Automatically Generating Tree Adjoining Grammars from Abstract Specifications

Fei Xia*, Martha Palmer
Dept of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
fxia/mpalmer@linc.cis.upenn.edu
(215)898-9513 (Tel)
(215)898-0587 (Fax)

K. Vijay-Shanker
Dept of Computer and Information Science
University of Delaware
Newark, DE 19716, USA
vijay@cis.udel.edu
(302)831-1952 (Tel)
(302)831-4091 (Fax)

January 21, 2004

*The work was done while the author attended the graduate school at the University of Pennsylvania. The author currently works at the IBM T. J. Watson Research Center, Yorktown Heights, N.Y. 10598, USA.

Abstract

The paper describes a system that can automatically generate tree adjoining grammars from abstract specifications. Our system is based on the use of tree descriptions to specify a grammar by separately defining pieces of tree structure that encode independent syntactic principles. Various individual specifications are then combined to form the elementary trees of the grammar. The system enables efficient development and maintenance of a grammar, and also allows underlying linguistic constructions (such as wh-movement) to be expressed explicitly. We have carefully designed our system to be as language-independent as possible and tested its performance by constructing both English and Chinese grammars, with significant reductions in grammar development time. Provably consistent abstract specifications for different languages also offer unique opportunities for investigating how languages relate to themselves and to each other. For instance, the impact of a linguistic structure such as wh-movement can be traced from its specification to the descriptions that it combines with, to its actual realization in trees. By focusing on syntactic properties at a higher level, our approach allowed a unique comparison of our English and Chinese grammars.

Keywords: natural language processing, grammar development, tree adjoining grammars, tree descriptions

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. Maintenance of grammars and the problems posed by redundancy in grammars are issues that are always faced by developers of large-scale grammars. The research community working in several grammatical frameworks has from the outset addressed these issues and has worked on developing methods and meta-formalisms to properly organize statements of grammatical principles that significantly aid in the development and maintenance of grammars and alleviate problems of redundancy. For instance, the lexical redundancy rules of Lexical Functional Grammar (Kaplan and Bresnan, 1982) represent an elegant method for stating some generalizations at the lexical level and reducing redundancy. The meta-rules and feature co-occurrence restrictions are two ways by which Generalized Phrase Structure Grammar (GPSG) (Gazdar et al., 1985) states grammatical generalizations. Head-driven Phrase Structure Grammars (HPSG) (Pollard and Sag, 1994) is a grammatical framework for which considerable effort has been made to state grammatical principles in an elegant way and eliminate redundancy. These include hierarchical organization of principles and lexical rules. In addition, meta-formalisms for Typed Feature Structures (TFS) (Carpenter and Penn, 1999; Martin, 1994) have been employed successfully to encode large HPSG grammars. In the ALE system (Carpenter and Penn, 1999), grammars are encoded as definite clause grammars with typed feature structures as terms; while in the TFS system (Martin, 1994), unification grammars are organized as inheritance networks of typed feature structures. Such formalisms offer many advantages such as abstraction, information sharing, modularity and reusability of grammatical descriptions. In contrast, the Lexicalized Tree Adjoining Grammar (LTAG) community has paid much less attention to to the development or maintenance of LTAG grammars for natural languages, and the emphasis in LTAGs on tree structures rather than typed feature structures precludes a straightforward adaptation of the generalization techniques incorporated into HPSG.

To overcome this deficiency for LTAG, we designed a grammar development system named LexOrg, which automatically generate LTAG grammars from abstract specifications. The system is based on the ideas expressed in (Vijay-Shanker and Schabes, 1992), for using tree descriptions in specifying a grammar by separately defining pieces of tree structure that encode independent syntactic principles. Various individual

specifications are then combined to form the elementary trees of the grammar. We have carefully designed our system to be as language-independent as possible and tested its performance by constructing both English and Chinese grammars, with significant time reductions. The system not only enables efficient development and maintenance of a grammar, but also allows underlying linguistic constructions (such as wh-movement) to be expressed explicitly.

One important factor that affects the design of LexOrg is the fact that most researchers working in the LTAG community have viewed LTAG strictly as a grammar formalism and have separated the linguistic principles behind the design of a particular grammar from the formal LTAG framework, at least to a much larger extent than is the case with the above-mentioned other grammatical systems. In other words, the LTAG framework is viewed somewhat similarly to context-free grammars, and just as there is no single widely-accepted context-free grammar for a particular language, there is not necessarily a single definitive LTAG for any language.[1] This aspect of LTAG has influenced our design of LexOrg. As designers of LexOrg, we wish to provide a similar flexibility to grammar designers so that they can express their own linguistic intuitions in the manner that they deem most appropriate. Therefore, our task is to provide mechanisms that we believe will be useful for the expression of a variety of grammar theories and intuitions.

The paper is organized as follows. In Section 2, after giving a brief overview of the LTAG formalism, we outline the way that LexOrg attacks the redundancy problem in LTAG grammars. In Section 3, we define descriptions, trees, and four classes of descriptions. In Sections 4 to 6, we describe the three main components of LexOrg. Section 7 includes a report on our experiments using LexOrg to generate grammars for English and Chinese. Finally, Section 8 contains our comparison of LexOrg with related work, including Typed Feature Structures, Becker's HyTAG system (Becker, 1994), a system by Evans, Gazdar and Weir (Evans et al., 1995) implemented in DATR (Evans and Gazdar, 1989), and Candito's system (Candito, 1996).

[1] While the XTAG grammar for English (XTAG-Group, 1995; XTAG-Group, 1998) is probably the most widely-known LTAG grammars, it is by no means the only large-scale grammar for English. Although it slightly departs from its original linguistic inspiration, this grammar is based on many of the linguistic ideas developed in the works of Kroch, Joshi and Frank (Kroch and Joshi, 1985; Kroch and Joshi, 1987; Kroch, 1989; Joshi and Schabes, 1997; Frank, 2002), which themselves are largely influenced by Chomskyan linguistics. In contrast, a large-scale grammar was developed in the Lexsys system, which differs in its linguistic basis and also the notion of what is localized in the elementary trees of LTAG. In addition, a large-scale LTAG for English was produced by compiling out various HPSG principles and lexicons in the form of LTAG elementary trees following the method described in (Kasper et al., 1995).

2 The issue of redundancy in LTAG grammars

In this section, we elaborate on the problem of ensuring consistency while scaling up large grammars, and give an overview of how LexOrg addresses this problem. We begin the section with a brief overview of the LTAG formalism. A more comprehensive discussion of the formalism can be found in (Joshi and Schabes, 1997) and the citations in the book.

2.1 The LTAG formalism

LTAGs are based on the Tree Adjoining Grammar (TAG) formalism developed by Joshi, Levy, and Takahashi (1975; 1997). In the last decade, LTAGs have been widely used in many NLP tasks, such as parsing (Schabes, 1990; Srinivas, 1997; Sarkar, 2001), semantics (Joshi and Vijay-Shanker, 1999; Kallmeyer and Joshi, 1999), lexical semantics (Palmer et al., 1999; Kipper et al., 2000), discourse (Webber and Joshi, 1998; Webber et al., 1999), machine translation (Palmer et al., 1998), and generation (Stone and Doran, 1997; McCoy et al., 1992).

As a constrained mathematical formalism, LTAG is more powerful than context-free grammar (CFG) as it can generate *mildly* context-sensitive languages such as the language $\{a^n b^n c^n d^n \mid n \geq 0\}$ and handle cross-serial dependencies in Dutch (Joshi, 1985). This extra generative power comes from the fact that the primitive elements of an LTAG are trees, rather than context-free rules. Each tree in an LTAG grammar is called an *elementary tree* and is anchored by a lexical item. There are two types of elementary trees: initial trees and auxiliary trees. Each auxiliary tree has a unique leaf node, called the *foot* node, which has the same category as the root. In both types of trees, leaf nodes other than anchors and foot nodes are called *substitution* nodes. One important property of LTAG is its extended domain of locality; that is, an elementary tree encapsulates all and only the arguments of the anchor, thus providing extended locality over which the syntactic and semantic constraints can be specified. Figure 1 shows an elementary tree for the verb *break*, in which the V is the anchor of the tree. The arguments of the verb — the subject NP_0 and the object NP_1 — are in the same tree,[2] and they are substitution nodes as marked by $\downarrow$.

Elementary trees are combined by two operations: substitution and adjoining. In the substitution op-

[2]The subscripts in elementary trees are used to distinguish nodes with the same syntactic categories, such as NP_0 and NP_1 in Figure 1. They have no linguistic content.

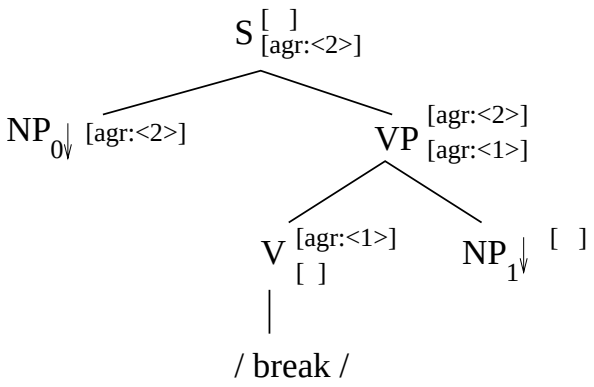


Figure 1: An elementary tree for the verb *break*

eration, a substitution node in a tree is replaced by another tree whose root has the same category as the substitution node. In an adjoining operation, an auxiliary tree is inserted into another tree. The root and the foot nodes of the auxiliary tree must have the same category as the node at which the auxiliary tree adjoins. The resulting structure of the combined trees is called a *parse tree* or a *derived tree*. The history of the combination process is recorded as a *derivation tree*.

In Figure 2, the four elementary trees in (a) are anchored by words in the sentence *John often breaks windows*. α_1 — α_3 are initial trees, and β_1 is an auxiliary tree. Foot and substitution nodes are marked by $*$ and $\downarrow$, respectively. To generate the derived tree for the sentence, α_1 and α_3 substitute into the nodes NP_0 and NP_1 in α_2 respectively, and β_1 adjoins to the VP node in α_2, thus forming the derived tree in (b). The dashed and dotted arrows between the elementary trees stand for the substitution and adjoining operations, respectively. The history of the composition of the elementary trees is recorded in the derivation tree in (c). In a derivation tree, a dashed line is used for a substitution operation and a dotted line for the adjoining operation.

In the LTAG framework, a feature structure is associated with each node in an elementary tree (Vijay-Shanker, 1987), and it consists of a top part and a bottom part. A feature structure contains information about how nodes interact with other nodes in the tree, and is used to specify linguistic constraints, such as the number/person agreement between a verb and its subject in English. When elementary trees are combined by substitution and adjoining operations, the feature structures of merged nodes are unified; and in a derived tree, the top part and the bottom part of every node must agree. For instance, the agreement between a verb phrase VP and its subject NP_0 in English can be expressed as a feature equation

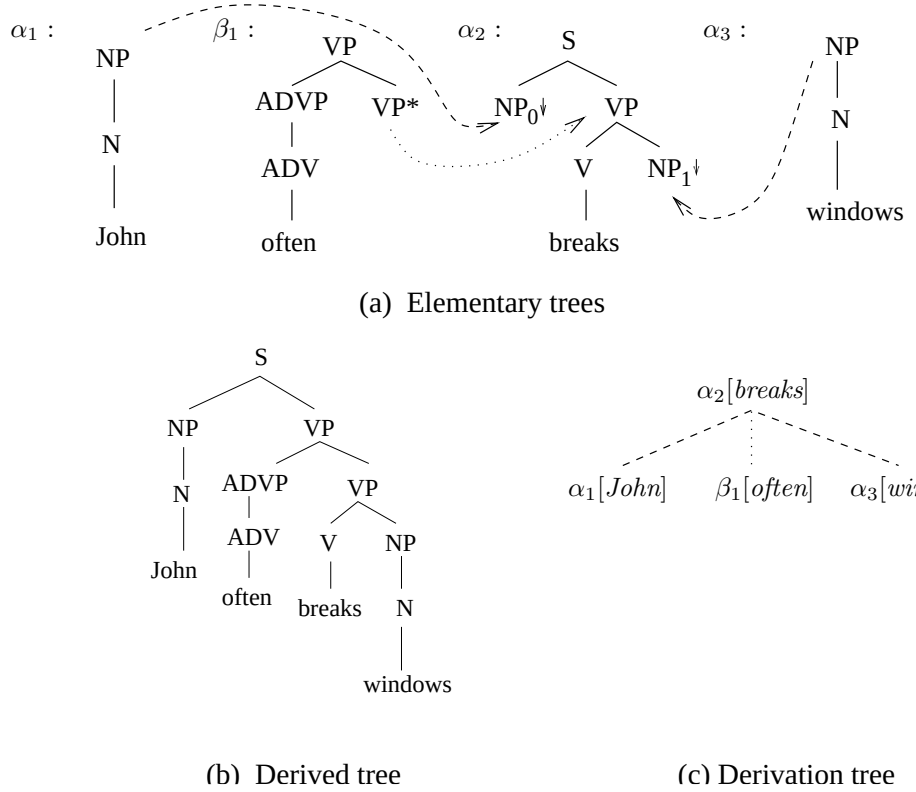


Figure 2: Elementary trees, derived tree and derivation tree for the sentence *John often breaks windows.*

$VP.t :< agr >= NP_0.t :< agr >$, where $X.t$ and $X.b$ are the top part and the bottom part of the feature structure for a node X respectively, and agr is a feature name. In Figure 1, this equation is displayed as $[arg :< 2 >]$ next to the NP_0 and VP nodes; similarly, the equation $VP.b :< agr >= V.t :< agr >$ is displayed as $[arg :< 1 >]$ next to the VP and V nodes in the elementary tree.[3] From now on, for the sake of simplicity, we shall not show the feature structures in elementary trees unless necessary.

In practice, it is customary to store an elementary tree as a (word, template) pair. A *template* is an elementary tree with the lexical item removed. The symbol @ in a template marks the place where the lexical item should be inserted back. For instance, the elementary tree in Figure 1 is stored as a pair (*break*, #1), where the template #1 is given in Figure 3. To add elementary trees for other ergative verbs such as *melt* and *sink*, we only need to add pairs (*melt*, #1) and (*sink*, #1), and the template #1 is stored only once. In this paper, we mainly discuss the generation of templates, rather than the generation of elementary trees. Templates with the same subcategorization frame are grouped into a template set, called *a tree family*

[3]The values of $VP.t :< agr >$ and $VP.b :< agr >$ in this elementary tree do not have to agree because other elementary trees could adjoin to this VP node and *split* this node into two nodes.

(XTAG-Group, 2001). A *subcategorization frame* specifies the categories of a head and its arguments, the positions of arguments with respect to the head. For instance, *(NP0 V NP1)* is a subcategorization frame, where *V* is the head, *NP0* is a left argument and *NP1* is a right argument.[4] Figure 3 shows some templates in two tree families: the top four elementary trees are for the verbs with the subcategorization frame *(NP0 V NP1)*, and the bottom three elementary trees are for the verbs with subcategorization frame *(NP1 V)*. Of course, these are not the only trees in the tree families associated with these subcategorization frames. In the XTAG grammar for English, for example, there are 19 templates in the transitive verb's tree family.[5]

2.2 The impact of redundancy on development and maintenance

LTAG is an appealing formalism for representing various phenomena (especially syntactic phenomena) in natural languages because of its linguistic and computational properties such as the Extended Domain of Locality, stronger generative capacity and lexicalized elementary trees. Because templates in an LTAG grammar often share some common structures, manually building an LTAG grammar presents a serious problem, as illustrated by the following example.

Figure 3 shows seven templates for ergative verbs such as *break*. The top four templates show the syntactic environments for ergative verbs when they act as transitive verbs, such as *break* in *John often breaks windows*. #1 is for a declarative sentence, #2 and #3 are for wh-questions, and #4 is for an infinitival clause.[6] There are two templates for wh-questions because the moved constituent can be either the subject (as in #2) or the object (as in #3). The symbol ϵ in a template is used to mark an *empty category*, such as a trace in a wh-question. The bottom three templates show the syntactic environments for ergative verbs when they act as intransitive verbs, such as *break* in *The window broke*. The top four templates form a tree family, while the bottom three form another.

Among these seven templates, #1, #2, #3, and #4 all share the structure in Figure 4(a), templates #2, #3, and #6 all have the structure in Figure 4(b), templates #4 and #7 both have the structure in Figure

[4] Just like in elementary trees, the subscripts in subcategorization frames are used to distinguish nodes with the same syntactic categories, such as NP_0 and NP_1 in the subcategorization frame (NP_0 V NP_1). They have no linguistic content. Also, strictly speaking, to distinguish arguments from the head in a subcategorization frame, the head should be followed by a symbol @; however, because the heads of all the subcategorization frames appeared in this paper are verbs, we drop the symbol @ from our notation.

[5] The XTAG grammar (XTAG-Group, 1998; XTAG-Group, 2001) is a large-scale LTAG grammar for English, which has been manually created and maintained by a group of linguists and computer scientists at the University of Pennsylvania since the early 1990s.

[6] Template #4 is used to handle sentences such as "*John brought a stone to break the window*", where the infinitival clause "*to break the window*" modifies the verb phrase "*brought a stone*" in the main clause.

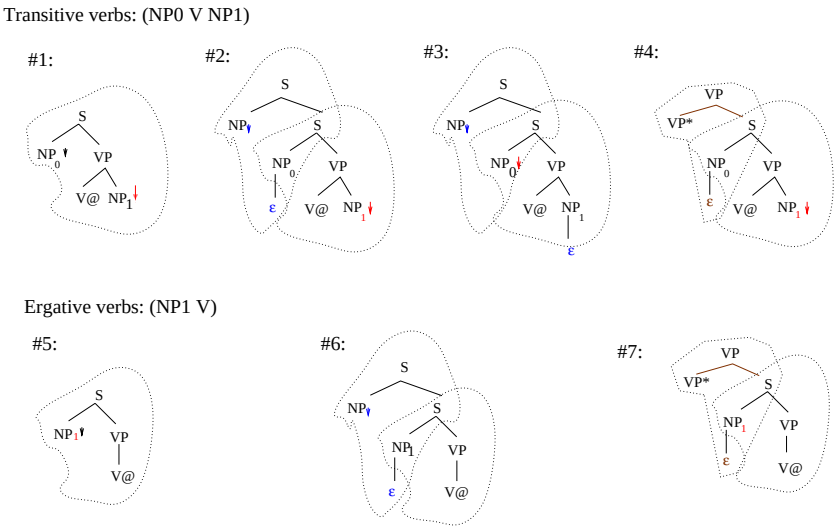


Figure 3: Templates in two tree families

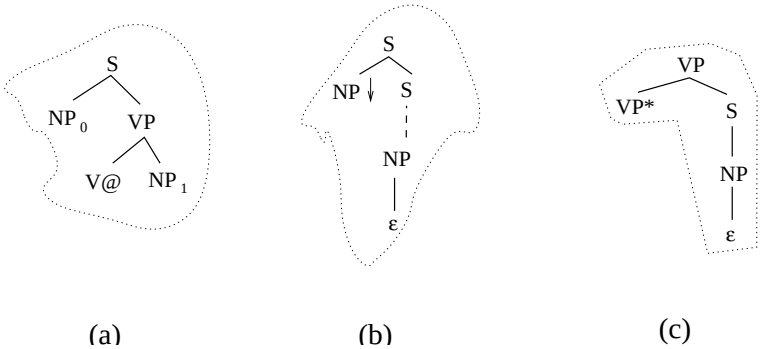


Figure 4: Structures shared by the templates in Figure 3

4(c), and so on. The dashed line in Figure 4(b) between the lower *S* and the node *NP* indicates that the *S* node dominates the *NP* node, but it is not necessarily the parent of the *NP*.

As the number of templates increases, building and maintaining templates by hand presents two major problems. First, the reuse of tree structures in many templates creates redundancy. To make a single change in a grammar, all the related templates have to be manually checked. The process is inefficient and cannot guarantee consistency (Vijay-Shanker and Schabes, 1992). For instance, a single change (such as the addition of a new feature to an *S* node) could affect a few hundred templates in a large-scale grammar. In Figure 3, if the analysis for wh-movement is changed, then templates #2, #3, and #6 have to be manually modified. Of course, they will be just some of the templates in that grammar that will need to be altered. In the current XTAG English grammar developed at the University of Pennsylvania, a change in the wh-movement rule

will require changes to be made to 169 templates from 54 tree families.[7] Second, the underlying linguistic information is not expressed explicitly. For instance, the analysis of wh-movement is expressed implicitly in three templates in Figure 3. As a result, from the grammar itself (i.e., hundreds of templates plus the lexicon), it is hard to grasp the characteristics of a particular language, to compare languages, and to build a grammar for a new language given existing grammars for other languages. LexOrg was designed to address these problems.

2.3 An overview of LexOrg

At first sight, the problems that we just described seem to be caused by the sharing of structures among templates. However, a closer look reveals that the problems exist only because the templates are built manually. If there exists a tool that combines these common structures to generate templates automatically (as illustrated in Figure 5), then the task of the grammar designers changes from building templates to building these common structures, providing an elegant solution. First, one can argue that the common structures form the appropriate level for stating the linguistic generalizations. Considering that these common structures are much smaller and simpler than templates and the number of the former is much less than that of the latter, the grammar development time will also be reduced significantly. Second, if grammar designers want to change the analysis of a certain phenomenon (e.g., wh-movement), they need to modify only the structure that represents the phenomenon (e.g., the structure in Figure 5(b) for wh-movement). The modifications in the structure will be automatically propagated to all the templates that subsume the structure, thus guaranteeing consistency among the templates. Third, the underlying linguistic information (such as wh-movement) is expressed explicitly, making it easy to grasp the main characteristics of a language and to compare languages.

All of these advantages will be derived if the grammar designer is able to state the linguistic principles and generalizations at the appropriate level. That is, the domain of the objects being specified must be only large enough to state these principles. While the enlarged domain of locality in templates is touted as one of the fundamental strengths of LTAG, it must be noted that from the grammar development point of view each template expresses several (often independent) principles. Thus, in coming up with a template, the designer

[7] The numbers are based on the XTAG grammar released on Feb 24, 2001, and the grammar can be downloaded from its web site http://www.cis.upenn.edu/~xtag.

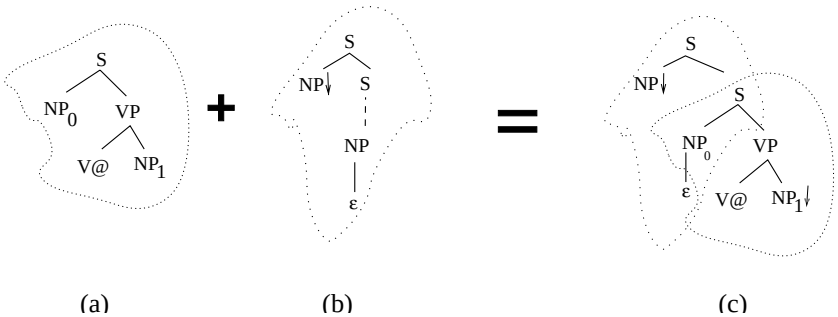


Figure 5: Combining descriptions to generate templates

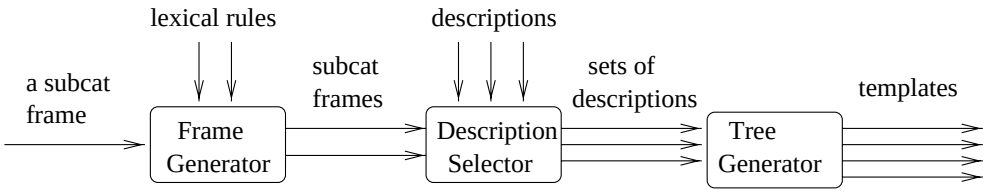


Figure 6: The architecture of LexOrg

has to consider the instantiation of several principles that could interact in some cases and also instantiate the same principles multiple times (sometimes hundreds of times). We believe that this aspect makes the grammar development process unnecessarily error-prone and cumbersome. Our aim in developing LexOrg is to let the grammar designer define individual grammar principles within a domain that is appropriate for that principle. (Roughly, these domains and the instantiation of principles would represent the shared structures found among templates in existing grammars as discussed earlier.) The LexOrg system then assumes the burden of considering what set of principles should fit together to make up a template and also considers the interactions and consistency of such a set of principles. Thus, the process of grammar development or prototyping can be significantly simplified, thereby speeded up and made less error-prone. Over a period of time as the grammar is developed to further its coverage, certain principles are bound to be modified. No matter how small the modification is, ensuring that the possible effect on all the templates already designed is properly accounted for by manually checking the templates is an onerous task. However, with a tool such as LexOrg, the focus is correctly placed on the principle. The propagation of this principle and ensuring the consistency of its interactions with other principles is now mechanized. Additionally, we have argued in (Xia et al., 1999) that LexOrg can be used to produce concise abstract specifications of individual languages which facilitate cross-lingual linguistic comparisons.

In LexOrg, instead of manually creating templates, a grammar designer needs to provide the following specifications for a language: "subcategorization" frames, lexical subcategorization rules, and four different kinds of tree descriptions: head-projection, head-argument, modification, and syntactic variation descriptions. These specifications relate closely to the different aspects of LTAG elementary trees and the notion of tree families. The *subcategorization frames* associated with different lexical items will specify which arguments of the lexical items that the designer intends to localize within the elementary trees. Together with *the head-projection* and *head-argument descriptions* (where the grammar designer expresses how the lexical heads project and how they combine with their "subcategorized" arguments), they will cause LexOrg to produce the basic tree structure for each subcategorization frame. Note we use the term "subcategorization" here to mean what the designer intends to be localized with the lexical head. Lexical items with the same subcategorization frames can thus be understood to share the same tree family. The *lexical subcategorization rules* allow the grammar designer to specify the processes that they consider to be lexical which define related subcategorization frames. For example, the difference between the passive and active forms can be stated using this machinery. In addition to head-projection and head-argument descriptions, there are two additional kinds of descriptions: *modification descriptions* and *syntactic variation descriptions. Modification descriptions* are used to describe the other kind of elementary tree in LTAG, modifier auxiliary trees (such as β_1 in Figure 2), which are used to represent the tree structures for various forms of modification. *Syntactic variation descriptions* are to be used to account for the design of the rest of the elementary trees that are not obtained from mere projections of the basic subcategorization frames or frames derived from using lexical subcategorization rules. In the next sections, we shall define each kind of specification in more detail.

We believe that most linguistic theories in some form or the other use these different types of grammatical mechanisms. We have separated out into the different kinds of specifications as described above because of their relationship to the different aspects of elementary trees and tree families now familiar to the LTAG community. Nevertheless, in spite of this connection, we make no a priori assumption about how a grammar designer should use these types of grammatical specification methods. For example, the treatment of wh-movement can be specified via syntactic variation descriptions or as a lexical process and hence by using lexical subcategorization rules; the descriptions can be structured hierarchically or could have a relatively

flat organization. Because we give grammar designers such freedom in choosing appropriate grammatical specification methods, evaluation of one particular set of specifications (such as the ones given in Section 7) is not central to LexOrg's evaluation. In the next few sections, we will describe pieces of a particular specification of an LTAG grammar. However, this should be understood as an attempt to suggest the usefulness of LexOrg and to provide examples of using the different aspects of LexOrg. Hence, by no means do we intend for this specification to suggest any particular grammatical principle to be associated with LTAG nor even how principles have to be stated in LexOrg.

Figure 6 shows the architecture of the LexOrg system which has three components: a Frame Generator, a Description Selector, and a Tree Generator. The input to the system are subcategorization frames, lexical subcategorization rules, and tree descriptions. The Frame Generator (described in Section 6) accepts the subcategorization frames and lexical subcategorization rules and for each subcategorization frame it considers all the applicable lexical subcategorization rules to produce a set of subcategorization frames in a format that is appropriate for later stages. The Description Selector (described in Section 5) automatically identifies the set of descriptions (used to construct the tree templates) appropriate for each template for each subcategorization frame. Finally, the Tree Generator (Section 4) produces the templates corresponding to the selected descriptions and subcategorization frame. In the next section, we first describe the language used in specifying the grammatical descriptions and consider four different classes of descriptions that a grammar designer may provide.

3 Tree descriptions

Tree descriptions (or *descriptions* for short) were introduced by Vijay-Shanker and Schabes (1992) in a scheme for efficiently representing an LTAG grammar. Rogers and Vijay-Shanker (1994) later gave a formal definition of (tree) *descriptions*. We extended their definition to include features and further divided *descriptions* into four classes: head-projection descriptions, head-argument descriptions, modification descriptions, and syntactic variation descriptions. This section presents the characterizations of each of these classes in detail.

3.1 The definition of a *description*

In (Rogers and Vijay-Shanker, 1994), *descriptions* are defined to be formulae in a simplified first-order language L_K, in which neither variables nor quantifiers occur. L_K is built up from a countable set of constant symbols K, three predicates (*parent*, *domination*, and *left-of* relations), the equality predicate, and the usual logical connectives ($\wedge$, $\vee$, $\neg$). We extended their definition to include features because, in the LTAG formalism, feature structures are associated with the nodes in a template to specify linguistic constraints. Feature specifications, in a PATR-II like format, are added to descriptions so that when descriptions are combined by LexOrg, the features are carried over to the resulting templates. In addition to any feature that a user of LexOrg may want to include, there are two predefined features for each constant symbol in K: one is *cat* for *category*, the value of which can be N (noun), NP (noun phrase), and so on; the other feature is *type*, which has four possible values: $foot$, *anchor*, *subst*, and *internal*. The first three values are for the three types of leaf nodes in a template: foot node, anchor node, and substitution node, and the last value is for all the internal nodes in a template.

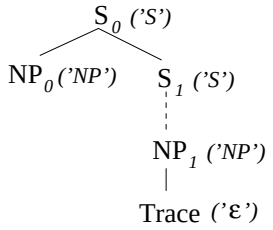
Figure 7(a) shows a description in this logical form, where $\lhd$, $\lhd^*$, and $\prec$ stand for *parent*, *domination*, and *left-of* predicates, respectively; *cat* stands for *category* and is a feature. For instance, $NP_0.cat = \ 'NP'$ means that NP_0 has a feature named *cat* whose value is NP (i.e., noun phrase). Most descriptions used by LexOrg can be represented in a tree-like figure. Figure 7(b) is the graphical representation for the same description. In this graph, dashed lines and solid lines stand for *domination* and *parent* predicates, respectively. The values of some features of nodes (such as the category of a node) are enclosed in parentheses. The graphical representation is more intuitive and easier to read, but not every description can be displayed as a graph because a description may use negation and disjunctive connectives. In the following sections, we shall use the graphical representation when possible and use the logical representation in other cases.

3.2 The definition of a *tree*

According to (Rogers and Vijay-Shanker, 1994), a *tree* is a structure which interprets the constants and predicates of L_K such that the interpretation of the predicates reflects the properties of the trees. A tree is said to *satisfy* a description if the tree as a structure satisfies the description as a formula in L_K. As we have extended the definition of description to include features, we also placed additional requirements on

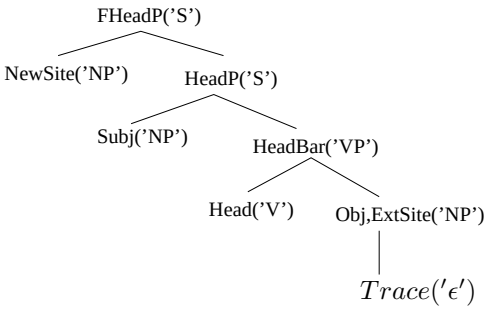
$(S_0 \lhd NP_0) \wedge (S_0 \lhd S_1)$
$\wedge (NP_0 \prec S_1) \wedge (S_1 \lhd^* NP_1)$
$\wedge (NP_1 \lhd Trace) \wedge (S_0.cat = \ 'S')$
$\wedge (NP_0.cat = \ 'NP') \wedge (S_1.cat = \ 'S')$
$\wedge (NP_1.cat = \ 'NP') \wedge (Trace.cat = \ '\epsilon')$

(a) the logical representation

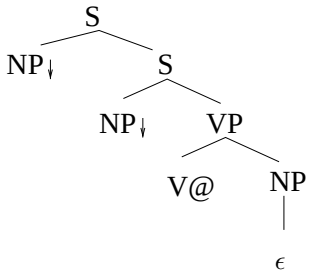


(b) the graphical representation

Figure 7: Two representations of a description



(a) a tree



(b) the corresponding template

Figure 8: A tree and the template that is built from the tree

trees with respect to features. For instance, the category of every node in a *tree* must be specified. For more details about this revision, please see Section 4.3.3. of (Xia, 2001).

From each satisfiable description, we can recover a representation of a minimal model. For example, Figure 8(a) represents a model for the description given in Figure 7. In this representation, a node has the form $\{k_i\}(\{f_m = v_m\})$, where $\{k_i\}$ is a list of node names, and v_m is the value of a feature f_m. For simplicity, we often omit from this graphical representation the curly brackets and all the features except the category of a node. When $\{k_i\}$ has more than one member, it means that several nodes from different descriptions are merged in the tree. In Figure 8(a), one such case is *Obj,ExtSite('NP')*. As we shall show later, *Obj* comes from a head-argument description, while *ExtSite* comes from a syntactic variation description. The two names refer to the same node in the tree. In Section 4, we shall show that it is trivial to build a unique template from such a representation.

3.3 Four classes of descriptions

In Section 4, we shall show that a component of LexOrg, namely the Tree Generator, generates templates (i.e., elementary trees with the lexical item removed) from descriptions. Because the goal of LexOrg is

to build grammars for natural languages, rather than any arbitrary LTAG grammar, descriptions used by LexOrg should contain all the syntactic information that could appear in the templates for natural languages. In this section, we identify four types of syntactic information in a template, and define a class of descriptions for each type of information.

3.3.1 Head-projection descriptions

An important notion in many contemporary linguistic theories such as X-bar theory (Jackendoff, 1977), GB theory (Chomsky, 1981), and HPSG (Pollard and Sag, 1994) is the notion of a *head*. A head determines the main properties of the phrase that it belongs to. A head may project to various levels, and the head and its projections form a projection chain.

The first class of description used by LexOrg is called a *head-projection description*. It gives the information about the head and its various projections. For instance, the description in Figure 9(a) says that a verb projects to a *VP*, and the *VP* projects to an *S*. Typically this should be straightforwardly derivable from head projection principles as found in X-bar theory or similar intuitions expressed in GPSG or HPSG. But in order to give the flexibility to a grammar designer to use any appropriate linguistic theory and, for example, to use any choice in the number of projection levels and categories, we do not implement the derivations from any specific linguistic principles, but rather expect the grammar designer to state the head-projection descriptions explicitly.

3.3.2 Head-argument descriptions

A head may have one or more arguments. For instance, a transitive verb has two arguments: a *subject* and an *object*. The second class of description, the *head-argument description*, specifies the number, the types, and the positions of arguments that a head can take, and the constraints that a head imposes on its arguments. For instance, the description in Figure 9(b) says that the subject — a left argument of the head — is a sister of the *HeadBar*.[8] In this case, the feature equation in the description, given below the tree in Figure 9(b), specifies that the subject and the *HeadBar* must agree with respect to number, person, and so on. The description in Figure 9(c) says that a head can take an *NP* argument, which appears as a right sister

[8] As a user of LexOrg, a grammar designer has the freedom to choose the linguistic theory to be incorporated in an LTAG grammar. In the examples given in this paper (such as in Figure 9), we do not strictly follow the X-bar theory or the GB theory. We name some nodes as *HeadBar* and *HeadP* only for the sake of convenience.

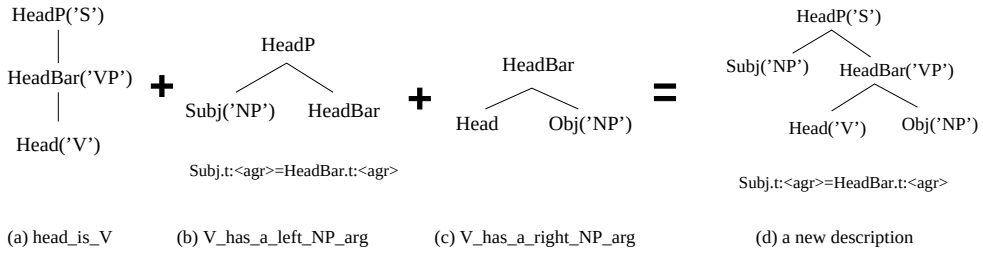


Figure 9: Subcategorization descriptions

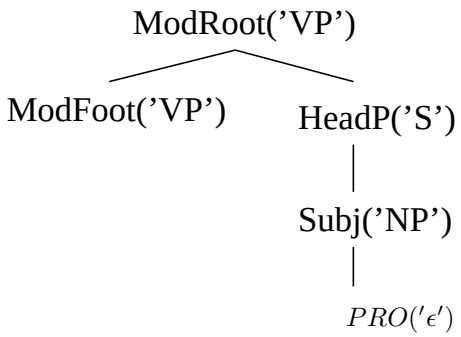


Figure 10: A description for purpose clauses

of the head. Combining head-projection and head-argument descriptions forms a description for the whole subcategorization frame, as in Figure 9(d); therefore, we use the term *subcategorization* description to refer to descriptions of either class. As we shall describe in Section 5, the Description Selector is responsible for choosing the right descriptions appropriate for a given subcategorization frame. Again, for reasons similar to the one that we used at the end of the previous section on head-projection descriptions, for flexibility in choosing structures for head-argument realizations, we do not include in LexOrg a reliance on any specific linguistic principle governing descriptions of head-argument structures.

3.3.3 Modification descriptions

A syntactic phrase can be modified by other phrases. The third class of description, called a *modification description*, specifies the type and the position of a modifier with respect to the modifiee, and any constraint on the modification relation. For instance, the description in Figure 10 says that a clause can modify a verb phrase from the right,[9] but the clause must be infinitival (as indicated by the PRO subject).

The modification descriptions are expected to describe the so-called modifier auxiliary trees in the re-

[9] In the sentence "*John brought a stone to break the window*", the infinitival clause "*to break the window*" modifies the *VP* "*brought a stone*". One may choose the analysis where the infinitival clause modifies the whole main clause "*John brought a stone*", instead of just the *VP* "*brought a stone*". To account for this analysis, we only have to change the categories of *ModRoot* and *ModFoot* from *VPs* to *Ss*.

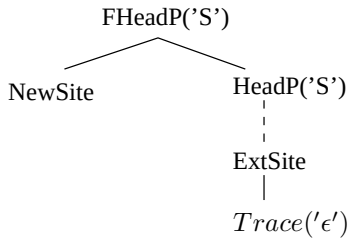


Figure 11: A description for wh-movement

sulting LTAG grammar, which are used to express modification in LTAG.

3.3.4 Syntactic variation descriptions

Head-projection and head-argument descriptions together define a basic tree structure for a subcategorization frame, which forms a subtree of every template in the tree family for that subcategorization frame. For instance, the structure in Figure 9(d) appears in every template in the transitive tree family, as shown on the top part of Figure 3. In addition to this basic structure, a template may contain other structures that represent syntactic variations such as wh-movement and argument drop. For example, template #2 in Figure 3 can be decomposed into two parts: the first part, as in Figure 4(a), is the basic structure that comes from head-projection and head-argument descriptions; the second part, as in Figure 4(b), comes from the description in Figure 11. We call this description a *syntactic variation description*, as it provides information on syntactic variations such as wh-movement. This description says that, in wh-movement, a component is moved from a position *ExtSite* under *HeadP* to the position *NewSite*, as indicated by the feature equation $NewSite.t :< trace >= ExtSite.t :< trace >$; *NewSite* is the left sister of *HeadP*; both *NewSite* and *HeadP* are children of *FHeadP*; both *FHeadP* and *HeadP* are of the category *S*.

Note that LexOrg allows descriptions to "inherit" from other descriptions; that is, a grammar designer has the flexibility of specifying a description as an instantiation or a further specification of other descriptions, which represent more general principles (such as X-bar theory couched in this framework). For example, a grammar designer may choose to create a description *head_has_an_arg* for the head-complement structure in X-bar theory, in which the position of the argument with respect to the head and the categories of the head and the argument are unspecified. The description is specialized further in giving the position of the argu-

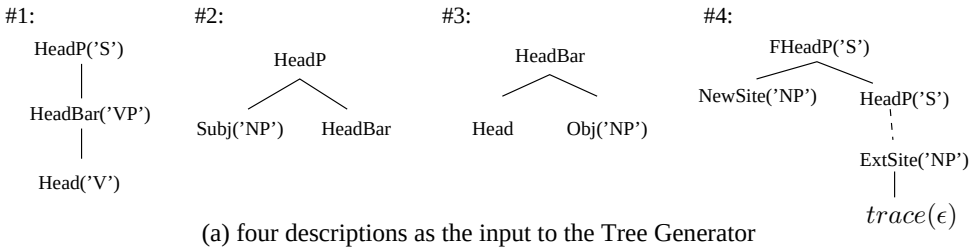
ment, resulting in a new description *head_has_a_right_arg*. The latter description can be further specialized for the case where the argument has a category of *NP*, yielding a new description *head_has_a_right_NP_arg*, and for the case where the argument is an *S*, yielding another description *head_has_a_right_S_arg*. Similarly, the description in Figure 11 can form the basis for movement specification in the grammar and can be further instantiated to cover not only wh-movement but also relative clauses, if desired. Using the inheritance among descriptions may reduce the redundancy among descriptions. For instance, if grammar designers later decide to change the representation for the head-complement structure, they need to change only the description *head_has_an_arg*, not the descriptions that inherit from this description. One final note about the inheritance relation among descriptions: while we (as the creators of LexOrg) encourage grammar designers to take advantage of this feature of LexOrg to make their descriptions more concise and hierarchical, we still allow a grammar designer to create all descriptions such that they are all "atomic" and there is no inheritance among them.

To summarize, we have discussed four classes of descriptions. In Section 5, we shall show that a component of LexOrg, namely the Description Selector, chooses descriptions according to their classes; that is, it will create sets of descriptions such that each set includes one head-projection description, zero or more head-argument descriptions, zero or one modification descriptions, and zero or more syntactic variation descriptions.

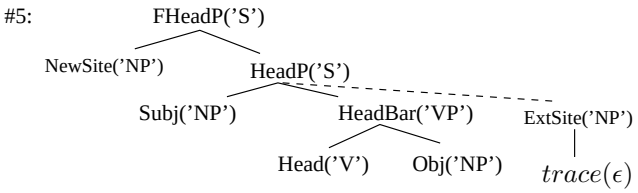
4 The Tree Generator

The most complex component of LexOrg is called the *Tree Generator* (*TreeGen*), which takes a set of descriptions as input and generates a set of templates as output. This is done in three steps: first, TreeGen combines the input set of descriptions to get a new description; second, TreeGen builds a set of trees such that each tree in the set satisfies the new description and has the minimal number of nodes; third, TreeGen builds a template from each tree in the tree set. In Figure 12, the descriptions in (a) are the input to TreeGen. Combining them results in a new description in (b).[10] There are many trees that satisfy this new description, but the two trees in (c) are the only ones with the minimal number of nodes. From these two trees, TreeGen builds two templates in (d). In this section, we explain each step in detail.

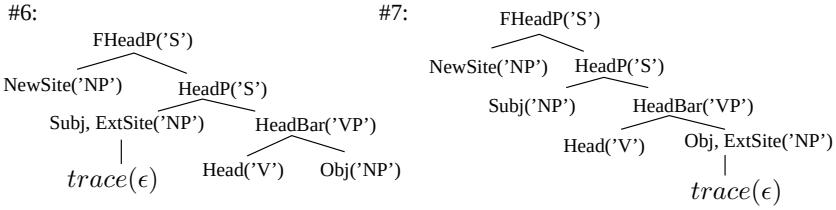
[10]Notice that in Figure 12(b) the position of *ExtSite* with respect to *Subj* and *HeadBar* is not specified.



(a) four descriptions as the input to the Tree Generator



(b) the new description that combines the four descriptions in (a)



(c) trees generated from the new description

(d) the templates as the output of the Tree Generator

Figure 12: The function of the Tree Generator

4.1 Step 1: Combining descriptions to form a new description

The Description Selector selects a set of descriptions that might potentially form one or more templates. TreeGen combines such a set of descriptions to form a new description. Recall that a description is a well-formed formula in a simplified first-order language. Given a set of descriptions $\{\phi_i \mid 1 \leq i \leq n\}$, the new description ϕ, which combines $\{\phi_i\}$, is simply the conjunction of ϕ_i; that is, $\phi = \phi_1 \wedge \phi_2 ... \wedge \phi_n$.

4.2 Step 2: Generating a set of trees from the new description

In the second step, TreeGen generates a set of trees, $TreeSet_{min}(\phi)$, for the new description ϕ. Let $TreeSet(\phi)$ be the set of trees that satisfies ϕ and $NumNodes(T)$ be the number of nodes in a tree T, then $TreeSet_{min}(\phi)$ is defined to be the subset of $TreeSet(\phi)$ in which each tree has the minimal number of nodes; that is,

$$TreeSet_{min}(\phi) = \{T \in TreeSet(\phi) \mid NumNodes(T) = min_{T^* \in TreeSet(\phi)} NumNodes(T^*)\}$$

With a little abuse of notation, we also use $NumNodes(\phi)$ to represent the number of nodes occurring in a description ϕ. According to our definition of *tree*, each node in a tree must have a category; therefore, each tree in $TreeSet(\phi)$ can have at most $NumNodes(\phi)$ nodes. Because $NumNodes(\phi)$ is finite for each ϕ, $TreeSet(\phi)$ and its subset $TreeSet_{min}(\phi)$ are finite too. As a result, $TreeSet_{min}(\phi)$ can be calculated by the following naive algorithm: first, initialize i to 1; second, generate a set $TS(i)$ that includes all the trees with i nodes; third, put into the set $TreeSet_{min}$ all the trees in $TS(i)$ that satisfy ϕ; if $TreeSet_{min}$ is empty, increase i by one and repeat the second and third steps until $TreeSet_{min}$ is not empty or i is more than $NumNodes(\phi)$. Because $NumNodes(\phi)$ is finite for any ϕ, the algorithm will always terminate; furthermore, when it terminates, $TreeSet_{min}$ is the same as $TreeSet_{min}(\phi)$ because by definition $TreeSet_{min}$ contains all the trees that satisfy ϕ with the minimal number of nodes. However, this algorithm is inefficient because it generates a huge number of trees which do not satisfy ϕ and have to be thrown away in later steps.[11]

[11] Recall that the number of possible rooted, ordered trees with n nodes is the $(n-1)^{th}$ Catalan Number, where the n^{th} Catalan Number b_n satisfies the following equation:

$$b_n = \frac{1}{n+1} \times \left(\begin{array}{c} 2n \\ n \end{array} \right) = \frac{4^n}{\sqrt{\pi} \times n^{3/2}} \times (1 + O(1/n)).$$

As the notion of tree in LexOrg is more complicated than the notion of rooted, ordered trees, the number of $TS(n)$ is much larger than b_{n-1}. Furthermore, most trees in $TS(n)$ do not satisfy ϕ, and therefore are not in $TreeSet_{min}(\phi)$.

Input: a description ϕ
Output: $TreeSet_m$ (i.e., $TreeSet_{min}(\phi)$)
Notation: $\lhd$ and $\lhd^*$ denote *parent* and *dominance* relations, respectively.
Algorithm: void GenTreesEff(ϕ, $TreeSet_m$)

// a description $\phi \Rightarrow$ a new description $\hat{\phi}$
(A) build a $\hat{\phi}$ which satisfies the following two conditions:
(1) $TreeSet(\phi) = TreeSet(\hat{\phi})$, and
(2) $\hat{\phi}$ is in the disjunctive normal form and does not use negation connectives;
that is, $\hat{\phi} = \hat{\phi_1} \vee ... \vee \hat{\phi_m}$, where $\hat{\phi_i} = \psi_{i_1} \wedge \psi_{i_2} ... \wedge \psi_{i_n}$ and ψ_{i_j} is a literal.

// a description $\hat{\phi} \Rightarrow$ a set of trees TC
(B) $TC = \{\}$;
(C) for (each $\hat{\phi_i}$)
// a description $\hat{\phi_i} \Rightarrow$ a graph G_i
(C1) draw a directed graph G_i. In G_i, there is a dashed edge (a solid edge, resp.)
from the node x to y *iff* one of the literals in $\hat{\phi_i}$ is $x \lhd^* y$ ($x \lhd y$, resp.).
(C2) store with the graph the *left-of* information that appears in $\hat{\phi_i}$.

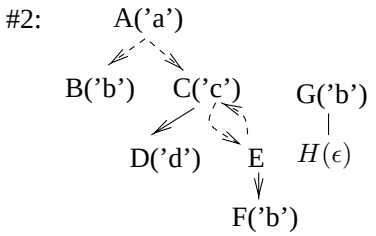
// a graph $G_i \Rightarrow$ a tree set TC_i
(C3) if (G_i has cycles)
then if (the set of nodes on each cycle are compatible)
then merge the nodes;
else $TC_i = \{\}$; continue;
(C4) merge the nodes in G_i until it does not have any compatible set;
(this step may produce more than one new graph)
(C5) for (each new G_i)
build a set of trees TC_i such that each tree
includes all the edges in G_i and
satisfies the *left-of* information;
$TC = TC \bigcup TC_i$;

// a set of trees $TC \Rightarrow$ a set of minimal trees $TreeSet_m$
(D) $a = min_{tr \in TC} NumNodes(tr)$;
(E) $TreeSet_m = \{tr \mid tr \in TC \text{ and } NumNodes(tr) = a\}$;

Table 1: A more efficient algorithm for building $TreeSet_{min}(\phi)$

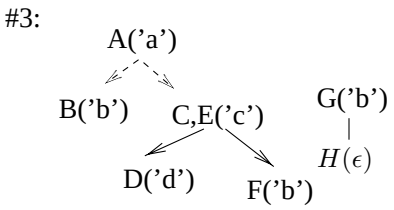
#1: $(A \lhd^* B) \wedge (A \lhd^* C) \wedge (C \lhd D) \wedge (C \lhd^* E)$
$\wedge(E \lhd^* C) \wedge (E \lhd F) \wedge (G \lhd H) \wedge (B \prec E)$
$\wedge(A.cat = \ 'a') \wedge (B.cat = \ 'b') \wedge (C.cat = \ 'c') \wedge (D.cat = \ 'd')$
$(F.cat = \ 'b') \wedge (G.cat = \ 'b') \wedge (H.cat = \ '\epsilon')$

(a) a description



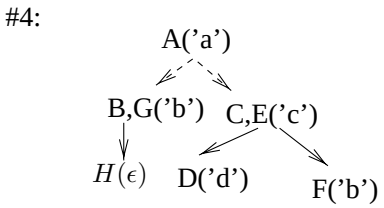
Left-of information: $B \prec E$

(b) a graph built from the description

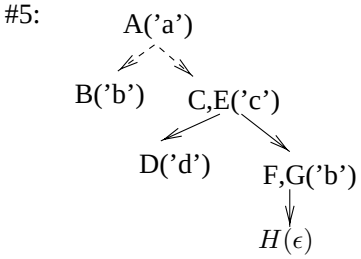


Left-of information: $B \prec C, E$

(c) the graph after cycles are removed

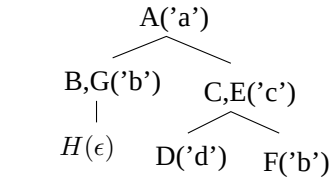


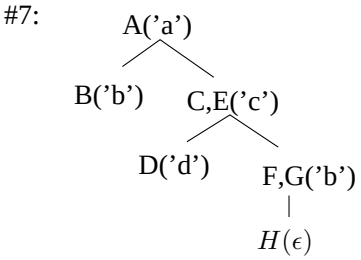
Left-of information: $B, G \prec C, E$



Left-of information: $B \prec C, E$

(d) the graphs after compatible sets are merged





(e) the trees built from the graphs

Figure 13: An example that illustrates how the new algorithm works: (a) is the original description in logical representation; (b) shows the graph built in Steps (C1) and (C2) in Table 1; (c) shows the graph after Step (C3) when cycles are removed; (d) shows two graphs produced in Step (C4), in which compatible sets are merged; and (e) shows the trees produced in Step (C5).

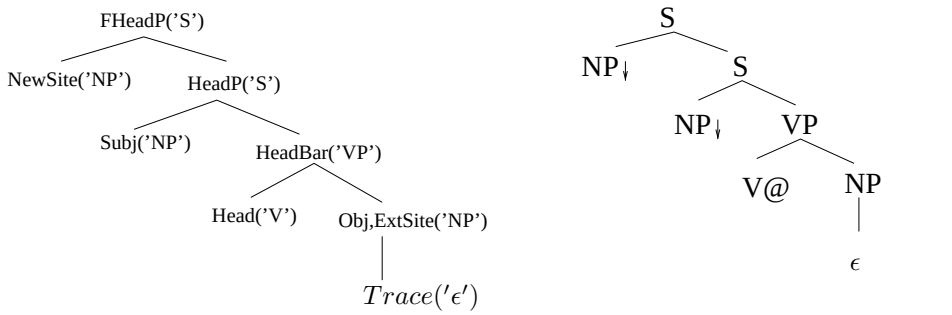
TreeGen uses a more efficient algorithm in which it first builds a new description $\hat{\phi}$ such that a tree satisfies $\hat{\phi}$ if and only it satisfies ϕ, and $\hat{\phi}$ is in disjunctive normal form $\hat{\phi_1} \vee ... \vee \hat{\phi_m}$, where each $\hat{\phi_i}$ uses only the conjunctive connectives. It does so by using rewrite rules that essentially capture the properties of trees to convert a negated formula into a disjunction of tree constraints, and then uses distributive rules to convert the formula into disjunctive normal form.[12] Second, for each $\hat{\phi_i}$ in $\hat{\phi}$, TreeGen builds a graph G_i. G_i is not necessarily a tree, as it might be disconnected, have loops, and so on. Third, TreeGen turns each G_i into a tree. There may be more than one possible tree for a graph; as a result, TreeGen gets a set of trees TC_i. Last, TreeGen chooses the subset of $\bigcup TC_i$ with the minimal number of nodes.

The new algorithm is in Table 1. The major steps of the algorithm are illustrated in Figure 13. The input description is in (a). Since the description is already in disjunctive normal form, TreeGen skips Step (A) in Table 1. In Steps (C1) and (C2), TreeGen creates a graphical representation for the description, as shown in Figure 13(b). A dashed edge (a solid edge, resp.) from the node x to y is in the graph if and only if $x \lhd^* y$ ($x \lhd y$, resp.) is one of the literals in the description. Steps (C3) – (C5) convert the graph into a tree. In (C3) TreeGen removes loops in the graph. If a loop contains only dashed edges, TreeGen removes the loop by merging all the nodes on the loop.[13] If a loop contains one or more solid edges, the nodes on the loop cannot be merged; that is, the description corresponding to the graph is inconsistent, and no templates will be created from this description. In this example, the nodes C and E are on a loop in graph #2 in Figure 13(b), and after merging, they become one node in the new graph, as shown in graph #3 in Figure 13(c). In Step (C4), TreeGen merges nodes that are compatible. A set of nodes are called *compatible* if the categories of the nodes in the set match and after merging the nodes there is at least one tree that can satisfy the new graph. In graph #3, the nodes G and B are compatible, so are G and F. Merging G and B results in graph #4 in (d), and merging G and F results in graph #5.[14] In Step (C5), for each graph produced

[12] In first order logic, two formulae are *equivalent* if any model that satisfies one formula also satisfies the other formula and vice versa. ϕ and $\hat{\phi}$ are not necessarily equivalent because we only require that the sets of *trees* (not *models*) that satisfy these two formulae are identical. Recall that trees are structures with special properties. For instance, given two symbols a and b in a tree, the formula $(a \prec b) \vee (b \prec a) \vee (a \lhd^* b) \vee (b \lhd a)$ is always true; therefore, a rewrite rule that replaces $\neg(a \prec b)$ with $(b \prec a) \vee (a \lhd^* b) \vee (b \lhd a)$ will not change the set of trees that satisfy a formula. The idea of using such rewrite rules originates from (Rogers and Vijay-Shanker, 1994). However, our goal of applying rewrite rules in this step is to get rid of negative connectives, rather than to find trees that satisfy each $\hat{\phi_i}$. Therefore, we use fewer numbers of rewrite rules and the $\hat{\phi_i}$ created by our algorithm can be inconsistent; that is, it is possible that no trees satisfy $\hat{\phi_i}$.

[13] When two nodes x and y are merged, in the graphic representation they become the same node after merging; in the logic representation, let ϕ be the description before the merging, after the merging the new description is $\phi \wedge (x = y)$.

[14] A node may appear in more than one compatible set. If a graph has two compatible sets, it is possible that after merging the nodes in one set, the other set is no longer compatible in the new graph. Therefore, if a graph has more than one compatible set, merging these sets in different orders may result in different graphs.



(a) a tree (b) the corresponding template

Figure 14: A tree and the template built from it

by Step (C4), TreeGen builds a set of trees that satisfy that graph. In this case, Step (C4) produces two graphs: #4 and #5. There is only one tree, #6, that satisfies graph #4, and one tree, #7, for graph #5. So the treeset TC after Step (C5) contains two trees: #6 and #7. Notice that without the *left-of* information, the node B in graph #4 could be C's sibling, parent, or child. But with the *left-of* information, B has to be C's left sibling. In Steps (D) and (E), TreeGen produces the final result $TreeSet_m$, which contains only the trees in TC that have the minimal number of nodes. In our example, the two trees in TC have the same number of nodes, so both are kept in the final result $TreeSet_m$.

4.3 Step 3: Building templates from the trees

In this step, TreeGen builds a unique template from each tree produced by the previous step. Recall that a node in a tree has the form $\{k_i\}(\{f_m = v_m\})$, where $\{k_i\}$ is a list of node names, and f_m is a feature and v_m is the feature value. In this step, LexOrg simply replaces $\{k_i\}(\{f_m = v_n\})$ with $l(\{f_m = v_m\})$, where l is the category of k_i (i.e., l is the value of $k_i.cat$). For a leaf node, if its type (i.e., anchor node, substitution node or foot node) is not specified by features, TreeGen determines its type by the following convention: if the leaf node is a head (an argument, a modifiee, respectively), it is marked as an anchor node (a substitution node, a foot node, respectively). Figure 8 (repeated as Figure 14) shows a tree and the template built from the tree.

5 The Description Selector

In the previous section, we showed that the Tree Generator builds templates from a set of descriptions. The set of descriptions used by the Tree Generator is only a subset of descriptions provided by the user. The

function of the second component of LexOrg, the *Description Selector*, is to choose the descriptions for the Tree Generator; to be more specific, it takes as input a subcategorization frame and the set of descriptions provided by the user, and produces sets of descriptions, which are then fed to the Tree Generator. This process, illustrated in Figure 15, is described below.

5.1 The definition of a *subcategorization frame*

A subcategorization frame specifies the categories of a head and its arguments, the positions of arguments with respect to the head, and other information such as feature equations. While our definition of a *subcategorization frame* is essentially the same as the one commonly used in the literature, we can also interpret a subcategorization frame as a subcategorization description.[15] For instance, the subcategorization frame $(NP_0\ V\ NP_1)$ can be seen as the shorthand version of the description

$$(leftarg \prec head) \wedge (head \prec rightarg) \wedge (leftarg.cat = \ 'NP') \wedge (head.cat = \ 'V')$$

$$\wedge (rightarg.cat = \ 'NP') \wedge (leftarg.subscript = 0) \wedge (rightarg.subscript = 1)$$

This interpretation allows LexOrg to treat a subcategorization frame the same way as other descriptions, as will be shown next.

5.2 The algorithm for the Description Selector

Recall that descriptions are divided into four classes: the ones for head-projection relations, head-argument relations, modification relations and syntactic variations. The first two classes (e.g., D_1, D_2 and D_3 in Figure 15) are also called *subcategorization* descriptions since they specify structures for a particular subcategorization frame. Because the templates in a tree family have the same subcategorization frame, the Description Selector should put in every description set SD_i all the subcategorization descriptions for that subcategorization frame. In addition to subcategorization information, in its choice of including other descriptions, the Description Selector's guiding principle is to capture the fact that elementary trees in an LTAG grammar reflect zero or more syntactic variations, and zero or one modification relations. Therefore, each description

[15] A subcategorization frame is different from other descriptions in that it cannot refer to any node other than the head and its arguments. For instance, it cannot refer to the *VP* which is the parent of the verb head. Another difference is that the categories of the nodes in a subcategorization frame must be specified. The reason for these differences is simply because we want to adopt the same definition of subcategorization frame as the one commonly used in the literature; namely, a subcategorization frame specifies the categories of the head and its arguments.

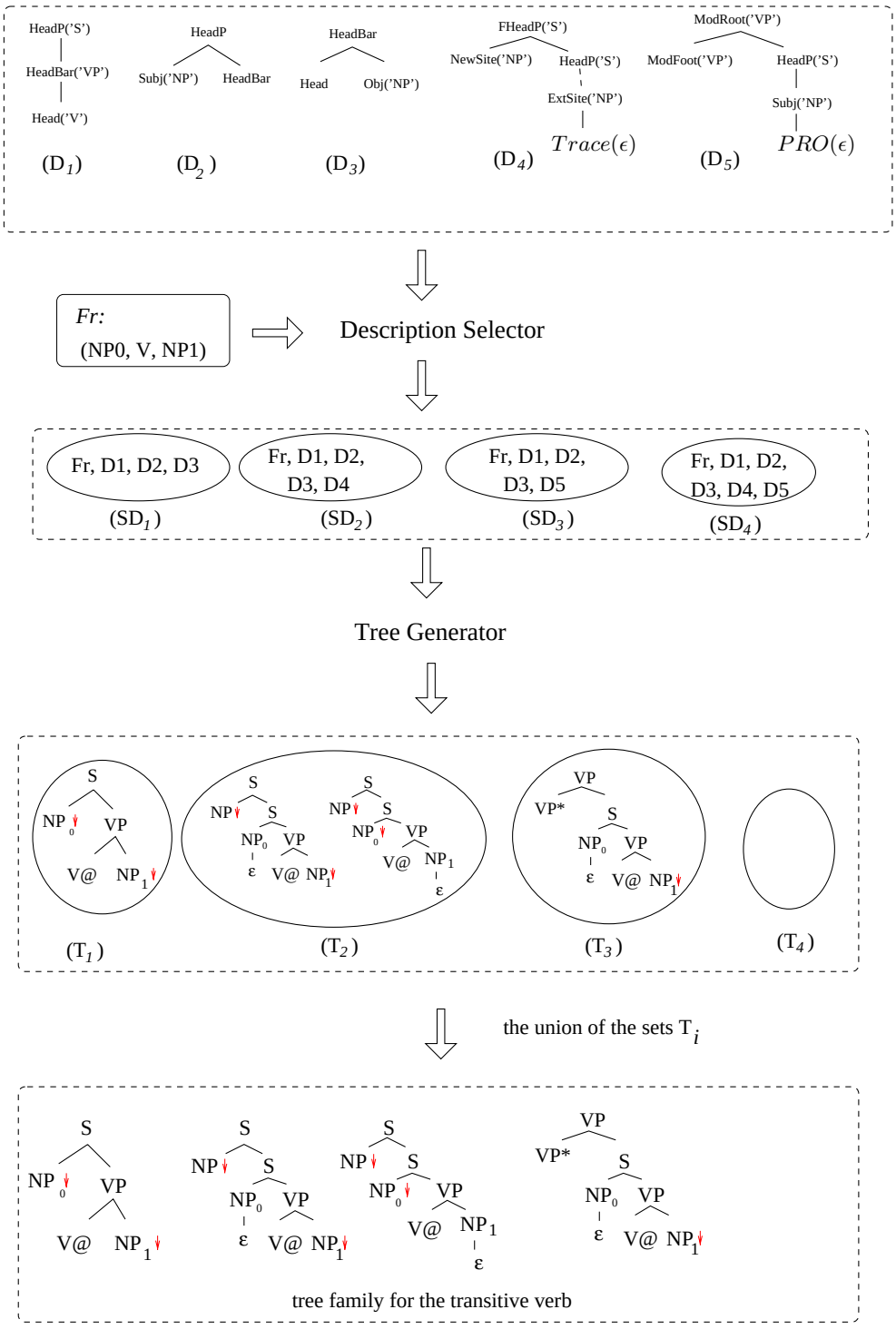


Figure 15: The function of the Description Selector

set built by the Description Selector should include all the related subcategorization descriptions, zero or more syntactic variation descriptions, and zero or one modification descriptions.

The algorithm is quite straightforward: given a subcategorization frame Fr, a set $Subcat$ of subcategorization descriptions, a set $Synvar$ of syntactic variation descriptions, and a set Mod of modification descriptions, the Description Selector's first responsibility is to select a subset $Subcat_1$ of $Subcat$ according to the arguments and category information mentioned in Fr. For instance, if Fr is $(NP_0\ V\ NP_1)$, $Subcat_1$ will include descriptions such as *head_is_V*, *head_has_a_left_NP_arg*, *head_has_a_right_NP_arg* and so on. As noted earlier, these descriptions need not be atomic and could be instantiations of more basic descriptions. Next, for each subset $Synvar'$ of $Synvar$ and each member m' of Mod, the Description Selector creates a set SD_i, which is $Subcat_1 \cup Synvar' \cup \{Fr\}$, and another set SD'_i, which is $SD_i \cup \{m'\}$.[16] This process is illustrated in Figure 15. In this example, $Subcat$ is $\{D_1, D_2, D_3\}$, $Synvar$ is $\{D_4\}$, and Mod is $\{D_5\}$. Given the subcategorization frame Fr, which is $(NP_0\ V\ NP_1)$, the Description Selector first chooses a subset $Subcat_1$ of $Subcat$, which happens to be the same as $Subcat$ in this case; it then creates multiple descriptions sets, each set including $Subcat_1$ and a subset of $Synvar$. Some description sets also include a member of Mod. As a result, the Description Selector produces four description sets for Fr: SD_1, SD_2, SD_3, and SD_4. Each SD_i is sent to the Tree Generator to generate a tree set T_i. Each T_i has zero or more trees. For instance, T_2 has two trees, whereas T_4 is empty because the descriptions in SD_4 (i.e., D_4 and D_5) are incompatible. The union of the T_is forms a tree family.

Notice the Description Selector considers different combinations of the descriptions that define the principles underlying the grammar design. The TreeGen produces the trees that are defined by the combinations of these principles when the combinations lead to consistent descriptions. Thus, these two components of LexOrg together take away from the LTAG grammar designer the burden of considering which set of principles are compatible with each other and which lead to inconsistencies. Thereby, the grammar designer can now focus on stating the individual linguistic principles, while the system automatically oversees the ramifications of these principles with respect to the details of the grammar.

[16] The number of description sets produced by the Description Selector is $2^{|Synvar|} \times (|\ Mod\ | +1)$. We can actually reduce this number by not producing some description sets that are obviously unproductive. A description set is *unproductive* if there exists no templates that satisfy all the descriptions in the set; as a result, the Tree Generator will produce nothing when it takes the set as the input. For instance, if in a head-projection description the head is a verb and its highest projection is a clause, the Description Selector will select a modification description only if the modifiee in that description is a clause.

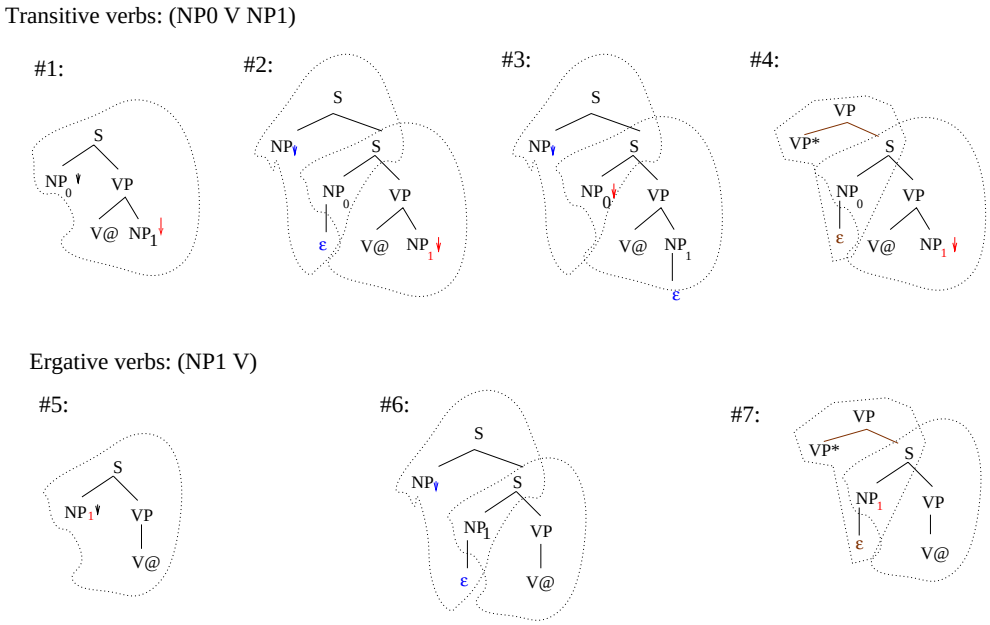


Figure 16: Templates in two tree families

(NP0 V NP1) => (NP1 V)

Figure 17: The lexical subcategorization rule for the causative/inchoative alternation

6 The Frame Generator

In an LTAG grammar, each word anchors one or more elementary trees. Figure 3 (repeated as Figure 16) shows seven templates anchored by ergative verbs such as *break*. The templates belong to two tree families because the subcategorization frames for them are different, but there is a clear connection between these two subcategorization frames, and all the ergative verbs (such as *break*, *sink*, and *melt*) have both frames. Levin (1993) listed several dozen alternations and classified English verbs according to alternations that they participate in. In LexOrg, we use lexical subcategorization rules to link related subcategorization frames.[17] Figure 17 shows the lexical subcategorization rule that links the two subcategorization frames in the causative/inchoative alternation. The function of the third component of LexOrg, the Frame Generator, is to apply lexical subcategorization rules to a subcategorization frame and generate all the related frames.

[17] In our previous papers on LexOrg, we called these rules *lexical rules*. However, the term *lexical rule* is heavily overloaded. For instance, lexical rules as defined in (Evans et al., 1995) can manipulate tree structures. They are used to account for wh-movement, topicalization, and so on. In contrast, the rules in LexOrg can manipulate only subcategorization frames. To avoid the confusion, in this paper we rename the rules in LexOrg *lexical subcategorization rules*, following a suggestion from one of the anonymous reviewers.

6.1 The definition of a *lexical subcategorization rule*

A lexical subcategorization rule is of the form $fr_1 \Rightarrow fr_2$, where fr_1 and fr_2 are just like subcategorization frames except that the categories of the nodes in fr_1 and fr_2 can be unspecified, in which case we will use a special label, *XP*, to represent an unspecified category. A lexical subcategorization rule $fr_1 \Rightarrow fr_2$ is said to be *applicable* to a subcategorization frame fr if fr and fr_1 are compatible; that is, fr and fr_1 have the same number of arguments and the features of the corresponding nodes can be unified.[18] Applying this rule to fr yields a new frame which combines the information in fr and fr_2. For instance, the lexical subcategorization rule $(XP\ V\ S) \Rightarrow (XP\ V\ NP)$ says that if a verb can take an S object, it can also take an NP object. Applying this rule to the frame $(NP_0\ V\ S_1)$ generates a new frame $(NP_0\ V\ NP)$. In this new frame, the category of the subject comes from the input frame, where the category of the object comes from the right frame of the lexical subcategorization rule. Because the category of the subject in the lexical subcategorization rule is not specified as indicated by the use of the label *XP*, the rule is also applicable to the frame $(S_0\ V\ S_1)$.

In addition to categories, the nodes in a lexical subcategorization rule may include other features. For instance, a lexical subcategorization rule for passivization is similar to the one in Figure 17 but the feature *voice* will have the value *'active'* for the verb in the left frame, and have the value *'passive'* for the same verb in the right frame. This feature will prevent the rule from being applied to a subcategorization frame in which the verb is already in the passive voice, such as *given* in *John is given a book*.

Lexical subcategorization rules and syntactic variation descriptions are very different in several aspects. First, a lexical subcategorization rule is a function that takes a subcategorization frame as input, and produces another frame as output; a syntactic variation description is a well-formed formula in a simplified first-order logic. Second, lexical subcategorization rules are more idiosyncratic than syntactic variations. For instance, the lexical subcategorization rule in Figure 17 is only applicable to ergative verbs, rather than to all the transitive verbs. In contrast, the description for wh-movement applies to all the verbs. Third, when lexical subcategorization rules are applied to a subcategorization frame in a series, the order of the rules

[18] In our current implementation, a lexical subcategorization rule $fr_1 \Rightarrow fr_2$ has to specify the numbers of arguments in fr_1 and fr_2. This requirement will be relaxed in the future to allow a more general version of the passive rule $(NP_0\ V\ NP_1\ XP^*) \Rightarrow (NP_1\ V\ XP^*)$, where * indicates that the argument XP is optional.

matters. In contrast, if a set of descriptions includes more than one syntactic variation description (e.g., the descriptions for topicalization and argument drop in Chinese), the order between the descriptions does not matter. Last, lexical subcategorization rules can be non-additive, allowing arguments to be removed; descriptions are strictly additive, meaning that a description can only add information and it cannot remove information. Notice that LexOrg does not place any constraint on which aspect of the grammar must be specified using lexical subcategorization rules or syntactic variation descriptions, and a grammar designer might even choose to use only one of these devices. However, because we believe that they can serve different purposes and we also like to provide flexibility to the grammar designer, both of these methods of grammar specification are available in LexOrg.

6.2 The algorithm for the Frame Generator

The Frame Generator takes a subcategorization frame Fr and a set of lexical subcategorization rules $Rules$ as input and produces as output a set $FrSet$ of related frames. The algorithm is in Table 2: first, Fr is the only member of $FrSet$; second, the Frame Generator applies each rule in $Rules$ to each frame in $FrSet$, and adds the resulting frames to $FrSet$.

```
Input: a subcategorization frame Fr and a set of lexical subcategorization rules Rules
Output: a list of related frames FrSet
Algorithm: void GenFrames(Fr, Rules, FrSet)

(A) let FrSet contain only the frame Fr
(B) for each frame f in FrSet
        for each lexical subcategorization rule r in Rules
            if r is applicable to f
              let f' be the new frame as r is applied to f
              if f' is not in FrSet
                 append f' to FrSet
```

Table 2: The algorithm for generating related subcategorization frames

In this process, the Frame Generator may first apply a rule r_1 to a frame f_1 which generates a new frame f_2; then it adds f_2 to $FrSet$, and apply r_2 to f_2 which generates f_3; and so on. When that happens, we say that a sequence $[r_1, r_2, ..., r_n]$ of lexical subcategorization rules is applied to the frame f_1. The order of the rules in such a sequence is important. For example, a passivization rule is applicable after the dative shift

rule is applied to the subcategorization frame for ditransitive verbs, but the dative shift rule is not applicable after a passivization rule is applied to the same frame. Rather than placing the burden of determining the order of applicability of the rules on the grammar designer, the system automatically tries all possible orders but will only succeed in producing the frames for ones with the correct ordering. Also, the set of possible sequences of lexical subcategorization rules is finite because the set of distinct lexical subcategorization rules is finite and in general each lexical subcategorization rule appears in a sequence at most once.[19] Therefore, the algorithm in Table 2 will always terminate.

7 The experiments

In previous sections, we have described the three components of LexOrg: the Tree Generator, the Description Selector, and the Frame Generator. To generate a grammar, the users of LexOrg need to provide three types of abstract specifications: subcategorization frames, lexical subcategorization rules, and tree descriptions. A natural question arises: *how does a user create such information*? To address this question and to test our implementation of LexOrg, we created two sets of abstract specifications: one for English, and the other for Chinese. From each set of the specifications, we used LexOrg to generate a grammar. We chose English because we wanted to compare our automatically generated grammar with the XTAG grammar, and we chose Chinese because one of the authors was very familiar with literature on Chinese linguistics which greatly facilitated the creation of the set of abstract specifications for Chinese. These languages also come from two very different language families, offering interesting points of comparison and a test of LexOrg's language independence. This section reports the results of our grammar production experiments.

7.1 Creating abstract specifications

Any large-scale grammar development requires a thorough study of various linguistic phenomena in the language to decide how these phenomena should be represented in the grammar, no matter whether or not tools such as LexOrg will be used. Once grammar designers have chosen the analyses, they can either create elementary trees by hand, or build abstract specifications and then use LexOrg to generate trees

[19] An arguable exception to this claim is the double causative construction in languages such as Hungarian (Shibatani, 1976). But in this construction it is not clear whether the second causativization is done in morphology or in syntax. Even if it is done at the morphological level, the two causativizations are not exactly the same and they will be represented as two distinct lexical subcategorization rules in LexOrg.

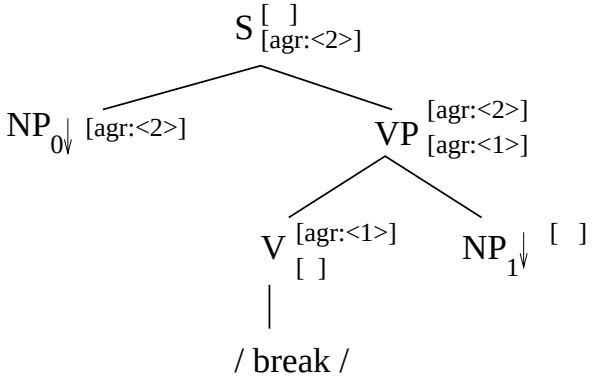


Figure 18: An elementary tree for the verb *break*

automatically. In Section 2.3, we have stated the advantages of using LexOrg for grammar development, one of which we emphasize here. LexOrg not only allows but actually requires grammar designers to state linguistic principles and generalization at the appropriate level; that is, Lexorg forces grammar designers to state the underlying linguistic principles explicitly. For instance, Figure 1 (repeated here as Figure 18) contains two feature equations, as indicated as the coindexes $< 1 >$ and $< 2 >$. The same equations appear in hundreds of tree templates in the XTAG grammar. If templates are created by hand, grammar designers have to consider for each template whether such equations should be included, and there is nothing to ensure that this process is done consistently. In contrast, if LexOrg is used to generate templates, grammar designers need to decide which abstract specifications such feature equations should belong to. Once the equations are added to appropriate specifications,[20] LexOrg will ensure that they are propagated to all relevant templates.

Now let us briefly discuss the abstract specifications that we created for English and Chinese. Only a limited number of categories (such as verbs and prepositions) take arguments and therefore have nontrivial subcategorization frames and lexical subcategorization rules. By *nontrivial*, we refer to subcategorization frames with at least one argument. Among these categories, verbs are the most complicated ones. To create subcategorization frames and lexical subcategorization rules for verbs, we studied the literature on verb classes such as (Levin, 1993) which discusses alternations and classifies verbs according to the alternations that the verbs can undergo. An alternation describes a change in the realization of the argument structure of a verb, and is illustrated by a pair of sentences in which a verb can appear. For instance, the spray/load alternation is illustrated by these two sentences "*Jack sprayed paint on the wall* " and "*Jack sprayed the*

[20] In the two sets of specifications that we created for English and Chinese, we added the feature equation $V.t :< agr >= VP.b :< agr >$ to the description in Figure 9(a), and the equation $VP.t :< agr >= NP_0.t :< agr >$ to the one in Figure 9(b).

wall with paint ". For each alternation, if all the dependents of the verb involved in the alternation are arguments of the verb, then each sentence in the sentence pair is abstracted into a subcategorization frame, and the alternation is represented as a lexical subcategorization rule. As the goal of the current experiment was to use LexOrg to create a grammar similar to the XTAG grammar, and the XTAG grammar has a very strict definition of arguments, only a few alternations (such as the causative alternation, the dative shift alternation, and the passive alternation) fall into this category and they are represented as lexical subcategorization rules.[21]

To create the first three classes of descriptions (namely, head-projection descriptions, head-argument descriptions, and modification descriptions), we adopt the following approach: in a head-projection description, the head and its projections form a chain, and the categories of the head and its projection are specified; in a head-argument description, the categories of the head and its argument are specified, as well as the positions of the arguments with respect to the head; in a modification description, the categories of the modifiee, the modifier and the head of the modifier are supplied, as well as the position of the modifier with respect to the modifiee.

To build a syntactic variation description, we started with the definition of the corresponding phenomenon. For example, *wh-movement* can be roughly defined as a phenomenon where *a constituent in a clause is moved from its base position to a new position.* Furthermore, the new position is to the left of the base position, the category of the parent of the new position is S, and the moved constituent includes a wh-word. From this definition, we created the description in Figure 11 (repeated as Figure 19).

7.2 Generating grammars

Once we created two sets of abstract specifications (one for English and the other for Chinese), we used LexOrg to generate grammars from these specifications. At that time, the XTAG grammar contained about one thousand elementary trees. Among them, about 700 trees were anchored by verbs. Because verbs have nontrivial subcategorization frames and lexical subcategorization rules, the goal of our experiment was to use LexOrg to "*reproduce*" this subset of trees with as little effort as possible. Given a pre-existing grammar

[21] All the other alternations contain some components that are considered to be adjuncts in the XTAG grammar. For instance, in the spray/load alternation, both the *PP* "*on the wall*" in the first sentence and the *PP* "*with paint*" in the second sentence are considered adjuncts in the XTAG grammar. As a result, no lexical subcategorization rule was created for this alternation, and the spray verbs are treated as normal transitive verbs.

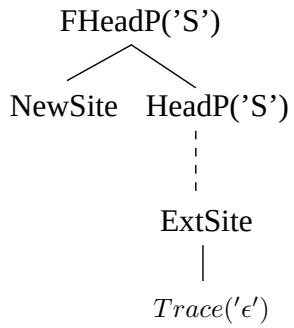


Figure 19: A description for wh-movement

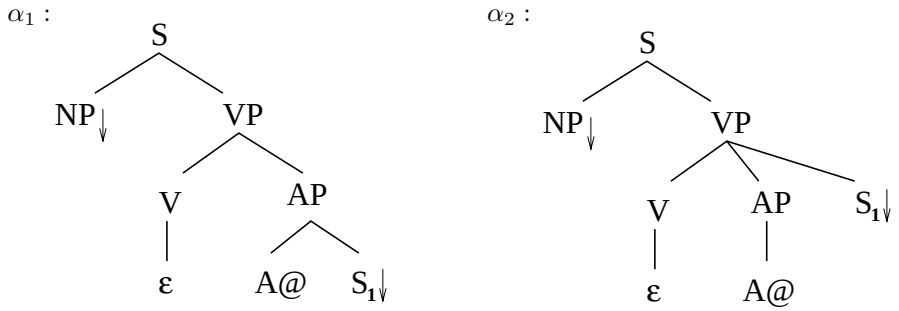


Figure 20: Two elementary trees for adjectives with sentential arguments

where the related linguistic phenomena had been well-studied, as in the English XTAG, creating a new version with LexOrg was quite straightforward, and required no more than a few weeks of effort. A tree-by-tree comparison of this new grammar and the original XTAG grammar allowed us to discover gaps in the XTAG grammar that needed to be investigated. The types of gaps included missing subcategorization frames that were created by LexOrg's Frame Generator and which would correspond to an entire tree family, a missing tree which would represent a particular type of syntactic variation for a subcategorization frame, or missing features in some elementary trees. Based on the results of this comparison, the English XTAG was extensively revised and extended.

The experiment also revealed that some elementary trees were easier to generate with LexOrg than other elementary trees. Figure 20 shows two elementary trees where an adjective such as *glad* takes a sentential argument. They differ in the positions of the S_1 node: in α_1 the S_1 node is a sister of the A node, but in α_2 it is a sister of the AP node. As both trees can handle a sentence such as *Mary was glad that John came to the party*, it is difficult to choose one tree over the other according to the set of sentences that each tree accepts. While it is equally easy to draw these two trees by hand, α_1 would be preferred over α_2 if LexOrg is used to

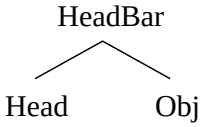


Figure 21: A head-argument description

generate a grammar. This is because the head-argument description in Figure 21, which is used to generate all the elementary trees anchored by transitive verbs or prepositions, can also be used to generate α_1. In contrast, the elementary tree α_2 would require a different head-argument description. Because our grammar includes the transitive verb family and one of the trees in Figure 20, choosing α_1 over α_2 will require a smaller set of descriptions. This example illustrates another advantage of using LexOrg besides the ease of creating and maintaining a grammar: the users of LexOrg are encouraged to create elegant, consistent, well-motivated grammars by defining structures that are shared across elementary trees and tree families.

In addition to English, we also used LexOrg to generate a medium-size grammar for Chinese. The Chinese grammar, although smaller than the English grammar, required several person-months, since many of the linguistic principles had to be defined along the way before the structures could be generated. Note that most of the time invested for the Chinese grammar was in linguistic analysis which would be applicable to any style of grammar, rather than in structure generation. In designing these two grammars, we have tried to specify grammars that reflect the similarities and the differences between the languages.

	English	Chinese
subcategorization frames	(NP, V, NP) (NP, V, NP, NP, S)	(NP, V, NP) (V)
lexical subcategorization rules	passive without by-phrase dative-shift	short *bei*-const *ba*-const
head-projection descriptions	S_has_V_head S_has_P_head	S_has_V_head
head-argument descriptions	V_has_NP_right_arg V_has_3_right_arg	V_has_NP_right_arg V_has_PP_left_arg
modification descriptions	NP_modify_NP_from_left S_modify_NP_from_right	NP_modify_NP_from_left S_modify_NP_from_left
syntactic variation descriptions	wh-question gerund etc	topicalization arg-drop etc.
# subcategorization frames	43	23
# lexical subcategorization rules	6	12
# descriptions	42	39
# templates	638	280

Table 3: Major features of English and Chinese grammars

To illustrate the similarities and differences between these two languages, for each language we give two examples for each type of abstract specification in Table 3: the first example has similar content in the two languages, while the second example appears in only one language. For example, the lexical subcategorization rule for passive without the by-phrase in English is very similar to the rule for the short bei-construction in Chinese, whereas the rule for dative-shift appears only in English, and the rule for the ba-construction appears only in Chinese. Similarly, both languages have wh-movement (topicalization in Chinese), but only English has a gerund form and only Chinese allows argument drop, as indicated by the row for syntactic variation descriptions. The bottom part of the table shows that with a small set of specifications, a fairly large number of templates were produced; and in the case of the English grammar, we were able to specify a grammar with a coverage comparable to that of the then current version of XTAG: LexOrg's English grammar covered more than 90% of the templates for verbs that were found in XTAG.[22] To maintain the grammars, only these specifications need to be modified, and all the elementary trees will be updated automatically.

We are encouraged by the utility of our tool and the ease with which both English and Chinese grammars were developed. We believe that, beginning with a pre-existing linguistic analysis and grammar design experience, a prototype grammar for a new language can be easily and rapidly developed in a few weeks. Furthermore, we see this approach as much more than just an engineering tool. Provably consistent abstract specifications for different languages offer unique opportunities to investigate how languages relate to themselves and to each other. For instance, the impact of a linguistic structure such as wh-movement can be traced from its specification to the descriptions that it combines with, to its actual realization in trees.

7.3 Handling free word order languages

We have just reported on our experience in creating abstract specifications for English and Chinese, and using LexOrg to generate grammars from the specifications. Because neither of these languages is a *free word order language*, which has received much attention in the literature, let us now briefly discuss how LexOrg could be used to handle free word order languages.[23]

[22] The remaining 10% of the templates are like α_2 in Figure 20 in that they require some abstract specifications which do not quite fit with the rest of the grammar. For example, as explained before, α_2 in Figure 20 would require a head-argument description which is very different from the one used for transitive verbs or prepositions. In order to keep our set of specifications for English elegant and well-motivated, we did not include such specification, although adding such specification will guarantee that the resulting new English grammar would cover all the templates for verbs that were found in XTAG.

[23] We would like to thank an anonymous reviewer for suggesting this topic.

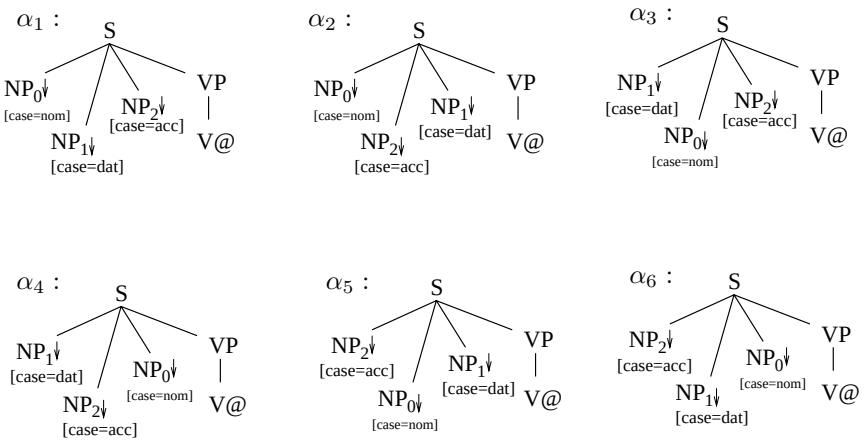
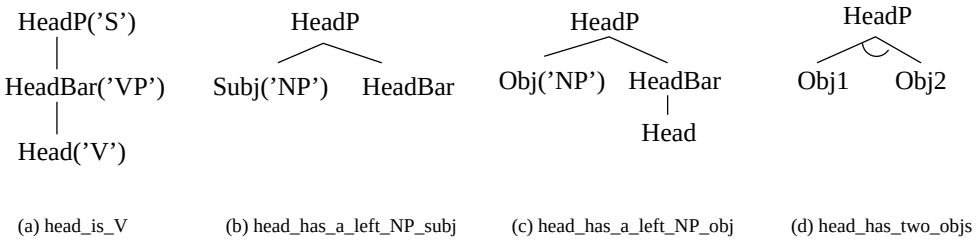


Figure 22: A set of templates for dative verbs in a free word order language

In a free word order language, words can appear in different orders under certain constraints. For instance, a ditransitive verb (such as *give* in English) has three arguments — let us call them NP_0 (the agent), NP_1 (the beneficiary), NP_2 (the theme) — and there are six possible orderings of these three arguments. In English, without the preposition *to* or a comma as in a topicalized sentence, only one of these six orderings, "NP_0 V NP_1 NP_2", is grammatical. In contrast, in a free word order language such as Korean or German, all six orderings are allowed because case markers make it clear which NP is the agent, the beneficiary, or the theme. Furthermore, one or more constituents in an embedded clause in a free word order language can move from that clause to the matrix clause and such movement can be unbounded, resulting in the so-called *long-distance scrambling* phenomenon.

There has been much discussion on how to handle scrambling in the LTAG framework. For instance, Becker (Becker, 1994) proposes two extensions of the basic LTAG framework, namely Free Order TAG (FO-TAG) and Multi-component TAG (MC-TAG), to handle scrambling. A detailed discussion of work on free word order languages and scrambling is beyond the scope of this paper; however, we want to emphasize a point that we made before: whether a grammar is created by hand or by LexOrg, a thorough study of various linguistic phenomena in the language is inevitable in order to decide how these phenomena should be represented in the grammar; once the grammar designer has chosen linguistic analyses, in general it is quite obvious what type of abstract specifications should be used to generate the corresponding elementary trees, and LexOrg facilitates the creation and maintenance of the resulting grammars.

In this case, for example, a grammar designer may decide to use the templates in Figure 22 to handle all



(a) head_is_V (b) head_has_a_left_NP_subj (c) head_has_a_left_NP_obj (d) head_has_two_objs

Figure 23: Descriptions for the templates in Figure 22

six possible orderings of the three arguments of a dative verb in Korean. Note that the three arguments in the templates are distinguished by both the subscripts and the values of the feature *case*: *nom* for nominative, *dat* for dative, and *acc* for accusative. To generate these templates using LexOrg, one only needs to create the descriptions in Figure 23 and a subcategorization frame (NP_0 NP_1 NP_2 V): the arc in the last description means that the precedence relation between $Obj1$ and $Obj2$ is unspecified.[24] Given this subcategorization frame, LexOrg will select all four descriptions in Figure 23 — with the third description selected twice because a dative verb has two objects. Since the precedence relation between subject and two objects is not specified in these descriptions, all 3! permutations of the arguments are possible and the descriptions will yield all the six templates in Figure 22.

It is common for grammar designers to change analyses while developing a large-scale grammar. In this case, one might later decide to use the analysis in which the two NP objects are base generated as children of the VP, and then moved up and become children of the S. As a result of the new analysis, the template α_1 should be changed to the template in Figure 24, and the other five templates need to be modified in a similar way. When LexOrg is used, instead of changing the six templates manually, one only needs to modify the last two descriptions in Figure 23 to the ones in Figure 25, and LexOrg will ensure that the changes are propagated to all the related templates. In summary, the scrambling phenomenon in a free word order language requires thorough linguistic study; once a grammar designer chooses an appropriate analysis, LexOrg will greatly reduce the work of creating and maintaining the templates.

[24]The users of LexOrg should decide whether the values of the feature *case* for the three arguments in Figure 22 (e.g., "case=nom") should come from the subcategorization frame for the verb that anchors these templates or from the descriptions in Figure 23. Therefore, we do not mark these features in Figure 23. Also, as stated before, LexOrg allows languages to be compared at the specification level, rather than at the elementary tree level. In this case, comparing the descriptions in Figure 23 for free word order languages with the descriptions in Figure 9 for English will reveal some differences between these languages. First, the positions of the object are different (see (c) in both figures). Second, the precedence relation of two objects is not specified in the description for free word order languages (as indicated by the arc in Figure 23), but it is specified for English (the description is not shown in Figure 9, but it is identical to Figure 23(d), except that it does not have the arc between two objects).

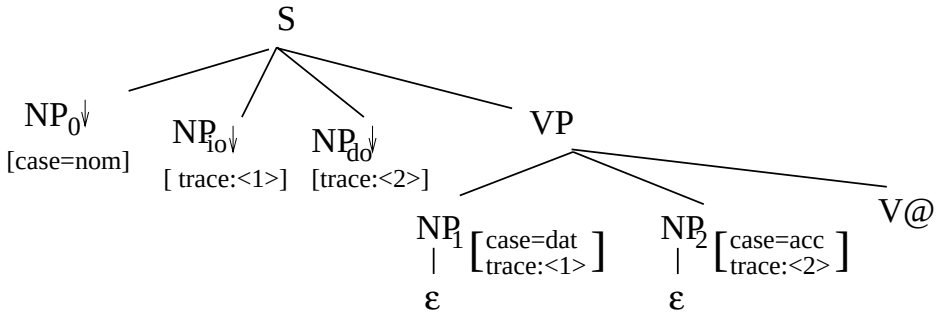


Figure 24: A different analysis for dative verbs

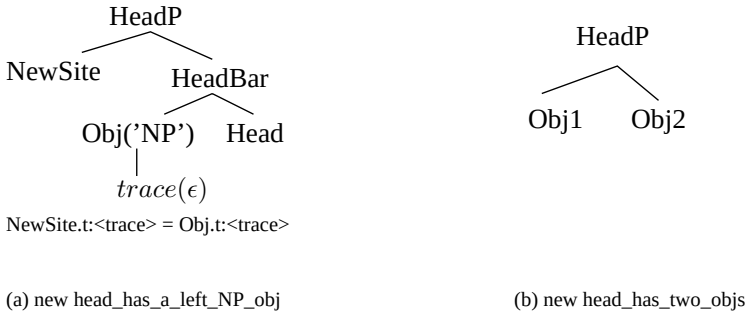


Figure 25: Modified descriptions for the templates in Figure 24

8 Comparison with other work

It has been long observed that the templates in an LTAG grammar are related to one another and could be organized in a compact way for efficient development and maintenance. In this section we first explain why we use tree descriptions in LexOrg, instead of (typed) feature structures. Then we compare LexOrg with three alternative approaches to LTAG organization: Becker's HyTAG system (Becker, 1994); a system by Evans, Gazdar and Weir (Evans et al., 1995) implemented in DATR (Evans and Gazdar, 1989); and Candito's system (Candito, 1996).[25] These three systems and LexOrg differ in how they handle two basic relations: the grouping into tree families of templates that share tree structures which describe the same subcategorization frame; and the organization of the tree families themselves into hierarchies, either manually or automatically. The differences with respect to the first relation will be discussed later. Here, we shall point out the differences with respect to the second relation.

In a lexical hierarchy, a class inherits attributes from its superclasses as illustrated by Figure 26. (For a detailed example of a verb subcategorization frame hierarchy adhering to strict inheritance properties, see (Copestake and Sanfilippo, 1993; Briscoe et al., 1994).) Although the hierarchy seems intuitive, building

[25] For more details of these systems and the comparisons, see Chapter 4 of (Xia, 2001).

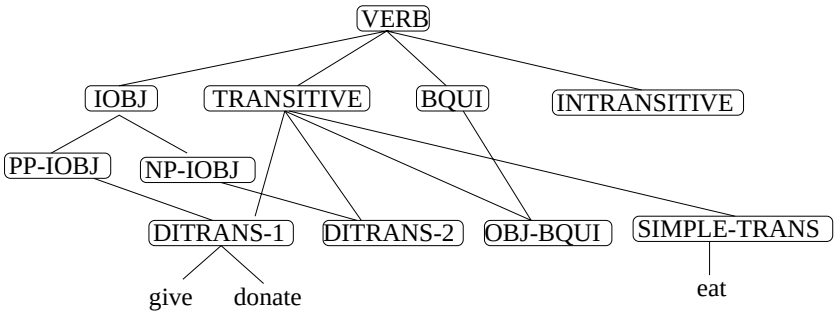


Figure 26: The lexical hierarchy given in (Vijay-Shanker & Schabes, 1992)

it manually is by no means a trivial task. Grammar designers have to answer questions such as *should the hierarchy be a tree or a network? If it is a network, how should the conflicts between multiple superclasses be resolved? Can a subclass overwrite attributes that are inherited from its superclasses? If some verb classes share certain structures, is it necessary to create an abstract superclass for them — such as the node VERB in Figure 26? What information should be included in the definition of what class?* Answers to these questions may vary, resulting in different hierarchies.

LexOrg differs from all the other approaches in that LexOrg does not need to use a pre-defined hierarchy at all. The inheritance relation between tree families is implicit. For instance, the description set selected by LexOrg (as in Section 5) for the ditransitive verb family is a superset of the descriptions selected for the transitive verb family. Therefore, the ditransitive family implicitly "inherits" all the information from the transitive family without referring directly to an explicit hierarchy or the transitive family at all. If one wishes to make explicit this implicit inheritance hierarchy, he can simply build the hierarchy by adding an inheritance link between every tree family pair that satisfies the following condition: the subcategorization description set selected for one family is a superset of the subcategorization description set selected for the other family.

8.1 Tree description vs. Feature structure

The focus in this paper on the use of tree descriptions in LexOrg is directly related to the fact that grammarians using LTAG focus more on tree structures than on the feature structures associated with nodes of these trees. LexOrg's use of tree descriptions rather than feature description logic reflects this bias.

We stated above that with LTAG, like CFG, there is more of a separation between the formalism and the

linguistic theories used to instantiate grammars. Yet, there is one aspect that is almost universally adopted in the design of LTAG grammars for natural languages. It concerns the localization of dependencies within the elementary trees of LTAG. This means that nodes that can be correlated with respect to some form of dependency (e.g., a head and its arguments, or even the so-called long-distance dependency between the filler and the gap) co-occur within the same elementary tree. We also stated that LTAG is viewed as a tree-rewriting system, with much of the focus of grammar design on capturing the phrase structure trees (with the feature structures associated with the nodes almost treated as syntactic sugar). Hence much of the grammar design focuses on the specification of the tree structure relations between nodes that co-occur because of dependencies. Furthermore, in cases of long-distance dependencies, the relation between the filler and the gap nodes is often one of c-command, a relation which itself is normally defined in terms of unbounded dominance. As shown in Figure 11, the structural description for such a long-distance dependency (but localized in LTAG) is stated succinctly using unbounded dominance. In addition to this relation (between nodes of trees), immediate-dominance (parent-child relation) and precedence (left-of) relations are also desirable for describing the phrase structure representation of other forms of dependencies (such as the structural relationship between a head and its arguments).

The above discussion motivates our choice of using these three primitives (dominance, immediate dominance, precedence) for describing the different aspects of elementary trees in an LTAG. When such descriptions are used, clearly a system like LexOrg has to confirm whether a given set of descriptions is consistent, and if so it should obtain the elementary trees of the LTAG being described as minimal models of such satisfiable descriptions. We believe that description languages developed for (typed) feature structures are not immediately suitable for our purpose. As the underlying logic behind these grammar development systems take the models to be essentially directed acyclic graphs, the logics would need to be extended to capture those constraints that define when a directed acyclic graph is also a tree. Thus, for example, we would need to in some form enforce that the dominance relation is the reflexive, transitive closure of the immediate dominance relation, that dominance and precedence relations are mutually exclusive, and that the precedence relation inherits through the dominance relation, and so on. Such properties are needed to verify satisfiability and for deriving the minimal models. The use of transitive closure of attribute paths have been

proposed in the context of *functional uncertainty* (Kaplan and Zaenen, 1988) and specialized algorithms to deal with the use of regular expression in paths have been developed by (Kaplan and Maxwell, III, 1988; Backofen, 1996). However, the addition of negation makes the satisfiability-checking problem undecidable (Baader et al., 1993).

In LexOrg, we keep separate the constraint solving for tree descriptions and for attribute value structures.[26] For the former, LexOrg borrows from the quantifier-free language for tree descriptions introduced in (Rogers and Vijay-Shanker, 1994). Much of the original motivation for the development of this language derived from LTAG considerations and for describing LTAG elementary trees and the adjoining operation as discussed in (Vijay-Shanker, 1992; Vijay-Shanker and Schabes, 1992). We not only base our work on this language but also use the associated machinery for satisfiability checking and the formulation of minimal models developed in (Rogers and Vijay-Shanker, 1994). Thus, the description language and the associated constraint solving mechanisms already exist and we adapt them here for our purpose. For the latter (i.e., the constraint solving for attribute value structures), given the localization of dependencies in the elementary trees (Kroch, 1987), features such as *SLASH* are unnecessary in LTAG,[27] and with LTAG, one doesn't need to use more than flat attribute value pairs. We believe that the use of full scale typed feature structures is unnecessary given the extent to which feature structure constraints are used in LTAG.

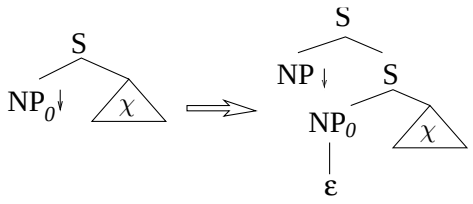
8.2 Becker's HyTAG

A *metarule* in general consists of an input pattern and an output pattern. When the input pattern matches an elementary structure in a grammar, the application of the metarule to the structure creates a new elementary structure. Metarules were first introduced in Generalized Phrase Structure Grammar (GPSG) (Gazdar et al., 1985). Later, Becker modified the definition of metarules in order to use them for LTAG in his HyTAG system (Becker, 1994). In addition to metarules, Becker's HyTAG system also uses an handcrafted inheritance hierarchy such as the one just discussed.

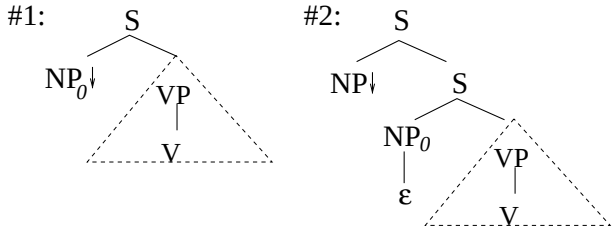
In HyTAG, the input-pattern and the output-pattern of a metarule are elementary trees with the exception that any node may be a meta-variable. A *meta-variable* describes part of a template that is not

[26] In one sense, a common aspect in LTAG grammars developed so far is that the association of feature structures with nodes of the trees are like the mapping between the c-structure nodes and f-structures in LFG and the two types of structures are not represented with an uniform representation.

[27] One of the original motivations for the work on compiling HPSG into a LTAG was to investigate the extent to which any of such features with unboundedly large feature structures as values are necessary after the LTAG is obtained.



(a) a metarule



(b) two templates for intransitive verbs

Figure 27: Applying metarules to templates

affected if the metarule is applied. If a template matches the input-pattern, the application of the metarule creates a new template which could be added to the grammar. In Figure 27, (a) shows a metarule that links a declarative template and a wh-movement template, where χ is a meta-variable. Applying this metarule to the template #1 in (b) results in the template #2, as the meta-variable χ matches the whole *VP* in #1. More specifically, metarules can

- erase structures from the input template.
- specify dominance relations between nodes (but unlike LexOrg, precedence relations in metarules cannot be left unspecified and disjunction and negation cannot be expressed.)
- produce more than one template.
- be applied to a template in a series. Without any constraints, a metarule sequence can be infinitely long, and the application process may never terminate unless additional restrictions are applied.[28]

[28] One such restriction is called the *finite closure* constraint, which requires a metarule to appear in a sequence at most once. Becker proposed a different restriction which requires that the output trees of metarules must be smaller than a given limit M of a grammar. However, he did not define what he meant by *smaller*, nor did he elaborate on where the M comes from. Furthermore, having a limit on the size of output trees can guarantee that the application process will terminate, but it may not guarantee that all the output trees that are smaller than the limit are linguistically sound.

LexOrg Comparison A major difference between HyTAG and LexOrg is that HyTAG uses metarules to describe both lexical and syntactic rules, whereas LexOrg uses two mechanisms: lexical subcategorization rules and descriptions. Aside from the linguistic debate that argues for different treatments of lexical and syntactic rules, using different mechanisms results in LexOrg having a small number of lexical subcategorization rules which are simpler than metarules because they do not contain meta-variables. This makes it easier to ensure the termination of the application process. It also allows for more modular encoding of constraints such as feature agreements.

Another difference between HyTAG and LexOrg is the way that templates are related. To relate the templates in the same tree family, the users of HyTAG first build one template as the basic tree, and then create metarules that link the basic tree and other trees in the family. The input and the output patterns have to specify exactly what information is in one tree but not in the other. In contrast, the users of LexOrg provide a subcategorization frame and a set of descriptions; the rest is done automatically. The trees are related implicitly by the descriptions that they share, rather than by rules that link them explicitly.

Explicitly specifying input and output patterns is also necessary for metarules that link two related tree families. For instance, when relating a transitive tree with its corresponding ergative tree in HyTAG, the agreement features between the surface subject and the VP have to be explicitly detailed on both sides of the tree, since on one side the surface subject is NP_0 and on the other it is NP_1. In contrast, to relate templates in two tree families, LexOrg uses a simpler lexical subcategorization rule to link the two subcategorization frames, and a description that includes the feature agreement will automatically be selected.

8.3 The DATR system

Evans, Gazdar and Weir (Evans et al., 1995) discuss a method for organizing the trees in a TAG hierarchically, using an existing lexical knowledge representation language called DATR (Evans and Gazdar, 1989). In the DATR system, an elementary tree is described from its lexical anchor upwards as a feature structure using three tree relations: the left, right, and parent relations. Like HyTAG, the DATR system uses an inheritance hierarchy to relate verb classes. For instance, the *VERB+NP* class inherits the structure from the *VERB* class and adds a right *NP* complement as the sister of the anchor. The system uses lexical rules to capture the relationships between elementary trees. A lexical rule defines a derived output tree structure

in terms of an input tree structure. Since the lexical rules in this system relate elementary trees rather than subcategorization frames, they are more similar to metarules in HyTAG than to lexical subcategorization rules in LexOrg.

In addition to topicalization and wh-movement, lexical rules in the DATR system are also used for passive, dative-shift, subject-auxiliary inversion, and relative clauses.[29] In the passive rules, instead of stating that the first object of the input tree is the subject of the output tree, the lexical rule simply discards the object. As a result, the relationship between the object in an active sentence and the subject in the corresponding passivized sentence is lost.

LexOrg Comparison Similarly to HyTAG, the DATR system requires a hand-crafted hierarchy and does not distinguish between syntactic rules and lexical rules, in contrast with LexOrg which can generate its hierarchy automatically and which clearly separates syntactic rules and lexical subcategorization rules. There are two other major differences: (1) the descriptions used by LexOrg are constrained to be strictly monotonic, whereas the DATR system allows non-monotonicity in its application of rules; (2) the DATR system can only directly capture relations between nodes in a tree (such as the parent-child relationship or precedence), and must use feature-equations to simulate other tree relations. This means that an abstract concept like dominance can only be specified by spelling out explicitly all of the different possible path lengths for every possible dominance relationship. For instance, in wh-movement, the moved *NP* is dominated by the *S* in the input tree. The *NP* can be the subject, the object of the verb, or the object of a *PP* where the *PP* is an object of the verb. In the DATR system, three distinct lexical rules are needed to specify these three possibilities. In contrast, since descriptions used by LexOrg are more expressive, only one description is needed to cover all three cases.

8.4 Candito's system

Like LexOrg, Candito's system (Candito, 1996) is built upon the basic ideas expressed in (Vijay-Shanker and Schabes, 1992) for the use of descriptions to encode tree structures shared by several elementary trees. Her system uses a hand-written hierarchy which has three dimensions. In the first dimension, canonical

[29] In LexOrg, passive and dative-shift are handled by lexical subcategorization rules. A parse tree for a subject-auxiliary inversion sentence is created by adjoining an auxiliary tree (anchored by an auxiliary verb) to an elementary tree (anchored by the main verb). LexOrg uses descriptions to express the information in topicalization, wh-movement, and a relative clause.

subcategorization frames are put into a hierarchy similar to the ones in HyTAG and the DATR system. The second dimension includes all possible redistributions of syntactic functions. The third dimension lists syntactic realizations of the functions. It expresses the way that the different syntactic functions are positioned at the phrase-structure level. The definitions of classes in these dimensions include descriptions and meta-equations.

A *terminal* class is formed in two steps. First, it inherits a canonical subcategorization from dimension 1 and a compatible redistribution from dimension 2. This pair of super-classes defines an actual subcategorization frame. Second, the terminal class inherits exactly one type of realization for each function of the actual subcategorization from dimension 3. A terminal class is actually a description. Elementary trees are the minimal trees that satisfy the description. For instance, a terminal class inherits the ditransitive frame (NP_0 V NP_1 NP_2) from dimension 1 and the passive redistribution from dimension 2; this yields the actual subcategorization frame (NP_1 V NP_2). It then inherits *subject-in-wh-question* and *object-in-canonical-position* realizations from dimension 3. The resulting elementary tree is anchored by a passivized ditransitive verb whose surface subject (i.e., the indirect object in the active voice) undergoes wh-movement, such as *given* in *who was given a book?*

A terminal class inherits one class from dimension 1, one from dimension 2, and one or more from dimension 3. These superclasses may be incompatible. For instance, in English, only one argument of a verb can undergo wh-movement; therefore, the classes *subj-in-wh-question* and *obj-in-wh-question* from dimension 3 are incompatible. To ensure that all the superclasses of a terminal class are compatible, the system provides several ways for its users to express compatibility constraints.[30] The users can mark a class C in the hierarchy as a *disjunctive* node, meaning that a terminal class cannot inherit more than one subclass of C. The user can also specify positive or negative inheritance constraints. A positive constraint (A, B) requires that any class that inherits from A must also inherit from B. A negative constraint (A, B) requires that any class that inherits from A cannot inherit from B. Another type of constraint is called *constrained crossing*. A constrained crossing is a tuple (A, B, C), meaning whenever a class inherits from A and B, it has to also inherit from C.

[30]The content of this paragraph comes from the handout of the talk that Candito gave at the University of Pennsylvania in 1997.

LexOrg Comparison There are many similarities between these two systems as both use descriptions to encode tree structures shared by several elementary trees, and there is a separation of lexical rules and syntactic rules. There is an obvious parallel between Candito's subcategorization dimension and our subcategorization descriptions, between her redistribution dimension and our lexical subcategorization rules, and between her realization dimension and our syntactic variation/modification descriptions. However, there are also several major differences.

The first difference is that Candito's system requires a hand-written hierarchy whereas LexOrg does not. Candito's system also requires that each terminal class should select exactly one class from dimension 2. This means that if two lexical subcategorization rules can be applied in a series (such as passive and causative) to a subcategorization frame, a node that represents that sequence must be manually created and added to dimension 2. In other words, dimension 2 should have a node for every rule sequence that is applicable to some subcategorization frame. LexOrg does not need users to build this dimension manually because the Frame Generator in LexOrg automatically tries all the rule sequences when given a subcategorization frame.

The two systems also differ in the way that syntactic variations are represented. In Candito's third dimension, each argument/function in a subcategorization frame requires an explicit representation for each possible syntactic realization. For example, the subject of a ditransitive verb has a different representation for the canonical position, for wh-extraction, and so on. So do the direct object and indirect object. To generate templates for wh-questions of ditransitive verbs, Candito's system needs to build three separate terminal classes. In contrast, LexOrg does not need descriptions for the various positions that each argument/function can be in. To generate the template for wh-questions, LexOrg only needs one wh-movement description. Combining this description with the set of subcategorization descriptions will yield all the templates for wh-questions.

Another difference between the two systems is that Candito's system requires its users to specify constraints on the selection of superclasses. For instance, the user has to build a class for wh-movement and a subclass for each position from which a constituent can be moved (e.g., *subj-in-wh-position*, *indobj-in-wh-question*, and *dobj-in-wh-question*). The users then have to mark the extraction class as a disjunctive node, so that the system will not choose more than one of its children for a terminal class. In LexOrg, only one

description for wh-movement is needed, which covers all possible cases. There is no need to write constraints to rule out illegal combinations.

9 Summary

In LTAG there is a clear distinction made between a grammar and the grammatical principles that go into developing this grammar. Arguments have been made on linguistic and computational grounds that the use of a suitably enlarged domain of locality provided by the elementary trees and the operations of substitutions and adjoining provide many advantages. But it is clear that these elementary trees, especially given that they have an enlarged domain of locality, are themselves not atomic but rather encapsulate several individual independent grammatical principles. Although this fact is widely understood in the LTAG context, most of the large scale grammar development efforts have directly produced the elementary trees, thereby in essence manually compiling out subsets of independent principles into elementary trees. Of course, with such manual efforts, the larger the grammar, the more prone to errors it becomes and the harder it is to maintain. The emphasis on tree structure rather than typed feature structures in LTAG precludes the adaptation of generalization techniques incorporated into HPSG, as discussed in detail in Section 8.1.

LexOrg is a computational tool that alleviates such problems in grammar design for LTAGs. It takes three types of abstract specifications (i.e., subcategorization frames, lexical subcategorization rules, and descriptions) as input and produces LTAG grammars as output. Descriptions are further divided into four classes according to the information that they provide. In grammar development and maintenance, only the abstract specifications need to be edited, and any changes or corrections will automatically be proliferated throughout the grammar.

LexOrg has several advantages over other similar semi-automated approaches to LTAG grammar development and maintenance that are also based on a more abstract grammar perspective, as discussed in detail in Section 8. First, unlike HyTAG, the DATR system and Candito's system, not only does LexOrg not require its users to construct a hand-crafted lexical hierarchy, but it can actually produce a hierarchy automatically by checking the descriptions selected by subcategorization frames. Second, unlike HyTAG and the DATR system, LexOrg distinguishes lexical subcategorization rules from syntactic rules and uses

two different mechanisms to represent these rules resulting in simpler rules that are easier to encode. Third, unlike Candito's system, LexOrg does not require users to provide various kinds of constraints manually to ensure that a terminal class inherits the correct combinations of superclasses from three dimensions. LexOrg automatically detects illegal combinations. Finally, unlike the DATR system and Candito's system, LexOrg needs only one description, which is generally applicable, to specify the information for wh-movement.

Given a pre-existing linguistic analysis, a new grammar can be developed with LexOrg in a few weeks, and easily maintained and revised. This provides valuable time savings to grammar designers, but, perhaps even more importantly, the reuse of descriptions encourages a comprehensive and holistic perspective on the grammar development process that highlights linguistic generalizations. The users of LexOrg are encouraged to create elegant, consistent, well-motivated grammars by defining structures that are shared across elementary trees and tree families.

In addition to greatly shortening grammar development time and lightening the more tedious aspects of grammar maintenance, this approach also allows a unique perspective on the general characteristics of a language. The abstract level of representation for the grammar both necessitates and facilitates an examination of the linguistic analyses. The more clearly the grammar designer understands the underlying linguistic generalizations of the language, the simpler it will be to generate a grammar using LexOrg. In using LexOrg to create an English LTAG, we demonstrated that this process is very useful for gaining an overview of the theory that is being implemented and exposing gaps that remain unmotivated and need to be investigated. The type of gaps that can be exposed include a missing subcategorization frame that might arise from the automatic combination of subcategorization descriptions and which would correspond to an entire tree family, a missing tree which would represent a particular type of syntactic variation for a subcategorization frame, and trees with inconsistent feature equations. The comparison of the LexOrg English grammar with the pre-existing XTAG grammar led to extensive revisions of XTAG, resulting in a more elegant and more comprehensive grammar. Provably consistent abstract specifications for different languages offer unique opportunities to investigate how languages relate to themselves and to each other. For instance, the impact of a linguistic structure such as wh-movement can be traced from its specification to the descriptions that it combines with, to its actual realization in trees. By focusing on syntactic properties

at a higher level, our approach allowed a unique comparison of our English and Chinese grammars.

10 Acknowledgment

Joseph Rosenzweig is acknowledged for his original implementation of tree descriptions in Prolog which demonstrated the feasibility of this endeavor. Aravind Joshi has provided continued guidance and support and Marie Candito participated in several lengthy discussions with the authors during her visit to the University of Pennsylvania. This work has been supported by DARPA N66001-00-1-8915, DOD MDA904-97-C-0307, NSF SBR-89-20230-15 and NSF 9800658.

References

Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka. 1993. On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations. *Journal of Logic, Language and Information*, 2:1–18.

Rolf Backofen. 1996. Controlling Functional Uncertainty. In *Proc. of the European Conference on Artificial Intelligence*, pages 557–561.

Tilman Becker. 1994. Patterns in Metarules. In *Proc. of the 3rd International Workshop on TAG and Related Frameworks (TAG+3)*, Paris, France.

E. J. Briscoe, A. Copestake, and V. de Paiva. 1994. *Inheritance, Defaults and the Lexicon.* Cambridge University Press.

Marie-Helene Candito. 1996. A Principle-Based Hierarchical Representation of LTAGs. In *Proc. of the 16th International Conference on Computational Linguistics (COLING-1996)*, Copenhagen, Denmark.

Bob Carpenter and Gerald Penn. 1999. ALE: The Attribute Logic Engine User's Guide. Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ, version 3.2, beta edition.

Noam Chomsky. 1981. *Lectures on Government and Binding.* Dordrecht: Foris.

Ann Copestake and Antonio Sanfilippo. 1993. Multilingual Lexical Representation. In *Proc. of the AAAI Spring Symposium: Building Lexicons for Machine Translation*, Stanford, California.

Roger Evans and Gerald Gazdar. 1989. Inference in DATR. In *Proc. of the 4th Conference of the European Chapter of the Association for Computational Linguistics (EACL-1989)*.

Roger Evans, Gerald Gazdar, and David Weir. 1995. Encoding Lexicalized Tree Adjoining Grammars with a Nonmonotonic Inheritance Hierarchy. In *Proc. of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-1995)*, Cambridge, MA.

Robert Frank. 2002. *Phrase Structure Composition and Syntactic Dependencies*. MIT Press, Cambridge, MA.

G. Gazdar, E.Klein, G. Pullum, and I. Sag. 1985. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford.

Ray S. Jackendoff. 1977. *X-bar Syntax: A Study of Phrase Structure*. Cambridge, MA: MIT Press.

Aravind Joshi and Yves Schabes. 1997. Tree Adjoining Grammars. In A. Salomma and G. Rosenberg, editors, *Handbook of Formal Languages and Automata*. Springer-Verlag, Herdelberg.

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

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

Aravind K. Joshi. 1985. Tree Adjoining Grammars: How Much Context Sensitivity Is Required to Provide a Reasonable Structural Description. In D. Dowty, I. Karttunen, and A. Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, Cambridge, U.K.

Laura Kallmeyer and Aravind Joshi. 1999. Factoring Predicate Argument and Scope Semantics: Underspecified Semantics with LTAG. In Paul Dekker, editor, *Proc. of the 12th Amsterdam Colloquium*, pages 169–174.

Ronald M. Kaplan and Joan Bresnan. 1982. Lexical Functional Grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. The MIT Press, Cambridge, MA.

Ronald M. Kaplan and John T. Maxwell, III. 1988. An Algorithm for Functional Uncertainty. In *Proc. of the* 12^{th} *International Conference on Computational Linguistics (COLING-1988)*, pages 297–302.

Ronald M. Kaplan and Annie Zaenen. 1988. Long-distance dependencies, constituent structure, and functional uncertainty. In M. Baltin and A. Kroch, editors, *Alternative Conceptions of Phrase Structure*. University of Chicago Press, Chicago.

R. Kasper, B. Kiefer, K. Netter, and K. Vijay-Shanker. 1995. Compilation of HPSG to TAG. In *Proc. of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-1995)*.

Karin Kipper, Hoa Trang Dang, and Martha Palmer. 2000. Class-based Construction of a Verb Lexicon. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI-2000)*.

Anthony S. Kroch and Aravind K. Joshi. 1985. The Linguistic Relevance of Tree Adjoining Grammars. Technical Report MS-CIS-85-16, Department of Computer and Information Science, University of Pennsylvania.

Anthony S. Kroch and Aravind K. Joshi. 1987. Analyzing Extraposition in a Tree Adjoining Grammar. In G. Huck and A. Ojeda, editors, *Discontinuous Constituents, Syntax and Semantics*, volume 20 of *Syntax and Semantics*. Academic Press.

Anthony S. Kroch. 1987. Unbounded Dependencies and Subjacency in a Tree Adjoining Grammar. In Alexis Manaster-Ramer, editor, *Mathematics of Language*. John Bebjamins Publishing Co, Amsterdam/Philadelphia.

Anthony S. Kroch. 1989. Asymmetries in Long Distance Extraction in a TAG Grammar. In M. Baltin and A. Kroch, editors, *Alternative Conceptions of Phrase Structure*. University of Chicago Press.

Beth Levin. 1993. *English Verb Classes and Alternations: A Preliminary Investigation*. The University of Chicago Press.

Emele Martin. 1994. TFS – The Typed Feature Structure Representation Formalism. In *Proc. of the International Workshop on Sharable Natural Language Resource (SNLR)*.

K. F. McCoy, K. Vijay-Shanker, and G. Yang. 1992. A Functional Approach to Generation with TAG. In *Proc. of the 30th Annual Meeting of the Association for Computational Linguistics (ACL-1992)*.

Martha Palmer, Owen Rambow, and Alexis Nasr. 1998. Rapid Prototyping of Domain-Specific Machine Translation System. In *Proc. of the Third Conference of the Association for Machine Translation in the Americas (AMTA-1998)*, Langhorne, PA.

Martha Palmer, Joseph Rosenzweig, and William Schuler. 1999. Capturing Motion Verb Generalizations with Synchronous TAG. In Patrick St. Dizier, editor, *Predicative Forms in NLP: Text, Speech and Language Technology Series*, pages 229–256. Kluwer Press, Dordrecht, The Netherlands.

Carl Pollard and Ivan A Sag. 1994. *Head-Driven Phrase Structure Grammar*. Chicago: The University of Chicago Press and Stanford: CSLI Publications.

James Rogers and K. Vijay-Shanker. 1994. Obtaining Trees from Their Descriptions: An Application to Tree Adjoining Grammars. *Journal of Computational Intelligence*, 10(4).

Anoop Sarkar. 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)*.

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

Masayoshi Shibatani, editor. 1976. *The Grammar of causative constructions*. New York: Academic Press.

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

Matthew Stone and Christine Doran. 1997. Sentence Planning as Description Using Tree Adjoining Grammar. In *Proc. of the 35th Annual Meeting of the Association for Computational Linguistics (ACL-1997)*.

K. Vijay-Shanker and Yves Schabes. 1992. Structure Sharing in Lexicalized Tree Adjoining Grammar. In *Proc. of the 14th International Conference on Computational Linguistics (COLING-1992)*, Nantes, France.

K. Vijay-Shanker. 1987. *A Study of Tree Adjoining Grammars*. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

K. Vijay-Shanker. 1992. Using Descriptions of Trees in a Tree Adjoining Grammar. *Computational Linguistics*, 18.

Bonnie Webber and Aravind Joshi. 1998. Anchoring a Lexicalized Tree Adjoining Grammar for Discourse. In *Proc. of the COLING-ACL'98 Workshop on Discourse Relations and Discourse Markers*.

Bonnie Webber, Alistair Knott, Matthew Stone, and Aravind Joshi. 1999. What Are Little Trees Made of: A Structural and Presuppositional Account Using Lexicalized TAG. In *Proc. of International Workshop on Levels of Representation in Discourse (LORID-1999)*.

Fei Xia, Martha Palmer, and K. Vijay-Shanker. 1999. Toward Semi-Automating Grammar Development. In *Proc. of the 5th Natural Language Processing Pacific Rim Symposium (NLPRS-1999)*, Beijing, China.

Fei Xia. 2001. *Automatic Grammar Generation from Two Different Perspectives*. Ph.D. thesis, University of Pennsylvania.

The XTAG-Group. 1995. A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS 95-03, University of Pennsylvania.

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

The XTAG-Group. 2001. A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS 01-03, University of Pennsylvania.

List of Tables

List of Figures