Lexicalized Context-Free Grammars

Yves Schabes and Richard C. Waters
Mitsubishi Electric Research Laboratories
201 Broadway, Cambridge, MA 02139
e-mail: schabes@merl.com and dick@merl.com

Lexicalized context-free grammar (LCFG) is an attractive compromise between the parsing efficiency of context-free grammar (CFG) and the elegance and lexical sensitivity of lexicalized tree-adjoining grammar (LTAG). LCFG is a restricted form of LTAG that can only generate context-free languages and can be parsed in cubic time. However, LCFG supports much of the elegance of LTAG's analysis of English and shares with LTAG the ability to lexicalize CFGs without changing the trees generated.

Motivation

Context-free grammar (CFG) has been a well accepted framework for computational linguistics for a long time. While it has drawbacks, including the inability to express some linguistic constructions, it has the virtue of being computationally efficient, $O(n^3)$-time in the worst case.

Recently there has been a gain in interest in the so-called 'mildly' context-sensitive formalisms (Vijay-Shanker, 1987; Weir, 1988; Joshi, Vijay-Shanker, and Weir, 1991; Vijay-Shanker and Weir, 1993a) that generate only a small superset of context-free languages. One such formalism is lexicalized tree-adjoining grammar (LTAG) (Schabes, Abeillé, and Joshi, 1988; Abeillé et al., 1990; Joshi and Schabes, 1992), which provides a number of attractive properties at the cost of decreased efficiency, $O(n^6)$-time in the worst case (Vijay-Shanker, 1987; Schabes, 1991; Lang, 1990; Vijay-Shanker and Weir, 1993b).

An LTAG lexicon consists of a set of trees each of which contains one or more lexical items. These elementary trees can be viewed as the elementary clauses (including their transformational variants) in which the lexical items participate. The trees are combined by substitution and adjunction.

LTAG supports context-sensitive features that can capture some language constructs not captured by CFG. However, the greatest virtue of LTAG is that it is lexicalized and supports extended domains of locality. The lexical nature of LTAG is of linguistic interest, since it is believed that the descriptions of many linguistic phenomena are dependent upon lexical data. The extended domains allow for the localization of most syntactic and semantic dependencies (e.g., filler-gap and predicate-argument relationships).

A further interesting aspect of LTAG is its ability to lexicalize CFGs. One can convert a CFG into an LTAG that preserves the original trees (Joshi and Schabes, 1992).

Lexicalized context-free grammar (LCFG) is an attractive compromise between LTAG and CFG, that combines many of the virtues of LTAG with the efficiency of CFG. LCFG is a restricted form of LTAG that places further limits on the elementary trees that are possible and on the way adjunction can be performed. These restrictions limit LCFG to producing only context-free languages and allow LCFG to be parsed in $O(n^3)$-time in the worst case. However, LCFG retains most of the key features of LTAG enumerated above.

In particular, most of the current LTAG grammar for English (Abeillé et al., 1990) follows the restrictions of LCFG. This is of significant practical interest because it means that the processing of these analyses does not require more computational resources than CFGs.

In addition, any CFG can be transformed into an equivalent LCFG that generates the same trees (and therefore the same strings). This result breaks new ground, because heretofore every method of lexicalizing CFGs required context-sensitive operations (Joshi and Schabes, 1992).

The following sections briefly, define LCFG, discuss its relationship to the current LTAG grammar for English, prove that LCFG can be used to lexicalize CFG, and present a simple cubic-time parser for LCFG. These topics are discussed in greater detail in Schabes and Waters (1993).

Lexicalized Context-Free Grammars

Like an LTAG, an LCFG consists of two sets of trees: initial trees, which are combined by substitution and auxiliary trees, which are combined by adjunction. An LCFG is lexicalized in the sense that every initial and auxiliary tree is required to contain at least one terminal symbol on its frontier.

More precisely, an LCFG is a five-tuple (Σ, NT, I, A, S), where Σ is a set of terminal symbols, NT is a set of non-terminal symbols, I and A are sets of trees labeled by terminal and non-terminal symbols, and S is a distinguished non-terminal start symbol.

Each initial tree in the set I satisfies the following requirements.

(i) Interior nodes are labeled by non-terminal symbols.
(ii) The nodes on the frontier of the tree consist of zero or more non-terminal symbols and one or more terminal symbols.
(iii) The non-terminal symbols on the frontier are marked for substitution. By convention, this is annotated in diagrams using a down arrow ($\downarrow$).

Each auxiliary tree in the set A satisfies the following requirements.

(i) Interior nodes are labeled by non-terminal symbols.
(ii) The nodes on the frontier consist of zero or more non-terminal symbols and one or more terminal symbols.
(iii) All but one of the non-terminal symbols on the frontier are marked for substitution.
(iv) The remaining non-terminal on the frontier of the tree is called the *foot*. The label on the foot must be identical to the label on the root node of the tree. By convention, the foot is indicated in diagrams using an asterisk ($*$).
(v) the foot must be in either the leftmost or the rightmost position on the frontier.

Figure 1, shows seven elementary trees that might appear in an LCFG for English. The trees containing 'boy', 'saw', and 'left' are initial trees. The remainder are auxiliary trees.

Auxiliary trees whose feet are leftmost are called *left recursive*. Similarly, auxiliary trees whose feet are rightmost are called *right recursive* auxiliary trees. The path from the root of an auxiliary tree to the foot is called the *spine*.

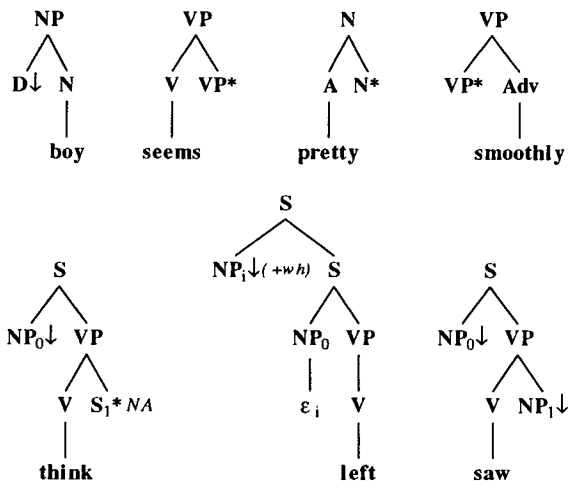


Figure 1: Sample trees.

In LCFG, trees can be combined with substitution and adjunction. As illustrated in Figure 2, substitution replaces a node marked for substitution with a copy of an initial tree.

Adjunction inserts a copy of an auxiliary tree into another tree in place of an interior node that has the same label as the foot of the auxiliary tree. The subtree that was previously connected to the interior node is reconnected to the foot of the copy of the auxiliary tree. If the auxiliary tree is left recursive, this is referred to as left recursive adjunction (see Figure 3). If the auxiliary tree is right recursive, this is referred to as right recursive adjunction (see Figure 4).

Crucially, adjunction is constrained by requiring that a left recursive auxiliary tree cannot be adjoined on any node that is on the spine of a right recursive auxiliary tree and a right recursive auxiliary tree cannot be adjoined on the spine of a left recursive auxiliary tree.

An LCFG derivation must start with an initial tree rooted in S. After that, this tree can be repeatedly extended using substitution and adjunction. A derivation is complete when every frontier node is labeled with a terminal symbol.

The difference between LCFG and LTAG is

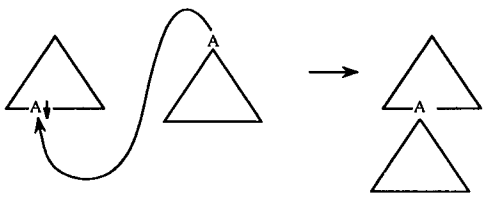


Figure 2: Substitution.

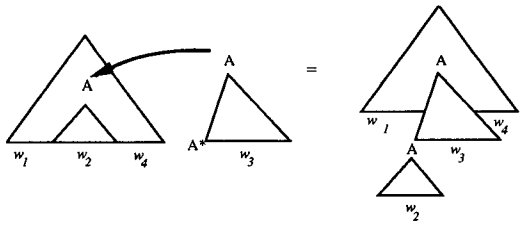


Figure 3: Left recursive adjunction.

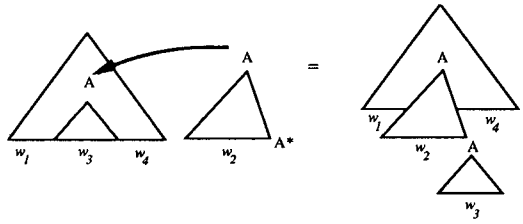


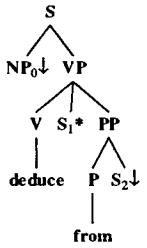
Figure 4: Right recursive adjunction.

that LTAG allows the foot of an auxiliary tree to appear anywhere on the frontier and places no limitations on the interaction of auxiliary trees. In this unlimited situation, adjunction encodes string wrapping and is therefore more powerful than concatenation (see Figure 5). However, the restrictions imposed by LCFG guarantee that no context-sensitive operations can be achieved. They limit the languages that can be generated by LCFGs to those that can be generated by CFGs.

Coverage of LCFG and LTAG

The power of LCFG is significantly less than LTAG. Surprisingly, it turns out that there are only two situations where the current LTAG grammar for English (Abeillé et al., 1990) fails to satisfy the restrictions imposed by LCFG.

The first situation, concerns certain verbs that take more than one sentential complement. An example of such a verb is *deduce*, which is associated with the following auxiliary tree.



Since this tree contains a foot node in the center of its frontier, it is not part of an LCFG. Having the foot on the first sentential complement is convenient, because it allows one to use the standard LTAG wh-analyses, which depends on the

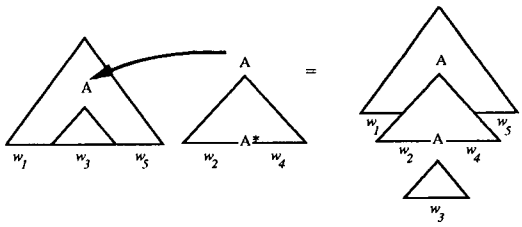


Figure 5: Adjunction in LTAG.

existence of an initial tree where the filler and gap are local. This accounts nicely for the pair of sentences below. However, other analyses of wh questions may not require the use of the auxiliary tree above.

(1) John deduced that Mary watered the grass from seeing the hose.
(2) What did John deduce that Mary watered from seeing the hose.

The second situation, concerns the way the current LTAG explains the ambiguous attachments of adverbial modifiers. For example, in the sentence:

(3) John said Bill left yesterday.

the attachment of *yesterday* is ambiguous. The two different LTAG derivations indicated in Figure 6 represent this conveniently.

Unfortunately, in LCFG the high attachment of *yesterday* is forbidden since a right auxiliary tree (corresponding to *yesterday*) is adjoined on the spines of a left auxiliary tree (corresponding to *John said*). However, one could avoid this problem by designing a mechanism to recover the high attachment reading from the low one.

Besides the two cases presented above, the current LTAG for English uses only left and right recursive auxiliary trees and does not allow any

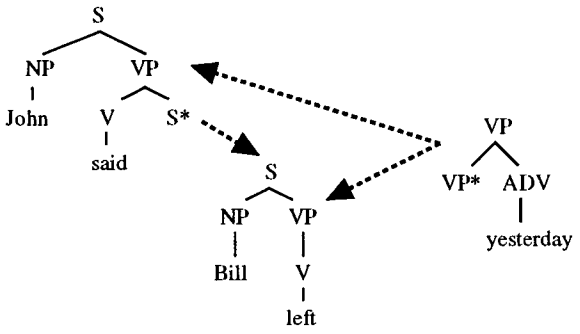


Figure 6: Two LTAG derivations for *John said Bill left yesterday.*

interaction along the spine of these two kinds of trees. This agrees with the intuition that most English analyses do not require a context-sensitive operation.

Lexicalization of CFGs

The lexicalization of grammar formalisms is of interest from a number of perspectives. It is of interest from a linguistic perspective, because most current linguistic theories give lexical accounts of a number of phenomena that used to be considered purely syntactic. It is of interest from a computational perspective, because lexicalized grammars can be parsed significantly more efficiently than non-lexicalized ones (Schabes and Joshi, 1990).

Formally, a grammar is said 'lexicalized' (Schabes, Abeillé, and Joshi, 1988) if it consists of:

- a finite set of elementary structures of finite size, each of which contains an overt (i.e., non-empty) lexical item.
- a finite set of operations for creating derived structures.

The overt lexical item in an elementary structure is referred to as its anchor. A lexicalized grammar can be organized as a lexicon where each lexical item is associated with a finite number of structures for which that item is the anchor.

In general, CFGs are not lexicalized since rules such as $S \rightarrow NP\ VP$ that do not locally introduce lexical items are allowed. In contrast, the well-known Greibach Normal Form (GNF) for CFGs is lexicalized, because every production rule is required to be of the form $A \rightarrow a\alpha$ (where a is a terminal symbol, A a non-terminal symbol and α a possibly empty string of non-terminal symbols) and therefore locally introduces a lexical item a.

It can be shown that for any CFG G (that does not derive the empty string), there is a GNF grammar G' that derives the same language. However, it may be impossible for the set of trees produced by G' to be the same as the set of trees produced by G.

Therefore, GNF achieves a kind of lexicalization of CFGs. However, it is only a *weak* lexicalization, because the set of trees is not necessarily preserved. As discussed in the motivation section, *strong* lexicalization that preserves tree sets is possible using LTAG. However, this is achieved at the cost of significant additional parsing complexity.

Heretofore, several attempts have been made to lexicalize CFG with formalisms weaker than LTAG, but without success. In particular, it is not sufficient to merely extend substitution so that it applies to trees. Neither is it sufficient to rely solely on the kind restricted adjunction used by LCFG. However, as shown below, combining extended substitution with restricted adjunction allows strong lexicalization of CFG, without introducing greater parsing complexity than CFG.

Theorem *If $G = (\Sigma, NT, P, S)$ is a finitely ambiguous CFG which does not generate the empty string (ε), then there is an LCFG $G' = (\Sigma, NT, I, A, S)$ generating the same language and tree set as G. Furthermore G' can be chosen so that it utilizes only left-recursive auxiliary trees.*

As usual in the above, a CFG G is a four-tuple, (Σ, NT, P, S), where Σ is a set of terminal symbols, NT is a set of non-terminal symbols, P is a set of production rules that rewrite non-terminal symbols to strings of terminal and non-terminal symbols, and S is a distinguished non-terminal symbol that is the start symbol of any derivation.

To prove the theorem we first prove a somewhat weaker theorem and then extend the proof to the full theorem. In particular, we assume for the moment that the set of rules for G does not contain any empty rules of the form $A \rightarrow \varepsilon$.

Step 1 We begin the construction of G' by constructing a directed graph LCG that we call the *left corner derivation graph.* Paths in LCG correspond to leftmost paths from root to frontier in (partial) derivation trees rooted at non-terminal symbols in G.

> LCG contains a node for every symbol in $\Sigma \cup NT$ and an arc for every rule in P as follows. For each terminal and non-terminal symbol X in G create a node in LCG labeled with X. For each rule $X \rightarrow Y\alpha$ in G create a directed arc labeled with $X \rightarrow Y\alpha$ from the node labeled with X to the node labeled Y.

As an example, consider the example CFG in Figure 7 and the corresponding LCG shown in Figure 8.

The significance of LCG is that there is a one-to-one correspondence between paths in LCG ending on a non-terminal and left corner derivations in G. A left corner derivation in a CFG is a partial derivation starting from any non-terminal where every expanded node (other than the root) is the leftmost child of its parent and the left corner is a non-terminal. A left corner derivation is uniquely identified by the list of rules applied. Since G does not have any empty rules, every rule in G is represented in LCG. Therefore, every path in LCG ending on a terminal corresponds to a left corner derivation in G and vice versa.

$$S \rightarrow A\ A$$
$$S \rightarrow B\ A$$
$$A \rightarrow B\ B$$
$$B \rightarrow A\ S$$
$$B \rightarrow b$$

Figure 7: An example grammar.

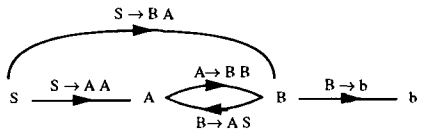


Figure 8: The LCG created by Step 1.

Step 2 The set of initial trees I for G' is constructed with reference to LCG. In particular, an initial tree is created corresponding to each non-cyclic path in LCG that starts at a non-terminal symbol X and ends on a terminal symbol y. (A non-cyclic path is a path that does not touch any node twice.)

> For each non-cyclic path in LCG from X to y, construct an initial tree T as follows. Start with a root labeled X. Apply the rules in the path one after another, always expanding the left corner node of T. While doing this, leave all the non-left corner non-terminal symbols in T unexpanded, and label them as substitution nodes.

Given the previous example grammar, this step produces the initial trees shown in Figure 9.

Each initial tree created is lexicalized, because each one has a non-terminal symbol as the left corner element of its frontier. There are a finite number of initial trees, because the number of non-cyclic paths in LCG must be finite. Each initial tree is finite in size, because each non-cyclic path in LCG is finite in length.

Most importantly, The set of initial trees is the set of non-recursive left corner derivations in G.

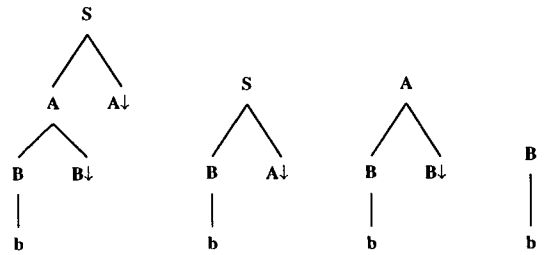


Figure 9: Initial trees created by Step 2.

Step 3 This step constructs a set of left-recursive auxiliary trees corresponding to the cyclic path segments in LCG that were ignored in the previous step. In particular, an auxiliary tree is created corresponding to each minimal cyclic path in LCG that starts at a non-terminal symbol.

> For each minimal cycle in LCG from X to itself, construct an auxiliary tree T by starting with a root labeled X and repeatedly expanding left corner frontier nodes using the rules in the path as in Step 2. When all the rules in the path have been used, the left corner frontier node in T will be labeled X. Mark this as the foot node of T. While doing the above, leave all the other non-terminal symbols in T unexpanded, and label them all substitution nodes.

The LCG in Figure 8 has two minimal cyclic paths (one from A to A via B and one from B to B via A). This leads to the the two auxiliary trees shown in Figure 10, one for A and one for B.

The auxiliary trees generated in this step are not necessarily lexicalized. There are a finite number of auxiliary trees, since the number of minimal cyclic paths in G must be finite. Each auxiliary tree is finite in size, because each minimal-cycle in LCG is finite in length.

The set of trees that can be created by combining the initial trees from Step 2 with the auxiliary trees from Step 3 by adjoining auxiliary trees along the left edge is the set of every left corner derivation in G. To see this, consider that every path in LCG can be represented as an initial non-cyclic path with zero or more minimal cycles inserted into it.

The set of trees that can be created by combining the initial trees from Step 2 with the auxiliary trees from Step 3 using both substitution and adjunction is the set of every derivation in G. To see this, consider that every derivation in G can be decomposed into a set of left corner derivations in G that are combined with substitution. In particular, whenever a non-terminal node is not the leftmost child of its parent, it is the head of a sep-

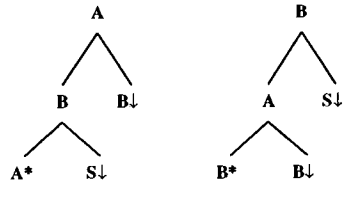


Figure 10: Auxiliary trees created by Step 3.

arate left corner derivation.

Step 4 This step lexicalizes the set of auxiliary trees built in step 3, without altering the trees that can be derived.

> For each auxiliary tree T built in step 3, consider the frontier node A just to the right of the foot. If this node is a terminal do nothing. Otherwise, remove T from the set of auxiliary trees replace it with every tree that can be constructed by substituting one of the initial trees created in Step 2 for the node A in T.

In the case of our continuing example, Step 4 results in the set of auxiliary trees in Figure 11.

Note that since G is finitely ambiguous, there must be a frontier node to the right of the foot of an auxiliary tree T. If not, then T would correspond to a derivation $X \stackrel{*}{\Rightarrow} X$ in G and G would be infinitely ambiguous.

After Step 4, every auxiliary tree is lexicalized, since every tree that does not have a terminal to the right of its foot is replaced by one or more trees that do. Since there were only a finite number of finite initial and auxiliary trees to start with, there are still only a finite number of finite auxiliary trees.

The change in the auxiliary trees caused by Step 4 does not alter the set of trees that can be produced in any way, because the only change that was made was to make substitutions that could be made anyway, and when a substitutable node was eliminated, this was only done after every possible substitution at that node was performed.

Note that the initial trees are left anchored and the auxiliary trees are almost left anchored in the sense that the leftmost frontier node other than the foot is a terminal. This facilitates efficient left to right parsing.

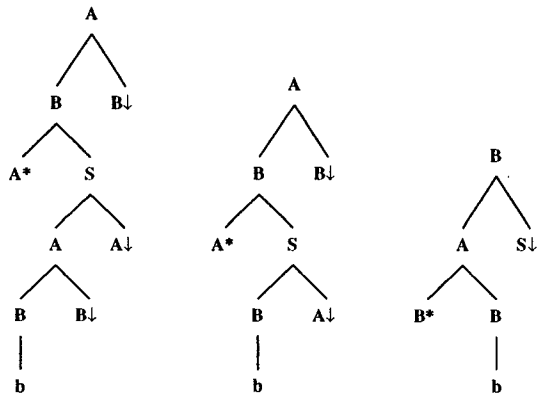


Figure 11: Auxiliary trees created by Step 4.

The procedure above creates a lexicalized grammar that generates exactly the same trees as G and therefore the same strings. The only remaining issue is the additional assumption that G does not contain any empty rules.

If G contains an empty rule $A \rightarrow \varepsilon$ one first uses standard methods to transform G into an equivalent grammar H that does not have any such rule. When doing this, create a table showing how each new rule added is related to the empty rules removed. Lexicalize H producing H' using the procedure above. Derivations in H' result in elements of the tree set of H. By means of the table recording the relationship between G and H, these trees can be converted to derivations in G.

□

Additional issues

There are several places in the algorithm where greater freedom of choice is possible. For instance, when lexicalizing the auxiliary trees created in Step 3, you need not do anything if there is any frontier node that is a terminal and you can choose to expand any frontier node you want. For instance you might want to choose the node that corresponds to the smallest number of initial trees.

Alternatively, everywhere in the procedure, the word 'left' can be replaced by 'right' and vice versa. This results in the creation of a set of right anchored initial trees and right recursive auxiliary trees. This can be of interest when the right corner derivation graph has less cycles than the left corner one.

The number of trees in G' is related to the number of non-cyclic and minimal cycle paths in LCG. In the worst case, this number rises very fast as a function of the number of arcs in LCG, (i.e., in the number of rules in G). (A fully connected graph of n^2 arcs between n nodes has $n!$ acyclic paths and $n!$ minimal cycles.) However, in the typical case, this kind of an explosion of trees is unlikely.

Just as there can be many ways for a CFG to derive a given string, there can be many ways for an LCFG to derive a given tree. For maximal efficiency, it would be desirable for the grammar G' produced by the procedure above to have no ambiguity in they way trees are derived. Unfortunately, the longer the minimal cycles in LCG, the greater the tree-generating ambiguity the procedure will introduce in G'. However, by modifying the procedure to make use of constraints on what auxiliary trees are allowed to adjoin on what nodes in which initial trees, it should be possible to reduce or even eliminate this ambiguity.

All these issues are discussed at greater length

in Schabes and Waters (1993).

Parsing LCFG

Since LCFG is a restricted case of tree-adjoining grammar (TAG), standard $O(n^6)$-time TAG parsers (Vijay-Shanker, 1987; Schabes, 1991; Lang, 1990) can be used for parsing LCFG. Further, they can be straightforwardly modified to require at most $O(n^4)$-time when applied to LCFG. However, this still does not take full advantage of the context-freeness of LCFG.

This section describes a simple bottom-up recognizer for LCFG that is in the style of the CKY parser for CFG. The virtue of this algorithm is that it shows in a simple manner how the $O(n^3)$-time worst case complexity can be achieved for LCFG. Schabes and Waters (1993) describes a more practical and more elaborate (Earley-style) recognizer for LCFG, which achieves the same bounds.

Suppose that $G = (\Sigma, NT, I, A, S)$ is an LCFG and that $a_1 \cdots a_n$ is an input string. We can assume without loss of generality[1] that every node in $I \cup A$ has at most two children.

Let η be a node in an elementary tree (identified by the name of the tree and the position of the node in the tree). The central concept of the algorithm is the concepts of *spanning* and *covering*. η spans a string $a_{i+1} \cdots a_j$ if and only if there is some tree derived by G for which it is the case that the fringe of the subtree rooted at η is $a_{i+1} \cdots a_j$. In particular, a non-terminal node spans a_j if and only if the label on the node is a_j. A non-terminal node spans $a_{i+1} \cdots a_j$ if and only if $a_{i+1} \cdots a_j$ is the concatenation in left to right order of strings spanned by the children of the node.

- If η does not subsume the foot node of an auxiliary tree then: η covers the string $a_{i+1} \cdots a_j$ if and only if it spans $a_{i+1} \cdots a_j$.
- If η is on the spine of a right recursive auxiliary tree T then: η covers $a_{i+1} \cdots a_j$ if and only if η spans some string that is the concatenation of $a_{i+1} \cdots a_j$ and a string spanned by the foot of T. (This situation is illustrated by the right drawing in Figure 12, in which η is labeled with B.)
- If η is on the spine of a left recursive auxiliary tree T then: η covers $a_{i+1} \cdots a_j$ if and only if η spans some string that is the concatenation of a string spanned by the foot of T and $a_{i+1} \cdots a_j$. (This situation is illustrated by the left drawing in Figure 12, in which η is labeled with B.)

[1] It can be easily shown that by adding new nodes any LCFG can be transformed into an equivalent LCFG satisfying this condition.

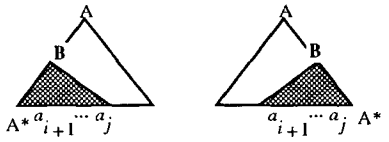


Figure 12: Coverage of nodes on the spine.

The algorithm stores pairs of the form $\langle \eta, pos \rangle$ in an n by n array C. In a pair, pos is either t (for top) or b (for bottom). For every node η in every elementary tree in G, the algorithm guarantees the following.

- $\langle \eta, b \rangle \in C[i,j]$ if and only if η covers $a_{i+1} \cdots a_j$.
- $\langle \eta, t \rangle \in C[i,j]$ if and only if $\langle \eta, b \rangle \in C[i,j]$ or $a_{i+1} \cdots a_j$ is the concatenation (in either order) of a string covered by η and a string covered by an auxiliary tree that can be adjoined on η.

The algorithm fills the upper diagonal portion of the array $C[i,j]$ $(0 \leq i \leq j \leq n)$ for increasing values of $j - i$. The process starts by placing each foot node in every cell $C[i,i]$ and each terminal node η in every cell $C[i, i+1]$ where η is labeled a_{i+1}.

The algorithm then considers all possible ways of combining covers into longer covers. In particular, it fills the cells $C[i, i+k]$ for increasing values of k by combining elements from the cells $C[i,j]$ and $C[j, i+k]$ for all j such that $i < j < i+k$. There are three situations where combination is possible: sibling concatenation, left recursive concatenation, and right recursive concatenation.

Sibling concatenation is illustrated in Figure 13. Suppose that there is a node η_0 (labeled B) with two children η_1 (labeled A) and η_2 (labeled A'). If $\langle \eta_1, t \rangle \in C[i,j]$ and $\langle \eta_2, t \rangle \in C[j, i+k]$ then $\langle \eta_0, b \rangle \in C[i, i+k]$.

Left recursive concatenation is illustrated in Figure 14. Here, the cover of a node is combined with the cover of a left auxiliary tree that can be adjoined at the node. Right recursive concatenation, which is shown in Figure 15 is analogous.

For simplicity, the recognizer is written in two parts. A main procedure and a subprocedure Add$(node, pos, i, j)$, which adds the pair $\langle node, pos \rangle$ into $C[i,j]$.

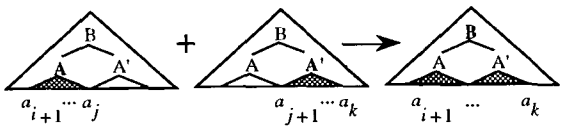


Figure 13: Sibling concatenation.

Procedure **recognizer**
begin

;; foot node initialization ($C[i,i]$)
for $i = 0$ to n
  for all foot node η in A call
    Add(η, b, i, i)

;; terminal node initialization ($C[i,i+1]$)
for $i = 0$ to $n-1$
  for all node η in $A \cup I$ labeled by a_{i+1}
    call Add($\eta, t, i, i+1$)

;; induction ($C[i,i+k] = C[i,j] + C[j,i+k]$)
for $k = 2$ to n
  for $i = 0$ to $n-k$
    for $j = i+1$ to $i+k-1$

      ;; sibling concatenation
      if $\langle \eta_1, t \rangle \in C[i,j]$
      and $\langle \eta_2, t \rangle \in C[j,i+k]$
      and η_1 is the left sibling of η_2
        with common parent η_0
      then Add($\eta_0, b, i, i+k$)

      ;; left recursive concatenation
      if $\langle \eta, b \rangle \in C[i,j]$
      and $\langle \rho, t \rangle \in C[j,i+k]$
      and ρ is the root node of a left recursive
        auxiliary tree that can adjoin on η
      then Add($\eta, t, i, i+k$)

      ;; right recursive concatenation
      if $\langle \eta, b \rangle \in C[j,i+k]$
      and $\langle \rho, t \rangle \in C[i,j]$
      and ρ is the root node of a right recursive
        auxiliary tree that can adjoin on η
      then Add($\eta, t, i, i+k$)

If $\langle \eta, t \rangle \in C[0,n]$
and η is labeled by S
and η is the root node of an initial tree in I
  then return acceptance
  otherwise return rejection
end

Note that the sole purpose of the codes t and b is to insure that only one auxiliary tree can adjoin on a node. The procedure could easily be modified to account for other constraints on the way derivation should proceed such as those suggested for LTAGs (Schabes and Shieber, 1992).

The procedure *Add* puts a pair into the array C. If the pair is already present, nothing is done. However, if it is new, it is added to C and other pairs may be added as well. These correspond to cases where the coverage is not increased: when a node is the only child of its parent, when the

Figure 14: Left recursive concatenation.

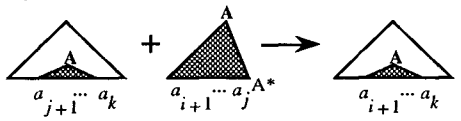


Figure 15: Right recursive concatenation.

node is recognized without adjunction, and when substitution occurs.

Procedure **Add**(η, pos, i, j)
begin

  Put $\langle \eta, pos \rangle$ in $C[i,j]$

  if $pos = t$ and η is the only child of a parent μ
    call Add(μ, b, i, j)

  if $pos = t$ and η is the root node of an
    initial tree, for each substitution node ρ
    at which η can substitute call Add(ρ, t, i, j)

  ;; no adjunction
  if $pos = b$
    if the node η does not have an OA constraint
      call Add(η, t, i, j)

end

The $O(n^3)$ complexity of the recognizer follows from the three nested induction loops on k, i and j. (Although the procedure *Add* is defined recursively, the number of pairs added to C is bounded by a constant that is independent of sentence length.)

By recording how each pair was introduced in each cell of the array C, one can easily extend the recognizer to produce all derivations of the input.

Conclusion

LCFG combines much of the power of LTAG with the computational efficiency of CFG. It supports most of the same linguistic analysis supported by LTAG. In particular, most of the current LTAG for English falls into LCFG. In addition, LCFG can lexicalize CFG without altering the trees produced. Finally, LCFG can be parsed in $O(n^3)$-time.

There are many directions in which the work on LCFG described here could be extended. In

particular, one could consider stochastic extensions, LR parsing, and non-deterministic LR parsing.

Acknowledgments

We thank John Coleman who, by questioning whether the context-sensitivity of stochastic LTAG was actually being used for English, triggered this work. We thank Aravind Joshi, Fernando Pereira, Stuart Shieber and B. Srinivas for valuable discussions.

REFERENCES

Abeillé, Anne, Kathleen M. Bishop, Sharon Cote, and Yves Schabes. 1990. A lexicalized tree adjoining grammar for English. Technical Report MS-CIS-90-24, Department of Computer and Information Science, University of Pennsylvania.

Joshi, Aravind K. and Yves Schabes. 1992. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages.* Elsevier Science.

Joshi, Aravind K., K. Vijay-Shanker, and David Weir. 1991. The convergence of mildly context-sensitive grammatical formalisms. In Peter Sells, Stuart Shieber, and Tom Wasow, editors, *Foundational Issues in Natural Language Processing.* MIT Press, Cambridge MA.

Lang, Bernard. 1990. The systematic constructions of Earley parsers: Application to the production of $O(n^6)$ Earley parsers for Tree Adjoining Grammars. In *Proceedings of the 1st International Workshop on Tree Adjoining Grammars*, Dagstuhl Castle, FRG, August.

Schabes, Yves, Anne Abeillé, and Aravind K. Joshi. 1988. Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, August.

Schabes, Yves and Aravind K. Joshi. 1990. Parsing with lexicalized tree adjoining grammar. In Masaru Tomita, editor, *Current Issues in Parsing Technologies.* Kluwer Accademic Publishers.

Schabes, Yves and Stuart Shieber. 1992. An alternative conception of tree-adjoining derivation. In 20^{th} *Meeting of the Association for Computational Linguistics (ACL'92).*

Schabes, Yves and Richard C. Waters. 1993. Lexicalized context-free grammar: A cubic-time parsable formalism that strongly lexicalizes context-free grammar. Technical Report 93-04, Mitsubishi Electric Research Laboratories, 201 Broadway. Cambridge MA 02139.

Schabes, Yves. 1991. The valid prefix property and left to right parsing of tree-adjoining grammar. In *Proceedings of the second International Workshop on Parsing Technologies*, Cancun, Mexico, February.

Vijay-Shanker, K. and David Weir. 1993a. The equivalence of four extensions of context-free grammars. *To appear in Mathematical Systems Theory.*

Vijay-Shanker, K. and David Weir. 1993b. Parsing some constrained grammar formalisms. *To appear in Computational Linguistics.*

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

Weir, David J. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.