteed to cover the source treebank.

We have used LexTract for four types of tasks. First, treebank grammars produced by LexTract are useful for grammar development and comparison. For example, a treebank grammar can be used as a stand-alone grammar. We also used a treebank grammar extracted from the English Penn Treebank to estimate and improve the coverage of the XTAG grammar. Second, the treebank grammar and derivation trees produced by LexTract were used to train a Supertagger and a statistical LTAG parser with satisfactory results. Third, we used LexTract to detect annotation errors in the Chinese Penn Treebank. Last, we used LexTract to find all the nonlocal examples from the English Penn Treebank in order to test the tree-locality hypothesis. The details of these experiments can be found in Xia (2001). All these applications indicate that LexTract is not only an engineering tool of great value, but it is also very useful for investigating theoretical linguistics.

Notes

1. Treebank grammars may contain less information (e.g., feature structures associated with nonterminal nodes) than some hand-crafted grammars, but they are sufficient for some NLP tasks such as Supertagging and parsing as described in section 2.4.

2. Another possible name for a prototype is *template*. We do not use this name because the term *template* in the LTAG formalism is used to refer to an unlexicalized elementary tree.

3. Some users may prefer to treat the conjunction as the anchor of the tree, and treat the conjoined constituents as substitution nodes. It is also possible that two conjoined constituents are connected by two conjunctions (e.g., the conjunction pair "*both ... and ...*" in English). LexTract can accommodate both cases, but we omit the details in this chapter for the sake of simplicity.

4. One exception is the predicative auxiliary tree for verbs such as *think*. The elementary tree for the verb *think* in a declarative sentence is an auxiliary tree to handle long-distance movement, but it is also a spine-etree as it shows the predicate-argument relation of the verb.

5. This choice will not change the elementary trees extracted from a *ttree*, but they will result in different derivation trees, as in figures 2.9 and 2.10. It is possible that one option is better than the other for training a particular LTAG parser.

6. 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 derived tree is cut into pieces along the boundaries of the top and bottom parts of some nodes. In any partition, 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 of the derived tree. In the second step, two nonadjacent pieces in a partition can be glued together to form a bigger piece under certain conditions. Therefore, each partition will result in one or more decompositions of the derived tree. In total, there are at least 2^n decompositions for any derived tree with n nodes; that is, the number of possible decompositions is $\Omega(2^n)$.

7. The prototypes actually allow the arguments of an anchor to be further expanded and lexicalized in order to handle noncompositional phrases such as *keep the bucket*; however, because the Penn Treebanks currently do not mark these noncompositional phrases, all the *etrees* extracted from the treebanks will have single anchors, and the arguments of the anchor are substitution nodes.

8. Unfortunately, we do not have access to the CFG parser they used; therefore, we cannot compare our grammar with their grammars with respect to the precision and recall rates of the parser.

9. We leave out the last type mainly because understanding our experiment for this type requires the knowledge of Multiple-Component Tree-Adjoining Grammars (MCTAG), a topic that we do not cover in this chapter.

10. Anoop Sarkar attended graduate school at the University of Pennsylvania when the experiment was conducted. He is currently a faculty member at Simon Fraser University in Vancouver, Canada.

11. Zipf's Law says that in a large corpus the rank of a word multiplied by its frequency is a constant (Zipf, 1949). The *rank* of a word is the position of the word in the word list when the list is sorted in decreasing order according to the words' frequencies in the corpus. Graphically, if the frequency of each word is plotted as a function of rank on doubly logarithmic axes, the curve is close to a straight line with slope -1. To achieve a closer fit to the empirical distribution of words, Mandelbrot (1954) derives the following more general relationship between rank and frequency:

$$freq = P(rank + \rho)^{-B}$$

where P, B, and ρ are parameters of a text, that collectively measure the richness of the text's use of words (Manning and Schütze, 1999). Interestingly, the curve in figure 2.13 shows that the relationship between the rank and frequency of templates satisfies Mandelbrot's equation.

12. Chiang (2000) did similar experiments for the tree insertion grammar that he extracted from the PTB. His grammar had 3,626 templates, of which 1,587 occur once. He found that out of 100 randomly selected once-seen templates, 34 results from annotation errors, 50 from deficiencies in the heuristics used by his extraction algorithm, and 4 from performance errors; only 12 appeared to be genuine. It is hard to compare the results of these two experiments because the treebank grammars and the extraction algorithms in his and our experiments were very different.

13. We chose those sections because most state-of-the-art parsers are trained and tested on those sections.

14. The Chinese Treebank had about 100,000 words when our experiments were conducted. Since then, the treebank has been expanded and now it contains more than 500,000 words.

15. Ideally, the coverage of a grammar should be measured as the percentage of sentences for which the *correct* parse trees can be generated by the grammar. However, because it is time consuming to manually check whether the parse trees produced by the grammar contain the correct ones, previous evaluations used a more lenient measure, which is the percentage of sentences for which the grammar will produce at least one parse tree.

16. For instance, in the XTAG grammar, the templates for intransitive verbs such as *come* and the templates for the intransitive usage of ergative verbs such as *break* in the sentence *The window broke* look the same except that the subject in the

former is named NP_0 while the subject in the latter is named NP_1. The subscript 0 implies that the NP is the agent, and the subscript 1 implies that the NP is the theme. On the other hand, the Penn Treebanks do not mark agents and themes, and therefore the templates for *come* and the intransitive usage of *break* will be identical in the treebank grammar. Because subscripts appear only in the XTAG grammar and are ignored when templates are matched, the two slightly different templates in the XTAG grammar will match the same template in G_2.

17. The number 97.2% is the sum of two numbers: the first one is the percentage of matched template tokens (82.1% from Table 2.5). The second number is the percentage of template tokens in T1–T3 (16.8% - 1.7% = 15.1% from Table 2.6).

18. It is possible that the new derivation tree cannot fit together because t and t' are not identical.

19. A *word* in this section, as usual, refers to an inflected word, rather than a lemma. The average number of Supertags per word *type* is calculated as

$$\frac{\sum_{w \in W} stag(w)}{\mid W \mid}$$

The average number of Supertags per word *token* is calculated as

$$\frac{\sum_{w \in W} stag(w) * freq(w)}{\sum_{w \in W} freq(w)}$$

where W is the set of distinct words in a treebank, stag(w) is the number of Supertags that a word w has, and *freq(w)* is the number of occurrences of w in the treebank.

20. For these four numbers, we use PTB's tagset. The numbers would decrease a little bit if we used the reduced tagset instead. A word may appear to have more Supertags (or POS tags) in the treebank than it should due to treebank annotation errors.

21. Notably, the results we report on Srinivas's data, 85.78% on Section 23 and 85.53% on Section 22, are lower than the 92.2% reported in Srinivas (1997), 91.37% in Chen et al. (1999) and 88.0% in Doran (2000). There are several reasons for the differences. First, the size of training data in our experiment is smaller than the one for his previous work, which was trained on Sections 00–24 except for Section 20 and tested on Section 20. Second, we treat punctuation marks as normal words during evaluation because, like other words, punctuation marks can anchor *etrees*, whereas he treated the Supertags for punctuation marks as always correct. Third, he predefined some equivalence classes and used them during evaluations. If the correct Supertag for a word is x, and the output of the Supertagger for that word is y, he did not consider that output to be an error if x and y appeared in the same equivalence class. We suspect that the reason that these Supertagging errors were disregarded is that they might not affect parsing results when the Supertags are combined to form parse trees. 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 that should be tagged with one Supertag is mistagged with the other Supertag, it is likely that the wrong Supertag can still fit with the rest of Supertags in the sentence to produce the correct parse tree. In our experiment, we did not use these equivalence classes.

22. Let R be the root of T^*. If each node t in T^* is split into a *(t.top, t.bot)* pair, and each node e in $ESet$ is split into a pair *(e.top, e.bot)* except that the foot and substitution nodes have only the top part and the root nodes have only the bottom part, then there is a bidirectional function f from the set $\{t.top, t.bot\} - \{R.top\}$ to the set $\{e.top, e.bot\}$. Details can be found in section 5.4.4 in Xia (2001).

23. While the exact definition of *plausibility* could vary depending on the underlying linguistic theory, there are some requirements that most people would agree that a good elementary tree should satisfy: for example, arguments should appear in the same elementary tree as their head; if the category of a phrase is XP (e.g., *VP*), the category of the head of the phrase should be X (e.g., *V*). For the experiment described in this section, we let our linguistic expert use her own judgment to decide on the plausibility of elementary trees.

24. While the algorithm would point out the nodes in *ttrees* that need to be checked, it is up to the user of the tool to decide what kind of modification is needed to fix the errors in the *ttree*.

25. Because most of the readers are more familiar with the English Penn Treebank than the Chinese Penn Treebank, in this example we adopt the annotation convention and the tagset that are used in the English Penn Treebank (except for the tag *AS* for an aspect marker, which does not appear in the English tagset).

References

Charniak, E. (1996). Treebank Grammars. In *Proc. of the 13th National Conference on Artificial Intelligence (AAAI-1996).*

Chen, J., Srinivas, B., and Vijay-Shanker, K. (1999). New Models for Improving Supertag Disambiguation. In *Proc. of the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL-1999).*

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

Chiang, D. (2000). Statistical Parsing with an Automatically-Extracted Tree Adjoining Grammar. In *Proc. of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000).*

Chomsky, N. (1981). *Lectures on Government and Binding.* Foris.

Collins, M. (1997). Three Generative, Lexicalised Models for Statistical Parsing. In *Proc. of the 35th Annual Meeting of the Association for Computational Linguistics (ACL-1997),* Madrid, Spain.

Doran, C. (2000). Punctuation in a Lexicalized Grammar. In *Proc. of 5th International Workshop on TAG and Related Frameworks (TAG+5).*

Doran, C., Hockey, B. A., Sarkar, A., Srinivas, B., and Xia, F. (2000). Evolution of the XTAG System. In Abeillé, A. and Rambow, O., eds., *Tree Adjoining Grammar: Formalism, Computation, Applications*, CSLI Publications.

Jackendoff, R. S. (1977). *X-bar Syntax: A Study of Phrase Structure.* MIT Press.

Joshi, A. K. and Srinivas, B. (1994). Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing. In *Proc. of the 15th International Conference on Computational Linguistics (COLING-1994).*

Krotov, A., Hepple, M., Gaizauskas, R., and Wilks, Y. (1998). Compacting the Penn Treebank Grammar. In *Proc. of the 36th Annual Meeting of the Association for Computational Linguistics (ACL-1998)*, Montreal, Quebec, Canada.

Magerman, D. M. (1995). Statistical Decision-Tree Models for Parsing. In *Proc. of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-1995)*, Cambridge, Massachusetts.

Mandelbrot, B. (1954). Structure formelle des textes et communication. *Word*, 10.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing.* The MIT Press, Cambridge, Massachusetts, USA.

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

Neumann, G. (1998). Automatic Extraction of Stochastic Lexicalized Tree Grammars from Treebanks. In *Proc. of the 4th International Workshop on TAG and Related Frameworks (TAG+4).*

Prasad, R. and Sarkar, A. (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*, Athens, Greece.

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

Sarkar, A. (2001). Applying Co-Training Methods to Statistical Parsing. In *Proc. of the Second Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL-2001)*, Pittsburgh, Pennsylvania.

Sarkar, A. and Joshi, A. K. (1996). Coordination in Tree Adjoining Grammars: Formalization and Implementation. In *Proc. of the 16th International Conference on Computational Linguistics (COLING-1996)*, Copenhagen, Denmark.

Schabes, Y. and Shieber, S. M. (1992). An Alternative Conception of Tree-Adjoining Derivation. In *Proc. of the 30th Annual Meeting of the Association for Computational Linguistics (ACL-1992)*, Newark, Delaware, USA.

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

Skut, W., Krenn, B., Brants, T., and Uszkoreit, H. (1997). An Annotation Scheme for Free Word Order Languages. In *Proc. of 5th International Conference of Applied Natural Language.*

Srinivas, B. (1997). *Complexity of Lexical Descriptions and Its Relevance to Partial Parsing.* PhD thesis, University of Pennsylvania.

Srinivas, B., Sarkar, A., Doran, C., and Hockey, B. A. (1998). Grammar and Parser Evaluation in the XTAG Project. In *Proc. of the Workshop on Evaluation of Parsing Systems*, Granada, Spain.

Xia, F. (1999). Extracting Tree Adjoining Grammars from Bracketed Corpora. In *Proc. of 5th Natural Language Processing Pacific Rim Symposium (NLPRS-1999)*, Beijing, China.

Xia, F. (2001). *Automatic Grammar Generation from Two Different Perspectives.* PhD thesis, University of Pennsylvania.

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

Xia, F., Han, C., Palmer, M., and Joshi, A. K. (2001). Automatically Extracting and Comparing Lexicalized Grammars for Different Languages. In *Proc. of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, Washington.

Xia, F. and Palmer, M. (2000). Evaluating the Coverage of LTAGs on Annotated Corpora. In *Proc. of LREC Satellite Workshop Using Evaluation Within HLT Programs: Results and Trends*, Athens, Greece.

Xia, F., Palmer, M., and Joshi, A. K. (2000a). A Uniform Method of Grammar Extraction and Its Applications. In *Proc. of Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP-2000).*

Xia, F., Palmer, M., Xue, N., Okurowski, M. E., Kovarik, J., Huang, S., Kroch, A., and Marcus, M. P. (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.

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

Zipf, G. K. (1949). *Human Behavior and the Principle of Least Effort.* Hafner.