3

Developing Tree-Adjoining Grammars with Lexical Descriptions

FEI XIA, MARTHA PALMER, AND K. VIJAY-SHANKER

3.1 Introduction

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 (i.e., elementary trees with the lexical items removed) in an LTAG grammar often share some common structures, 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). Second, the underlying linguistic information (e.g., the analysis of wh-movement) is not expressed explicitly. 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.

To address these problems, we designed a grammar development system named LexOrg, which automatically generates 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 reductions in grammar development time. The system not only enables efficient development and maintenance of a grammar, but also

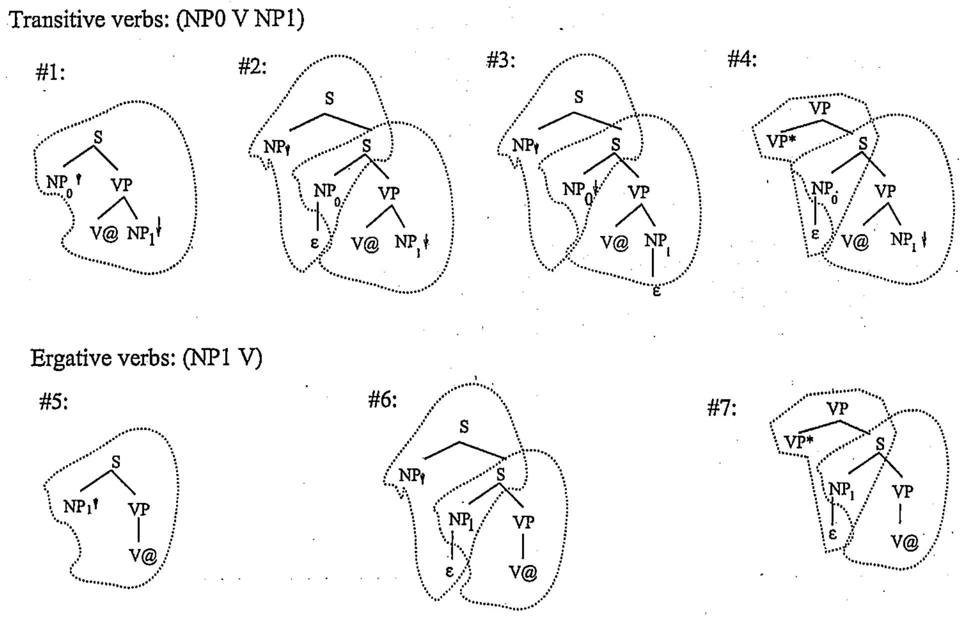


FIGURE 3.1 Templates in two tree families

allows underlying linguistic constructions (such as wh-movement) to be expressed explicitly.

The chapter is organized as follows. In section 3.2, we give an outline of the LexOrg system. In section 3.3, we define descriptions, trees, and four classes of descriptions. In sections 3.4 to 3.6, we describe the three main components of LexOrg. In section 3.7, we discuss how a user creates a set of abstract specifications for a language. section 3.8 includes a report on our experiments using LexOrg to generate grammars for English and Chinese. In section 3.9, we compare LexOrg with related work.

3.2 An Overview of LexOrg

To better understand the redundancy problem, let us look at an example. Figure 3.1 shows seven templates in two tree families:[1] 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 nineteen templates in the transitive verb's tree family.[2]

Among these seven templates, #1, #2, #3, and #4 all share the structure in figure 3.2a; templates #2, #3, and #6 all have the structure in figure 3.2b; templates

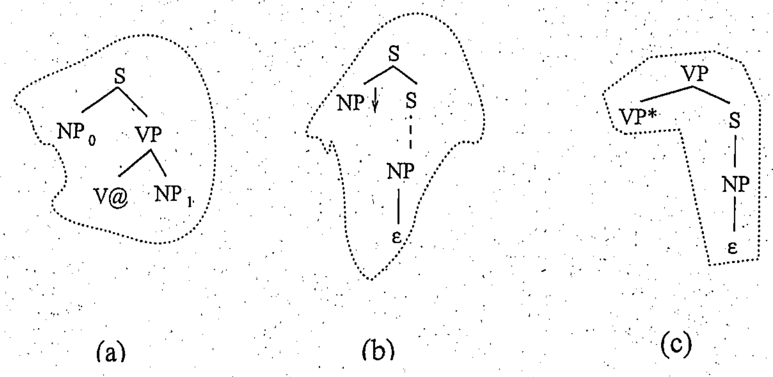


FIGURE 3.2 Structures shared by the templates in figure 3.1

#4 and #7 both have the structure in figure 3.2c. The dashed line in figure 3.2b 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*.

Because of the redundancy among templates, the process of manually creating and maintaining grammars is inefficient and cannot guarantee consistency. Nevertheless, if there exists a tool that combines these common structures to generate templates automatically (as illustrated in figure 3.3), 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 3.3b 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 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

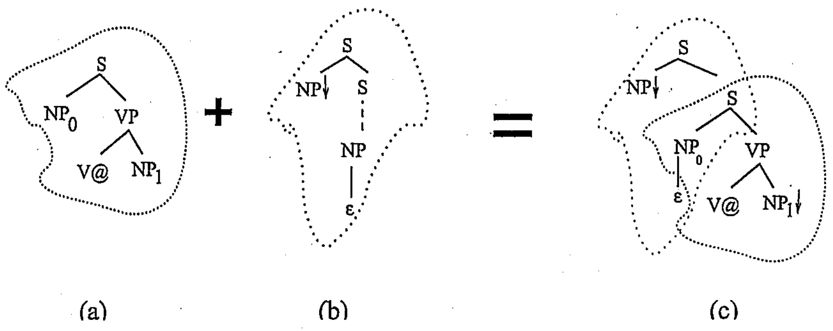


FIGURE 3.3 Combining descriptions to generate templates

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 made more rapid and 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.

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 specify which arguments of the lexical items 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.[3] 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

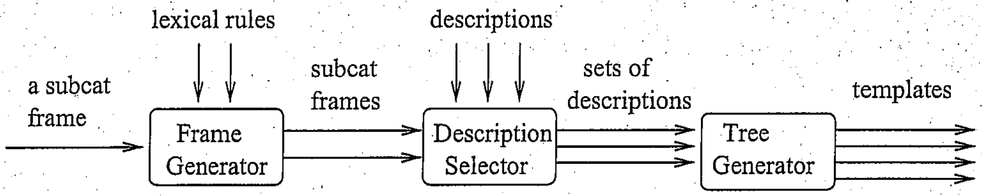


FIGURE 3.4 The architecture of LexOrg

are two additional kinds of descriptions: *modification descriptions* and *syntactic variation descriptions. Modification descriptions* are used to describe the other kind of elementary trees in LTAG, modifier auxiliary trees, 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 them out into the different kinds of specifications as described above because of their relationship to the different aspects of elementary trees and tree families, which are 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 3.8) 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 3.4 shows the architecture of the LexOrg system, which has three components: a Frame Generator, a Description Selector, and a Tree Generator. The inputs to the system are subcategorization frames, lexical subcategorization rules, and tree descriptions. The Frame Generator (described in section 3.6) 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 3.5) 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 3.4) 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.

3.3 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.

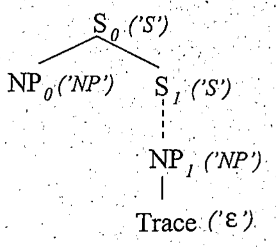
3.3.1 The definition of a *description*

In Rogers and Vijay-Shanker (1994), *descriptions* are defined to be formulas 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* for node types, 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 3.5a 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 treelike figure. Figure 3.5b 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

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

(a) the logical representation



(b) the graphical representation

FIGURE 3.5 Two representations of a description

following sections, we shall use the graphical representation when possible and use the logical representation in other cases.

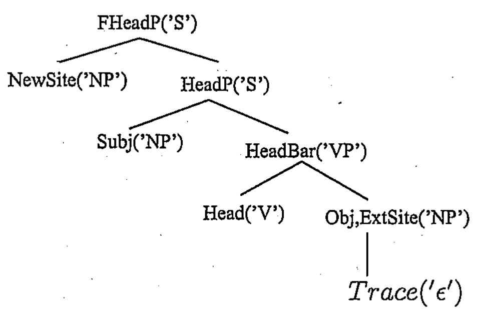
3.3.2 The definition of a *tree*

According to Rogers and Vijay-Shanker (1994), a *tree* is a structure that 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 *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).

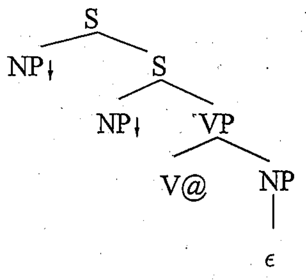
From each satisfiable description, we can always recover a minimal tree that satisfies the description. For example, figure 3.6a is a tree that satisfies the description given in figure 3.5. In this representation, a node has the form $\{k_i\}(\{f_m = v_m\})$, where $\{k_i\}$ is a list of node names, 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 3.6a, one such case is *Obj,ExtSite('NP')*. As we will 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 3.4, we shall show that it is trivial to build a unique template from such a representation.

3.3.3 Four classes of descriptions

In section 3.4, we shall demonstrate how a component of LexOrg, namely the Tree Generator, generates templates 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



(a) a tree



(b) the corresponding template

FIGURE 3.6 A tree and the template that is built from the tree

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.

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 3.7a 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 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.

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 3.7b says that the subject — a left argument of the head — is a sister of the *HeadBar.*[4] In this case, the feature equation in the description, given below the tree in figure 3.7b, specifies

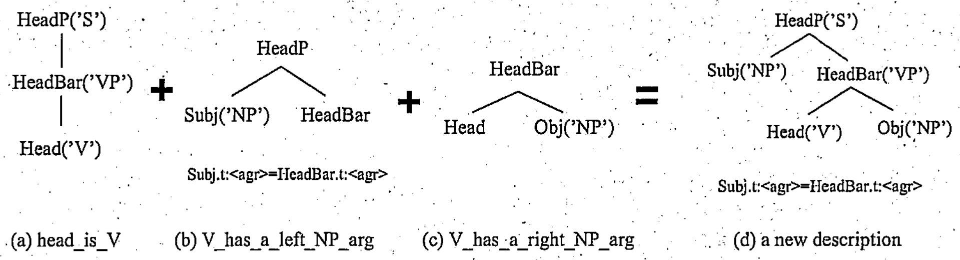


FIGURE 3.7 Subcategorization descriptions

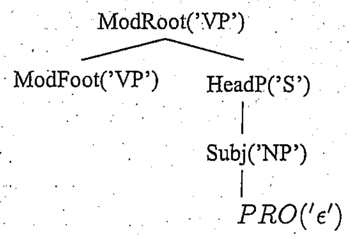


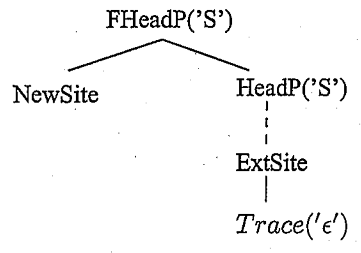
FIGURE 3.8 A description for purpose clauses

that the subject and the *HeadBar* must agree with respect to number, person, and so on. The description in figure 3.7c says that a head can take an *NP* argument, which appears as a right sister of the head. Combining head-projection and head-argument descriptions forms a description for the whole subcategorization frame, as in figure 3.7d; therefore, we use the term *subcategorization* description to refer to descriptions of either class. As we will describe in section 3.5, the Description Selector is responsible for choosing the set of 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.

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 3.8 says that a clause can modify a verb phrase from the right,[5] 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 resulting LTAG grammar, which are used to express modification in LTAG.



NewSite.t:<trace> = ExtSite.t:<trace>

FIGURE 3.9 A description for wh-movement

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 3.7d appears in every template in the transitive tree family, as shown on the top part of figure 3.1. 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 3.1 can be decomposed into two parts: the first part, as in figure 3.2a, is the basic structure that comes from head-projection and head-argument descriptions; the second part, as in figure 3.2b, comes from the description in figure 3.9. 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 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 argument, 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 3.9 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 3.5, we will show that another 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.

3.4 The Tree Generator

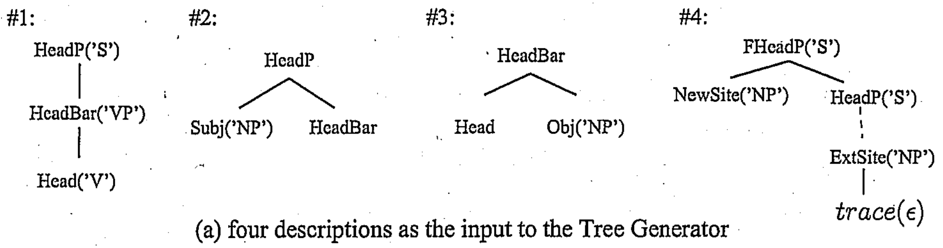
The most complex component of LexOrg is called the *Tree Generator* (*TreeGen*), which takes a set of descriptions as input and generates a set of templates as output. This is done in three steps: first, TreeGen combines the input set of descriptions to get a new description; second, TreeGen builds a set of trees such that each tree in the set satisfies the new description and has minimal number of nodes; third, TreeGen builds a template from each tree in the tree set. In figure 3.10, the descriptions in (a) are the inputs to TreeGen. Combining them results in a new description in (b).[6] There are many trees that satisfy this new description, but the two trees in (c) are the only ones with minimal number of nodes. From these two trees, TreeGen builds two templates in (d). In this section, we explain each step in detail.

3.4.1 Step 1: Combining descriptions to form a new description

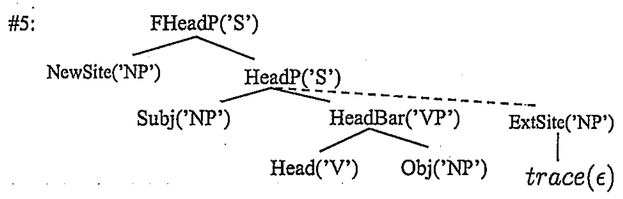
The Description Selector selects a set of descriptions that might potentially form one or more templates. TreeGen combines such a set of descriptions to form a new description. Recall that a description is a well-formed formula in a simplified first-order language. Given a set of descriptions $\{\phi_i\}$,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

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

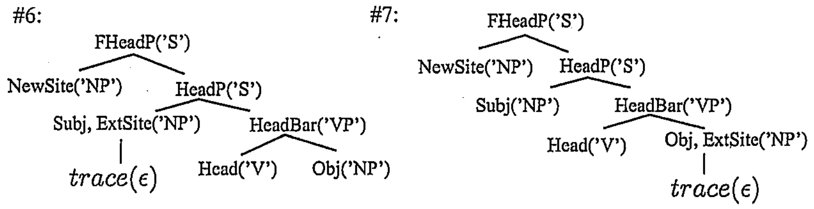
In the second step, TreeGen generates a set of trees, $TreeSet_{min}(\phi)$, for the new description ϕ. Let $TreeSet(\phi)$ be the set of trees that satisfies ϕ and $NumNodes(T)$



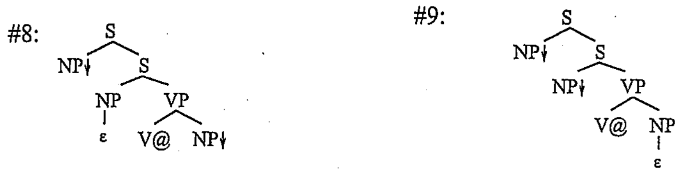
(a) four descriptions as the input to the Tree Generator



(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 3.10 The function of the Tree Generator

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,

$$TreeSet_{min}(\phi) = arg\, min_{T \in TreeSet(\phi)}\, NumNodes(T)$$

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 large number of trees that do not satisfy ϕ and have to be thrown away in later steps.[7]

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 the 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.[8] 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.

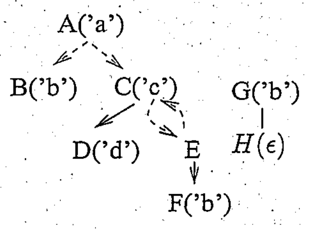
The new algorithm is in table 3.1. The major steps of the algorithm are illustrated in figure 3.11. The input description is in (a). Since the description is already in disjunctive normal form, TreeGen skips Step (A) in table 3.1. In steps (C1) and (C2), TreeGen creates a graphical representation for the description, as shown in figure 3.11b. 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.[9] 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 3.11b, and after merging, they become one node in the new graph, as shown in graph #3 in figure

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

// 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 $\psi_{i_j}$ 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) $TreeSet_m = \{tr \mid tr \in TC$ and $NumNodes(tr) = a\}$;
```

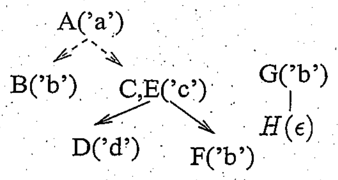
TABLE 3.1 A more efficient algorithm for building $TreeSet_{min}(\phi)$

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

(a) a description

#2:


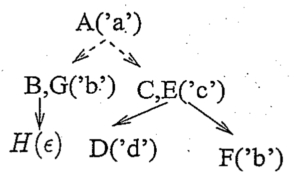
Left-of information: $B \prec E$

#3:


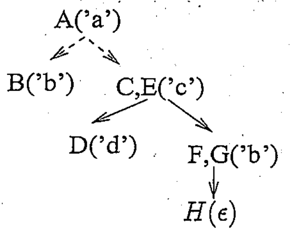
Left-of information: $B \prec C, E$

(b) a graph built from the description

(c) the graph after cycles are removed

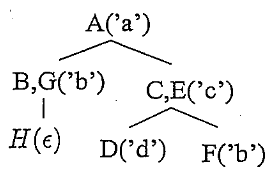
#4:


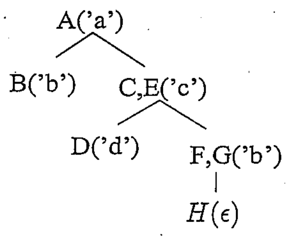
Left-of information: $B, G \prec C, E$

#5:


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

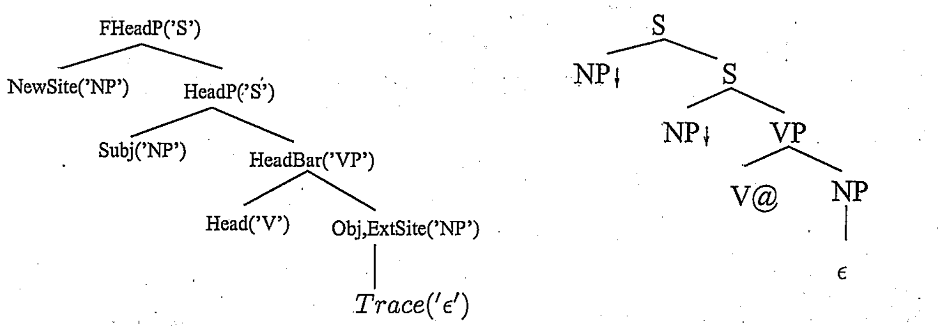
(d) the graphs after compatible sets are merged

#6:


#7:


(e) the trees built from the graphs

FIGURE 3.11 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 3.1; (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).



(a) a tree (b) the corresponding template

FIGURE 3.12 A tree and the template built from it

3.11c. 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.[10] In step (C5), for each graph produced 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 tree set 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 $TreeSet_m$.

3.4.3 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 3.6 (repeated as figure 3.12) shows a tree and the template built from the tree.

3.5 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 3.13, is described below.

3.5.1 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.[11] For instance, the subcategorization frame $(NP_0\ V\ NP_1)$ can be seen as the shorthand version of the description

$$(leftarg \prec head) \wedge (head \prec rightarg) \wedge (leftarg.cat = \text{'}NP\text{'}) \wedge (head.cat = \text{'}V\text{'})$$
$$\wedge (rightarg.cat = \text{'}NP\text{'}) \wedge (leftarg.subscript = 0) \wedge (rightarg.subscript = 1)$$

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

3.5.2 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 3.13) 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 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

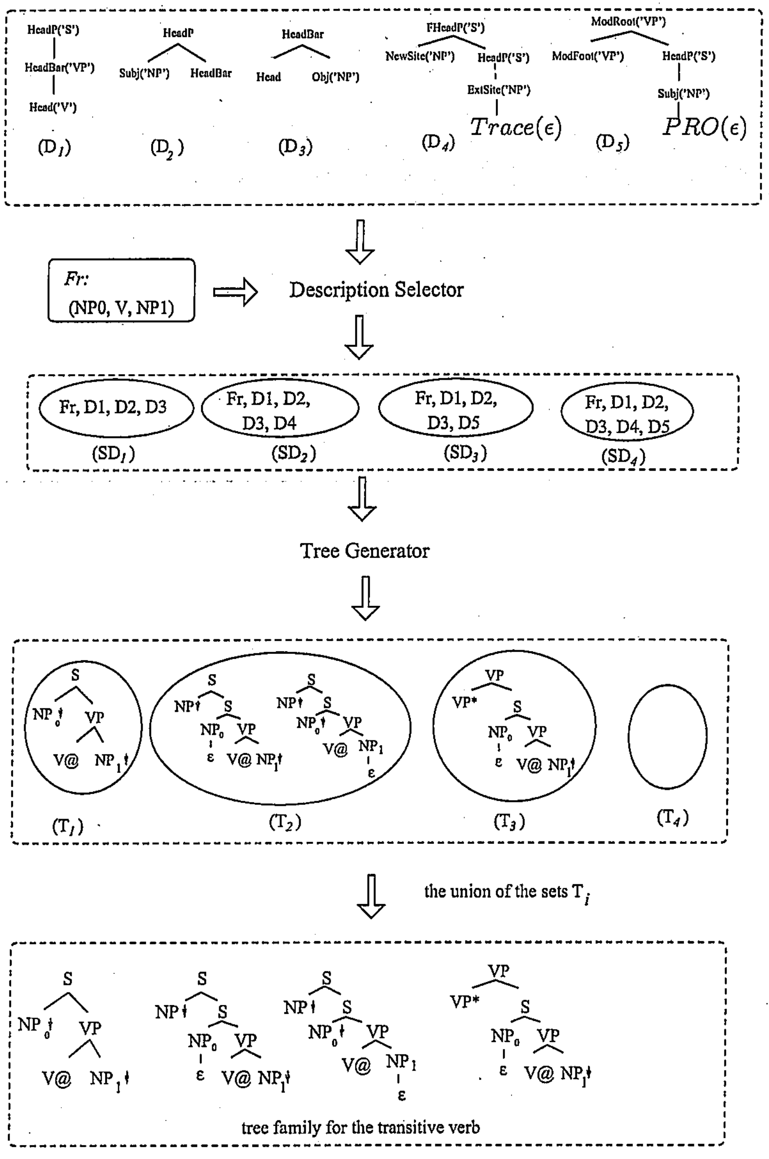


FIGURE 3.13 The function of the Description Selector

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 $SD_i \cup \{m'\}$.[12] This process is illustrated in figure 3.13. 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.

3.6 The Frame Generator

In an LTAG grammar, each word anchors one or more elementary trees. Figure 3.1 (repeated as figure 3.14) 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.[13] Figure 3.15 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.

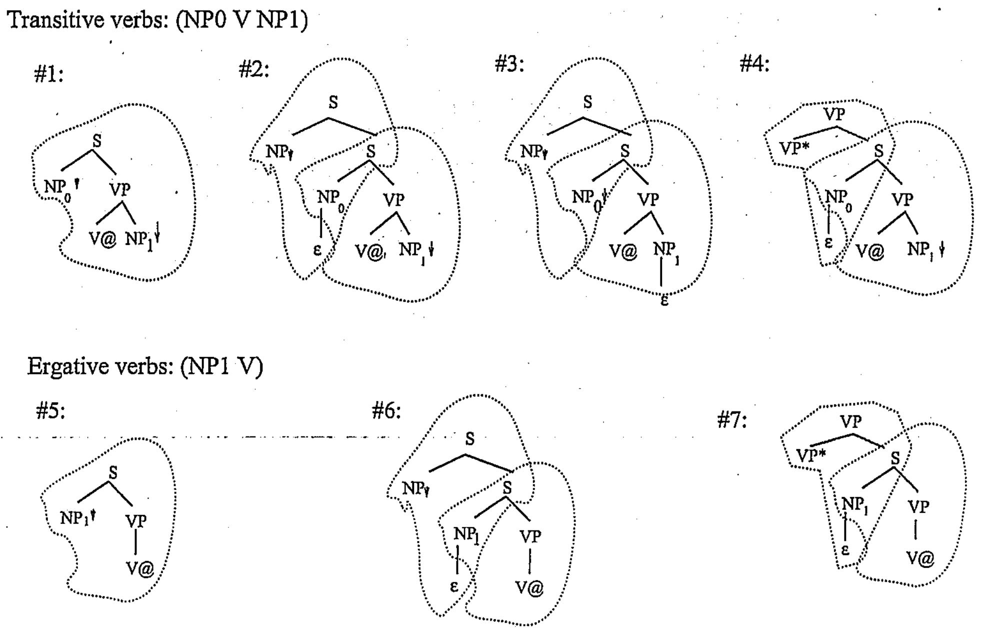


FIGURE 3.14 Templates in two tree families

$$(NP0\ V\ NP1) \Rightarrow (NP1\ V)$$

FIGURE 3.15 The lexical subcategorization rule for the causative/inchoative alternation

3.6.1 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.[14] Applying this rule to fr yields a new frame that 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 3.15 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 3.15 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 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.

3.6.2 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 3.2: Initially, $FrSet$ contains only one frame, Fr; the Frame Generator then applies each rule in *Rules* to each frame in $FrSet$, and appends the resulting new 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 3.2 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 and generate a new frame f_2 (which is added to $FrSet$); it may later apply another rule, r_2, to f_2 which generates f_3; and the process continues. 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.[15] Therefore, the algorithm in table 3.2 will always terminate.

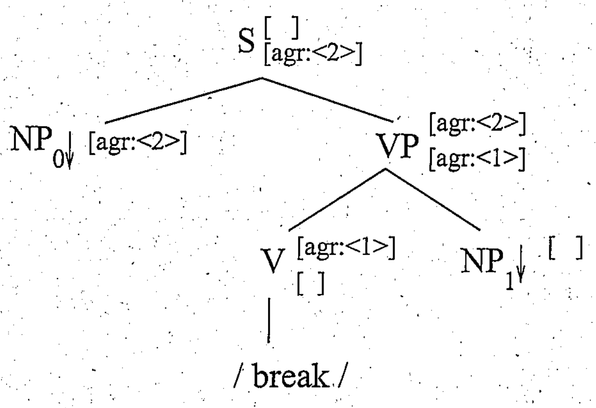


FIGURE 3.16 An elementary tree for the verb *break*

3.7 Creating abstract specifications

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*? In this section we briefly discuss our approach to this question.

Before we get into the details, let us first emphasize one point. 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 are used. The advantage of using LexOrg is that 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 3.16 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,[16] LexOrg will ensure that they are propagated to all relevant templates.

3.7.1 Subcategorization frames and lexical subcategorization rules

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 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.[17]

3.7.2 Tree descriptions

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 transformation variation description, we start with the definition of the corresponding phenomenon, which is language-independent. For example, *relative clause* can be roughly defined as *an NP is modified by a clause in which one constituent is extracted (or co-indexed with an operator).* We build a tree description (for clarity, we will call it *metablock*) according to the definition. Notice that the exact shape of the metablock often depends on the theory. For example, both metablocks in figure 3.17 are consistent with the definition of relative clause, the former follows the way that the Penn XTAG group treats the complementizer(COMP) as an adjunct, the latter follows more closely to the GB theory where COMP is the functional head of CP.

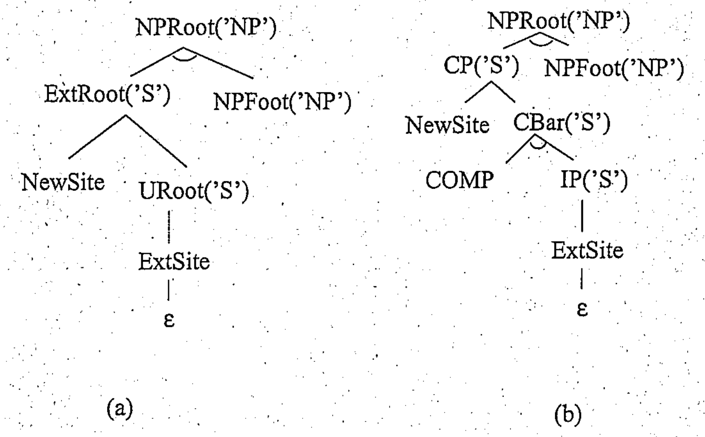
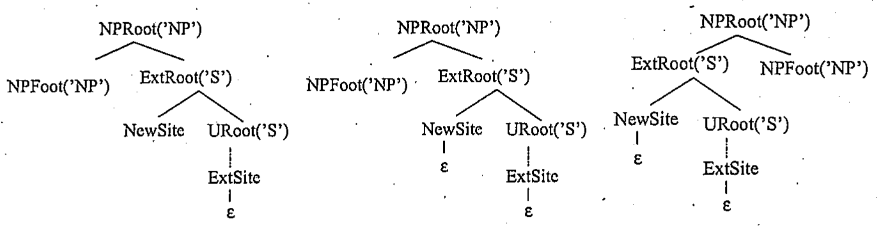


FIGURE 3.17 The possible metablocks for relative clause

	English	Portuguese	Chinese	Korean
position of NPFoot?	left	left	right	right
overt wh-movement?	yes	yes	no	no
has overt RelPron?	yes	yes	no	no
RelPron can be dropped?	yes*	yes*	-	-
position of COMP?	left	left	right	suffix
COMP can be dropped?	yes*	yes*	yes*	no
COMP and RelPron co-occurs?	no	no	-	-
COMP and RelPron both be dropped?	yes*	no	-	-

TABLE 3.3 Settings for relative clauses in four languages



(a) English and Portuguese with Relative Pronoun (b) English and Portuguese without Relative Pronoun (c) Chinese and Korean

FIGURE 3.18 The transformation variation descriptions for relative clauses in four languages

The metablocks must be general enough to be language-independent; therefore, certain relations in metablocks are are not fully specified. For instance, the order between the noun phrase and the relative clause in figure 3.17 is unspecified. To generate transformation variation descriptions for a particular language, metablocks have to be combined with language-specific information. We can elicit language-specific information by asking native speakers questions that are derived from the underspecification in metablocks.

For example, figure 3.17 shows the possible metablocks for relative clauses. Table 3.3 lists the questions about those metablocks and the answers for four languages. In a relative clause, a relative pronoun(RelPron) occupies the position marked by NewSite. If we choose the metablock in 3.17a, the top four questions should be asked, and the corresponding transformation variation descriptions are shown in figure 3.18.[18] If we choose the metablock in 3.17b, all the eight questions are relevant.

Several points are worth noting. First, the setting of some parameters follows from higher-level generalizations and some pairs of parameters are related. For example, the position of NPFoot follows from the head position in that language. Korean is an SOV language, so we can infer the position of the NPFoot without asking native speakers. Second, the setting of the parameters provides a way of measuring the similarities between the languages. According to the settings, Chinese is more similar to Korean than to English.

A word of caution is also in order. Both the construction of the metablock and the correct answers to the questions require some degree of linguistic expertise. Also, certain language-specific details cannot be easily expressed as yes-no questions. For example, the asterisk-marked answers in table 3.3 mean that they are true only under certain conditions; for instance, in English, COMP and RelPron can be both dropped only when the relativized NP is not the subject.

3.8 The Experiments

To test our implementation of LexOrg, we created two sets of abstract specifications (one for English and the other for Chinese) as discussed in the previous chapter. 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.

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 preexisting grammar 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

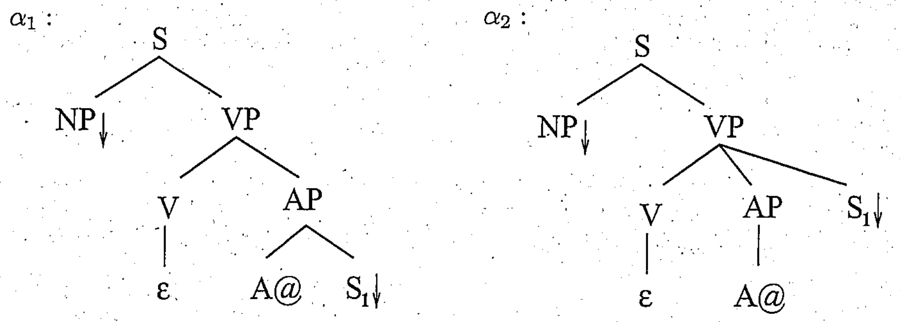


FIGURE 3.19 Two elementary trees for adjectives with sentential arguments

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 3.19 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 generate a grammar. This is because the head-argument description in figure 3.20, 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 3.19, 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

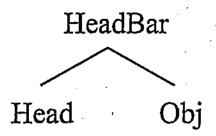


FIGURE 3.20 A head-argument description

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 3.4 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 3.4: 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.[19] 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 preexisting 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.

3.9 Comparison with Other Work

Systems such as Becker's HyTAG system (Becker, 1994), the one by Evans, Gazdar and Weir (1995) implemented in DATR (Evans and Gazdar, 1989), and Candito's system (Candito, 1996) have all been based on the same observation that motivated LexOrg; namely 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. This section briefly compares LexOrg to these other systems.[20]

In a lexical hierarchy, a class inherits attributes from its superclasses as illustrated by figure 3.21. (For a detailed example of a verb subcategorization frame hierarchy adhering to strict inheritance properties, see Copestake and Sanfilippo (1993) and Briscoe et al. (1994).) Although the hierarchy seems intuitive, it is difficult to build manually. Grammar designers first have to decide between a true hieararchy and a network. If a network is chosen, then conflicts between multiple superclasses must be resolved. The individual nodes also all need to be explicitly defined. For instance, could the nodes in figure 3.21 for *TRANSITIVE*, *SIMPLE-TRANS*, and *NP-IOBJ* be merged, or do they need to be distinct?

One major difference between LexOrg and the other three approaches is that LexOrg does not depend on a predefined hierarchy. The inheritance relations between tree families are implicit. For instance, the description set selected by LexOrg 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 needing to refer directly to an explicit hierarchy or to the transitive family. [21]

3.9.1 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

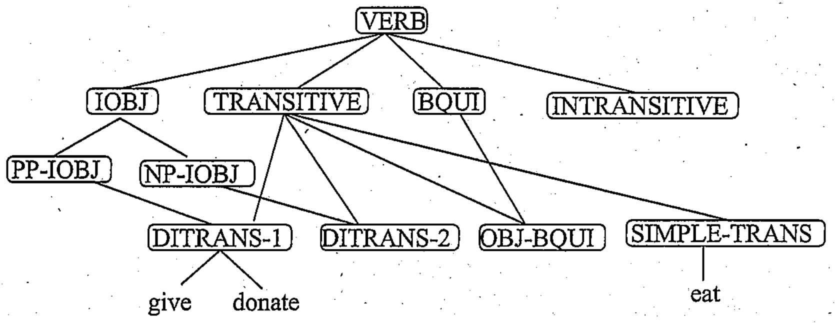


FIGURE 3.21 The lexical hierarchy given in Vijay-Shanker and Schabes (1992)

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 a 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 metavariable. A *metavariable* describes part of a template that is not 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.

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 metavariables. 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.

3.9.2 The DATR system

Evans, Gazdar and Weir (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. 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.

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 nonmonotonicity in its application of rules; (2) the DATR system can capture only direct relations between nodes in a tree (such as the parent-child relationship or precedence), and must use feature-equations to simulate other tree relations (such as dominance relation). This means that in their system, an abstract concept such as dominance must be specified by spelling out explicitly all of the different possible path lengths for every possible dominance relationship.

3.9.3 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 handwritten hierarchy that has three dimensions. In the first dimension, canonical 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 superclasses 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. To ensure that all the superclasses of a terminal class are compatible, the system provides several ways for its users to explicitly express compatibility constraints.

These are not needed in LexOrg, which automatically ensures that illegal combinations are ruled out.

There are many similarities between Candito's system and LexOrg 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.

First, Candito's system requires a handwritten hierarchy, whereas LexOrg does not. It 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.

3.10 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, as with similar hand-crafted grammars, the larger the grammar, the more prone to errors it becomes, and the harder it is to maintain.

LexOrg is a computational tool that alleviates these 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.

Given a preexisting 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 that 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 preexisting 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.

3.11 Acknowledgments

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. A longer version of this chapter was published in the *Journal of Computational Intelligence*, and we would also like to thank the journal and Blackwell Publishing for granting us the permission to include this chapter in the book.

Notes

1. A tree family is a set of elementary trees that have the same subcategorization frame.

2. The XTAG grammar (XTAG-Group, 1998, 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.

3. 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.

4. 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 chapter (such as in figure 3.7), 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.

5. 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*.

6. Notice that in figure 3.10b the position of *ExtSite* with respect to *Subj* and *HeadBar* is not specified.

7. 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:

$$b_n = \frac{1}{n+1} \times \binom{2n}{n} = \frac{4^n}{\sqrt{\pi} \times n^{3/2}} \times (1 + O(1/n)).$$

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 $TreeSet_{min}(\phi)$.

8. In first-order logic, two formulas 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 require only 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 $\hat{\phi}_i$.

9. 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 $\phi \wedge (x = y)$.

10. 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.

11. 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.

12. 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.

13. 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 chapter we rename the rules in LexOrg as *lexical subcategorization rules*, following a suggestion from one of the anonymous reviewers.

14. 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.

15. 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.

16. 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 3.7a, and the equation $VP.t :< agr >= NP_0.t :< agr >$ to the one in figure 3.7b.

17. 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.

18. The descriptions for relative clause in English and Portuguese look the same as in figure 3.18a and 3.18b but they differ in one aspect: when the ExtSite is not the subject, in English, both COMP and NewSite are optional, but in Portuguese, one of them must be present. The difference is captured by features which are not shown in the figure.

19. The remaining 10% of the templates are like α_2 in figure 3.19 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 3.19 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.

20. For more details of these systems and the comparisons, see Xia et al. (2005) or Chapter 4 of Xia (2001).

21. If one wishes to make explicit this implicit inheritance hierarchy, it can be built 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.

References

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

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

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

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

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

Evans, R. and Gazdar, G. (1989). Inference in DATR. In *Proc. of the 4th Conference of the European Chapter of the Association for Computational Linguistics (EACL-1989)*, Manchester, England.

Evans, R., Gazdar, G., and Weir, D. J. (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, Massachusetts, USA.

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

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

Levin, B. (1993). *English Verb Classes and Alternations: A Preliminary Investigation.* The University of Chicago Press, Chicago, USA.

Pollard, C. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press.

Rogers, J. and Vijay-Shanker, K. (1994). Obtaining Trees from Their Descriptions: An Application to Tree Adjoining Grammars. *Journal of Computational Intelligence*, 10(4):401–421.

Shibatani, M., ed. (1976). *The Grammar of Causative Constructions.* Academic Press.

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

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

Xia, F., Palmer, M., and Vijay-Shanker, K. (2005). Automatically Generating Tree Adjoining Grammars from Abstract Specifications. *Computational Intelligence*, 21(3):246–287.

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

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