MITSUBISHI ELECTRIC RESEARCH LABORATORIES
CAMBRIDGE RESEARCH CENTER

Technical Report 93-04 June 1993

Lexicalized Context-Free Grammar:
A Cubic-Time Parsable, Lexicalized Normal Form For Context-Free Grammar That Preserves Tree Structure

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

Abstract

Lexicalized context-free grammar (LCFG) is a tree-based formalism that makes use of both tree substitution and a restricted form of tree adjunction. Because of its use of adjunction, LCFG allows sufficient freedom in the way derivations can be performed that lexicalization of context-free grammars (CFGs) is possible while preserving the structure of the trees derived by the CFGs. However, the tree adjunction permitted is sufficiently restricted that LCFGs are string-wise equivalent to CFGs and have the same cubic-time worst-case complexity bounds for recognition and parsing.

Submitted to Computational Linguistics

Publication History:–

1. First printing, MTR 93-04 June, 1993

Contents

1 Introduction

Context-free grammar (CFG) has been a well accepted framework for computational linguistics for a long time. While it has drawbacks, it has the virtue of being computationally efficient, $O(n^3)$-time in the worst case for a sentence of length n. In addition, recognition and parsing algorithms exist that, while being no better than $O(n^3)$-time in the worst case, are very much better than this in the typical case.

A *lexicalized* grammar is one where every rule contains an explicit lexical item (e.g., a word in the lexicon). A grammar formalism $\mathcal{F}$ is said to *lexicalized* by another formalism $\mathcal{F}'$ if every grammar G written in $\mathcal{F}$ can be converted into an equivalent lexicalized grammar G' in $\mathcal{F}'$.

Lexicalization is of interest from a computational perspective, because other things being equal, lexicalized grammars can often be parsed much more efficiently than non-lexicalized ones. Lexicalization 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.

In general, CFGs are not lexicalized since they allow rules such as $S \rightarrow NP\ VP$ that do not locally introduce lexical items. In contrast, the well-known Greibach Normal Form (GNF) for CFG is lexicalized, because the first item on the right-hand side of every rule is required to be a lexical item (e.g., $S \rightarrow \text{John}\ VP$).

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 strings [3]. For most grammars G, strings can be recognized much more quickly using G' than G, using left-to-right bottom-up algorithms.

Unfortunately, for most grammars G, it is not possible for the set of trees produced by G' to be the same as the set of trees produced by G. As a result, parsing with G' does not produce the same result as parsing with G.

Therefore, while GNF lexicalizes CFGs, this is only a *weak* lexicalization in the sense that while it preserves the sets of strings produced by the original grammar, it does not preserve the set of trees produced. Of more interest is *strong* lexicalization that preserves tree sets as well as string sets.

In the following, we use the term lexicalization only its strong sense. Note that in this strong sense, GNF does not lexicalize CFG. One can go further than this and say that CFG does not lexicalize CFG. The fundamental reason for this is that CFG specifies a one-to-one mapping from derivations to trees. Changing a rule that participates in any derivations changes the tree set. To achieve the lexicalization of CFGs one must switch to a formalism with greater *derivational freedom*, i.e., one where it is possible to modify a grammar in interesting ways without changing the tree set.

An interesting example of a formalism with greater derivational freedom than CFG is tree substitution grammar (TSG) [4]. In TSG, the combining operation is substitution just as in CFG, however, the items to be combined are extended to multi-level trees (called initial trees) instead of just one-level rules. Substitution combines two trees by replacing a leaf in one tree with another tree. A TSG G can be altered by combining initial trees with substitution to create larger initial trees without changing the trees produced.

The extra derivational freedom provided by TSG makes it possible to lexicalize many CFGs. However, as shown by Schabes and Joshi [7, 4], TSG cannot be used to lexicalize every CFG.

Derivational freedom can be increased further by using the context-sensitive operation of adjunction. Adjunction combines two trees by expanding an internal node in one tree by inserting another tree. This opens up many more possibilities for altering a grammar without changing the trees produced and makes it possible to lexicalize CFG. In particular, Schabes and Joshi [7, 4] have shown that lexicalized tree adjoining grammar (LTAG) lexicalizes CFG in the strong sense defined above.

Unfortunately, context-sensitive operations entail much larger computation costs for parsing and recognition than CFGs. In particular, the fastest known LTAG parser requires $O(n^6)$-time in the worst case in contrast to $O(n^3)$ for CFG. As a result, there are no computational advantages to lexicalizing a grammar using LTAG because the speedup due to the grammar becoming lexicalized is swamped by the dramatic increase in fundamental worst-case cost.

Heretofore, every method for lexicalizing CFGs in the strong sense defined above has required context-sensitive operations [4]. As a result, every method for lexicalizing CFGs has shared with LTAG the unfortunate feature that lexicalization leads to dramatically decreased rather than increased computational performance.

The primary contribution of the work presented here is a grammar formalism called lexicalized context-free grammar (LCFG) that lexicalizes CFG while retaining $O(n^3)$-time worst-case computation bounds. This formalism allows the inherent computational advantages of lexicalization to be realized.

Like LTAG, LCFG allows adjunction. However, LCFG only allows a very restricted form of adjunction that is context-free in nature. As a result, LCFGs are only capable of generating context-free languages and are subject to the same worst case computation bounds as CFGs.

The second contribution of this work is an easily automatable procedure for converting any CFG G into an equivalent LCFG G'. The final contribution of this work is a highly efficient Earley-style parser for LCFG that maintains the valid prefix property. At least for many grammars G, it should be possible to use this parser to parse strings using G' significantly faster than can be done by a parser working directly with G.

The following sections: present LCFG, discuss lexicalization in more detail, prove that LCFG lexicalizes CFG by showing a constructive lexicalization procedure, prove that LCFG can be parsed in $O(n^3)$-time by exhibiting a simple $O(n^3)$-time parser, and describe a highly efficient parser for LCFG.

2 Lexicalized Context-Free Grammar

Lexicalized context-free grammar (LCFG) [11] is a tree generating system that is a restricted form of lexicalized tree-adjoining grammar (LTAG) [7, 4]. Informally speaking, the grammar 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 because every initial and auxiliary tree is required to contain a terminal symbol on its frontier.

Definition 1 (LCFG) 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 finite sets of finite trees labeled by terminal and non-terminal symbols, and S is a distinguished non-terminal start symbol. The set $I \cup A$ is referred to as the elementary trees.

The interior nodes in each elementary tree are labeled by non-terminal symbols. The nodes on the frontier of each elementary tree are labeled with terminal symbols, non-terminal symbols, and the empty string (ε). At least one frontier node is labeled with a terminal symbol. With the possible exception of one (see below), the non-terminal symbols on the frontier are marked for substitution. (By convention, substitutability is indicated in diagrams by using a down arrow ($\downarrow$).) Frontier nodes labeled with ε are referred to as empty.

The difference between auxiliary trees and initial trees is that each auxiliary tree has exactly one non-terminal frontier node that is marked as the foot. The foot must be labeled with the same label as the root. (By convention, the foot of an auxiliary tree is indicated in diagrams by using an asterisk ($*$).) The path from the root of an auxiliary tree to the foot is called the *spine*.

Auxiliary trees in which every non-empty frontier node is to the left of the foot are called *left* auxiliary trees. Similarly, auxiliary trees in which every non-empty frontier node is to the right of the foot are called *right* auxiliary trees. Other auxiliary trees are called *wrapping* auxiliary trees.[1]

LCFG does not allow adjunction to apply to foot nodes or nodes marked for substitution. LCFG allows the adjunction of a left auxiliary tree and a right auxiliary tree on the same node. However, LCFG does not allow the adjunction of either two left or two right auxiliary trees on the same node.

Crucially, LCFG does not allow wrapping auxiliary trees. It does not allow elementary wrapping auxiliary trees, and it does not allow the adjunction of two auxiliary trees, if the result would be a wrapping auxiliary tree.

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.

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.

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 T into another tree at an interior node η that has the same label as the root (and therefore foot) of T. In particular, η is replaced by a copy of T and the foot of the copy of T is replaced by the subtree rooted at η. The adjunction of a left auxiliary tree is referred to as left adjunction (see Figure 3). The adjunction of a right auxiliary tree is referred to as right adjunction (see Figure 4).

[1]In [11] these three kinds of auxiliary trees are referred to differently as right recursive, left recursive, and centrally recursive, respectively.

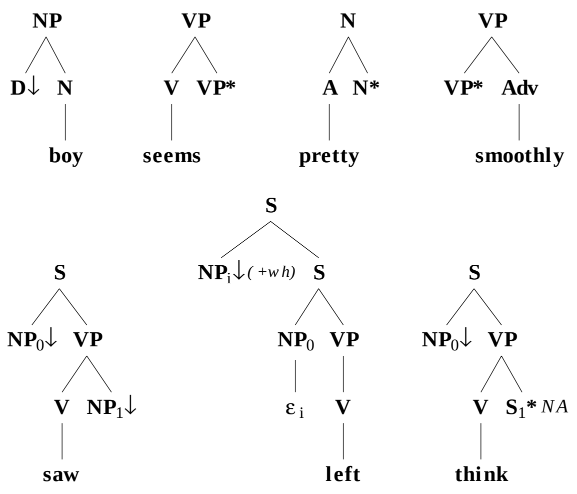


Figure 1: Example LCFG trees.

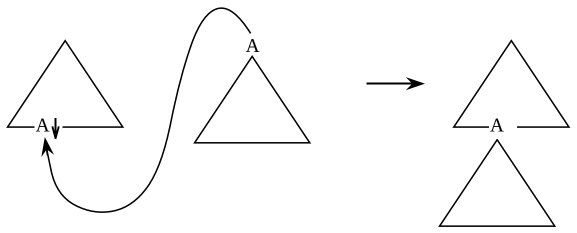


Figure 2: Substitution.

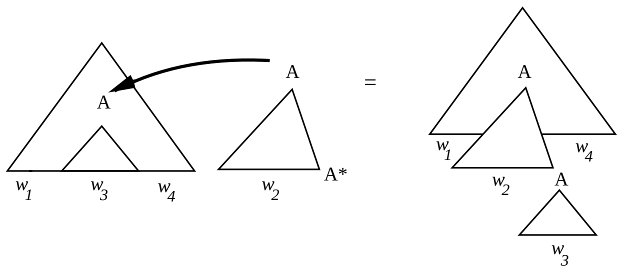


Figure 3: Left adjunction.

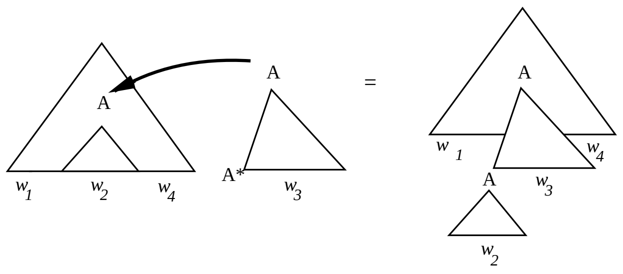


Figure 4: Right adjunction.

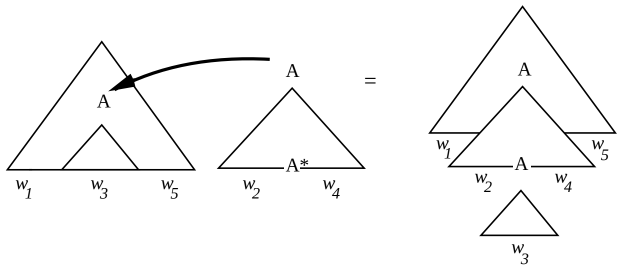


Figure 5: Wrapping adjunction.

LCFG's prohibition on wrapping auxiliary trees can be rephrased solely in terms of elementary trees. To start with, there must be no elementary wrapping auxiliary trees. In addition, an elementary left (right) auxiliary tree cannot be adjoined on any node that is on the spine of an elementary right (left) auxiliary tree. Further, no adjunction whatever is permitted on a node η that is to the right (left) of the spine of an elementary left (right) auxiliary tree T. (Note that for T to be a left (right) auxiliary tree, every frontier node subsumed by η must be labeled with ε.)

Tree adjoining grammar formalisms typically forbid adjunction on foot nodes and substitution nodes. In addition, they typically forbid multiple adjunctions on a node. However, in the case of LCFG, it is convenient to relax this latter restriction slightly by allowing right and left adjunction on a node, but at most once each. (Due to the other restrictions placed on LCFG, this relaxation increases the trees that can be generated without increasing the ambiguity of derivations.)

2.1 Comparison of LCFG With Other Formalisms

The only important difference between LCFG and LTAG is that LTAG allows both elementary and derived wrapping auxiliary trees. The importance of this is that wrapping adjunction (see Figure 5) encodes string wrapping and is therefore context sensitive in nature. In contrast, left and right adjunction (see Figures 3 & 4) merely support string concatenation. As a result, while LTAG is context sensitive in nature, LCFG is limited to generating only context-free languages.

To see that LCFGs can only generate context free languages, consider that any LCFG G can be converted into a CFG generating the same strings in two steps as follows. First, G is converted in to a TSG G' that generates the same strings. Then, this TSG is converted into a CFG G''.

A TSG is the same as an LCFG (or LTAG) except that there cannot be any auxiliary trees. To create G' first make every initial tree of G be an initial tree of G'. Next, make every auxiliary tree T of G be an initial tree of G'. When doing this, relabel the foot of T with ε (turning T into an initial tree). In addition, let A be the label of the root of T. If T is a left auxiliary tree, rename the root to A_L; otherwise rename it to A_R.

To complete the creation of G' alter every node η in every elementary tree in G' as follows: Let A be the label of η. If left adjunction is possible at η add a new first child of η labeled A_L, mark it for substitution, and add a new rule $A_L \rightarrow \varepsilon$. If right adjunction is possible at η add a new last child of η labeled A_R, mark it for substitution, and add a new rule $A_R \rightarrow \varepsilon$.

The TSG G' generates the same strings as G, because all cases of adjunction have been changed into equivalent substitutions. Note that the transformation would not work if LCFG allowed wrapping auxiliary trees. The TSG G' can be converted into a CFG G'' by renaming

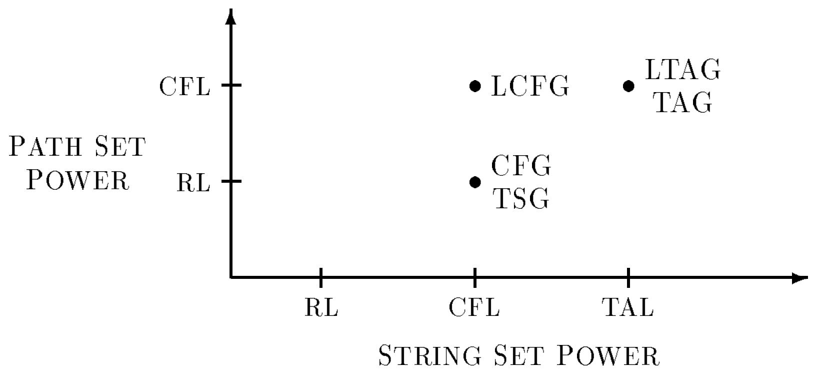


Figure 6: The tree and string complexity of LCFG and several other formalisms

internal nodes so that they are unique and then decomposing the trees into one-level rules.

Although the string sets generated by LCFG are the same as those generated by CFG, LCFG is capable of generating more complex sets of trees than CFGs. In particular, it is interesting to look at the path sets of the trees generated. (The path set of a grammar is the set of all paths from root to frontier in the trees generated by the grammar. The path set is a set of strings over $\Sigma \cup NT \cup \{\varepsilon\}$.)

The path sets for CFG (and TSG) are regular languages [14]. In contrast, just as for LTAGS and TAGS, the path sets for LCFGs are context-free languages. To see this, consider that adjunction makes it possible to embed a sequence of nodes (the spine of the auxiliary tree) in place of a node on a path. Therefore, from the perspective of the path set, auxiliary trees are analogous to context-free productions.

Figure 6 summarizes the relationship between LCFG and several other grammar formalisms. The horizontal axis shows the complexity of strings that can be generated by the formalisms, i.e., regular languages (RL), context-free languages (CFL), and tree adjoining languages (TAL). The vertical axis shows the complexity of the path sets that can be generated.

CFG (and TSG) create context-free languages, but the path sets they create are regular languages. LTAG and TAG generate tree adjoining languages and have path sets that are context-free languages. LCFG is intermediate in nature. It can only generate context-free languages, but has path sets that are also context-free languages.

3 Lexicalization

The question of whether a grammar is lexicalized is relevant no matter what the grammar is stated in terms of. The key concept is that each structure (e.g., production rule, elementary tree) in the grammar contains a lexical item that is realized. More precisely, a *lexicalized grammar* [9, 7] can be defined as follows:

Definition 2 (Lexicalized Grammars) A *lexicalized* grammar consists of a finite set of elementary structures of finite size, each of which contains an overt (i.e., non-empty) lexical item, and a finite set of operations for creating derived structures.

The overt lexical item in an elementary structure is referred to as its anchor. If more than one lexical item appears, either just one lexical item is designated as the anchor or a subset of the lexical items local to the structure are designated as a *multi-component anchor*.

Note that Categorial Grammars [5, 13] are lexicalized according to the definition above since each basic category has a lexical item associated with it.

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. This facilitates efficient parsing by, in effect, allowing the grammar to be dynamically restricted to only those rules that are relevant to the words that actually appear in a given sentence.

If each elementary structure contains a lexical item as its leftmost non-empty constituent, the grammar is said to be *left anchored*. This facilitates efficient left to right parsing.

Definition 3 (lexicalization [4]) A formalism F is *lexicalized* by another formalism F', if for every finitely ambiguous grammar G in F that does not derive the empty string, there is a lexicalized grammar G' in F' such that G and G' generate the same tree set (and therefore the same string set). A *lexicalization procedure* is a procedure that generates G' from G.

The restrictions on the form of G are motivated by two key properties of lexicalized grammars [4]. Lexicalized grammars cannot derive the empty string, because every structure introduces at least one lexical item. Thus, if a grammar is to be lexicalized, it must not be the case that $S \overset{*}{\Rightarrow} \varepsilon$. Lexicalized grammars are finitely ambiguous, because every rule introduces at least one lexical item into the resulting string. Thus, if a grammar is to be lexicalized, it must be only finitely ambiguous. In the case of a CFG, this means that the grammar cannot contain either elementary or derived recursive chain rules $X \overset{*}{\Rightarrow} X$.

As a focal point for discussion, consider the CFG in Figure 7. The language generated by this grammar is $a^{\star}$. The tree set has the property that for any integer i it includes a tree where the shortest path from the root to any leaf is longer than i. In particular, in addition to many other trees, the CFG in Figure 7 generates a balanced tree of the form shown on the right of the figure with minimum path length $i+1$ for the string a^{2^i}. As argued in [4], due to the unbounded nature of the trees produced, neither GNF, CFG, nor TSG is capable of lexicalizing the grammar in Figure 7.

3.1 GNF Does Not lexicalize CFG

Greibach normal form (GNF) is a restricted form of CFG where every production rule is required to be of the form $A \rightarrow a\alpha$ (where A is a non-terminal symbol, a is a terminal symbol, and α is a possibly empty string of non-terminal symbols). Every GNF is lexicalized, because every

$$S \rightarrow S\ S$$
$$S \rightarrow a$$

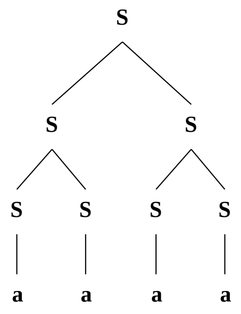


Figure 7: An example CFG and a tree it derives.

$$S \rightarrow a\ S$$
$$S \rightarrow a$$

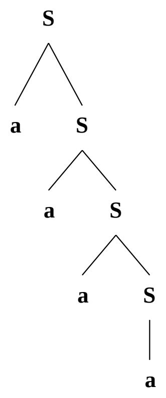


Figure 8: A GNF corresponding to Figure 7 and a tree it derives.

rule locally introduces a lexical item (a), and like any CFG, GNF is required to have a finite number of finite rules.

The CFG in Figure 7 can be put into GNF as shown in Figure 8. However, this GNF is not a strong lexicalization of the grammar in Figure 7, because while the two grammars generate the same language, they do not generate the same trees. In particular, the GNF shown only generates right linear trees like the one shown on the right of Figure 8.

No GNF grammar can produce the same tree set as the CFG in Figure 7, because no matter what the GNF grammar looks like, it is guaranteed that the first rule applied in any derivation will have a terminal symbol as its leftmost constituent. As a result, the length of the leftmost path in every tree derived by any GNF grammar is one.

3.2 CFG Does Not Lexicalize CFG

The same basic argument that can be used to show that GNF does not lexicalize CFG can be used to show that CFG does not lexicalize CFG. For a CFG to be lexicalized, every rule must contain at least one terminal on its right-hand side. As a result, every generated tree must contain at least one path of length one. Therefore, no lexicalized CFG can generate the tree shown in Figure 7.

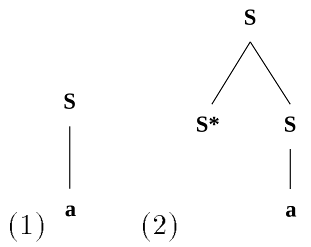


Figure 9: An LCFG that lexicalizes the CFG in Figure 7.

3.3 TSG Does Not Lexicalize CFG

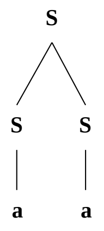
To see that no TSG lexicalizes the CFG in Figure 7 while preserving the trees generated, consider the following. For a TSG G to be lexicalized, it must consist of a finite number of finite initial trees and each tree must contain a lexical item. For each of these trees consider the length of the shortest path from the root to a non-terminal. Let i be the maximum of these shortest lengths. The topmost portion of any tree T derived by G is one of the initial trees of G. Therefore, the shortest path from the root to a non-terminal in T must be no longer than i. This is incompatible with the fact that there is no bound on the shortest path from the root to a terminal in the trees generated by the grammar in Figure 7.

3.4 LCFG Lexicalizes CFG

As argued in the next section, LCFG does lexicalize CFG. However, before looking at this in detail, it is instructive to consider how adjunction makes it possible to lexicalize the grammar in Figure 7.

The CFG in Figure 7 is lexicalized by the LCFG in Figure 9. This LCFG consists of one initial tree (1) and one right auxiliary tree (2).

Every derivation using the grammar in Figure 9 begins by instantiating the initial tree (1). This produces the only possible tree whose fringe is a. One can then continue by adjoining the auxiliary tree (2) on the root of the initial tree (1) to produce the only possible tree whose fringe is aa.



Before adjoining the auxiliary tree (2) onto the root of the initial tree (1), one could adjoin the auxiliary tree (2) onto itself at either the root or its right child. These two adjunctions result in the two derived auxiliary trees shown in Figure 10.

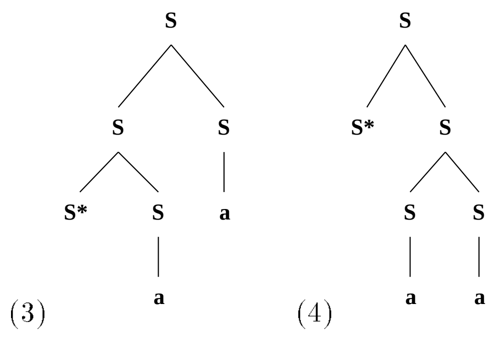
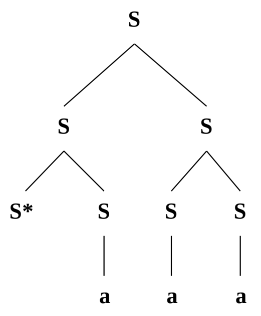


Figure 10: Some derived auxiliary trees.

Adjoining the derived auxiliary trees (3 & 4) on the initial tree (1) results in the two trees that have *aaa* as their fringe. By adjoining the elementary auxiliary tree (2) a second time on the derived trees (3 & 4) and so on, one can create every possible tree whose fringe is in $a^\star$. In particular, adjoining the elementary auxiliary tree (2) to the right child of the root of the the derived auxiliary tree (3) creates the following:



If this tree is adjoined on the initial tree (1), the balanced tree in Figure 7 results. The key thing to notice about the derived auxiliary tree above is that the minimum distance from the root to a non-terminal is 3—greater than the minimum distance to a non-terminal in either of the elementary trees.

In general, the reason that adjunction allows lexicalization, is that starting with initial trees of bounded size, it is possible to generate trees of unbounded size, while at every step introducing non-terminals. This can be done, because the path from the root of a derived tree to a non-terminal can be altered after it is first introduced.

4 LCFG Lexicalizes CFG

In the following, we give a constructive proof of the fact that LCFG lexicalizes CFG.

Theorem If $G = (\Sigma, NT, P, S)$ is a finitely ambiguous CFG that does not generate the empty string, then there is an LCFG $G_L = (\Sigma, NT, I, A, S)$ generating the same language and tree set as g. Furthermore, G_L can be chosen so that all the auxiliary trees are right auxiliary trees.

As usual, 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 finite set of finite 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_L by constructing a directed graph LG that we call the *lexicalization graph.* Paths in LG correspond to leftmost paths from root to frontier in (partial) derivation trees rooted at non-terminal symbols in G.

> LG 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 LG labeled with X. For each rule $X \rightarrow Y\alpha$ in G ($Y \in \Sigma \cup NT$) create a directed arc labeled with $X \rightarrow Y\alpha$ from the node in LG labeled with X to the node labeled Y.

Consider the example CFG in Figure 11. For this grammar, Step 1 creates the LG shown in Figure 12.

The significance of LG is that there is a one-to-one correspondence between paths in LG ending on a terminal and left-corner derivations in G. A left-corner derivation in a CFG is a (partial) derivation starting from a non-terminal where every expanded node (other than the root) is the leftmost child of its parent and the left corner is a 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 LG. Therefore, every path in LG ending on a terminal corresponds to a left-corner derivation in G and vice versa.

$$
\begin{array}{l}
S \rightarrow A\ A \\
S \rightarrow B\ A \\
A \rightarrow B\ B \\
B \rightarrow A\ S \\
B \rightarrow b
\end{array}
$$

Figure 11: An example grammar.

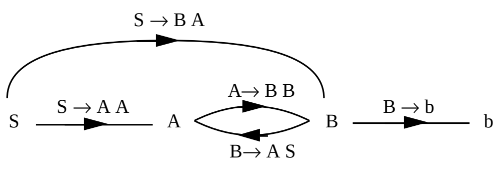

$$\begin{tikzcd}
S \arrow[r, "S \rightarrow A\,A"] \arrow[rr, bend left=40, "S \rightarrow B\,A"] & A \arrow[r, bend left, "A \rightarrow B\,B"] & B \arrow[l, bend left, "B \rightarrow A\,S"] \arrow[r, "B \rightarrow b"] & b
\end{tikzcd}$$

Figure 12: The LG created by Step 1.

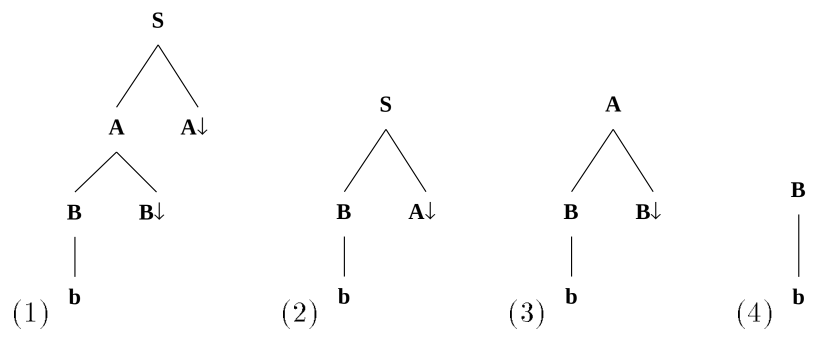


Figure 13: Initial trees created by Step 2.

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

> For each acyclic path in LG 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 mark them as substitution nodes.

Given the example grammar in Figure 11 and its corresponding LG in Figure 12, this step produces the initial trees shown in Figure 13.

Each initial tree created is lexicalized, because each one has a terminal symbol as the leftmost element of its frontier. (The last rule in each of the selected paths is a rule where the leftmost element on the right-hand side is a terminal.)

A finite set of finite initial trees is created, because a finite set of finite paths is selected. (Any directed graph containing a finite number of nodes and arcs has only a finite number of acyclic paths, each of which has a finite length.)

Most importantly, the set of initial trees created is the set of non-recursive left-corner derivations in G. (There is a one-to-one correspondence between paths in LG and left-corner derivations, and we have chosen every non-recursive path.)

Step 3 This step constructs a set of right auxiliary trees corresponding to the cyclic path segments in LG. In particular, an auxiliary tree is created corresponding to each minimal cycle in LG. (A minimal cycle is a path that does not visit any node twice, except for the node it starts and ends on.)

> For each minimal cycle in LG from a node X to X, 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 mark them all as substitution nodes.

The LG in Figure 12 has two minimal cyclic paths (one from A to A via B and one from B to B via A). For these cycles, Step 3 creates the two auxiliary trees shown in Figure 14, one for A and one for B.

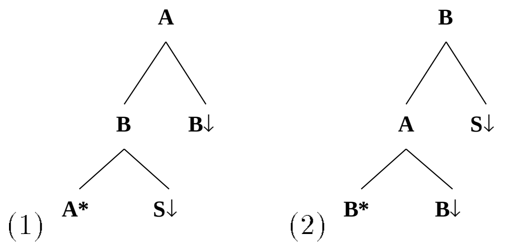


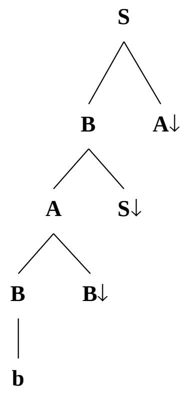
Figure 14: Auxiliary trees created by Step 3.

As in Step 2, a finite set of finite trees is created, because a finite set of finite paths is selected. (A finite graph can have only finitely many minimal cycles, each of which has finite length.) However unlike Step 2, the auxiliary trees created in Step 3 are not necessarily lexicalized.

Consider the set of trees that can be created by adjoining auxiliary trees from Step 3 with each other and the initial trees from Step 2. This resulting set of trees is the set of every left-corner derivation in G.

To see this, consider that every path in LG can be represented as an initial acyclic path with zero or more minimal cycles inserted into it. Given the one-to-one correspondence between paths in LG and left-corner derivations, this implies that any left-corner derivation can be generated by adjoining auxiliary trees corresponding to minimal cycles into initial trees corresponding to acyclic paths ending on terminals.

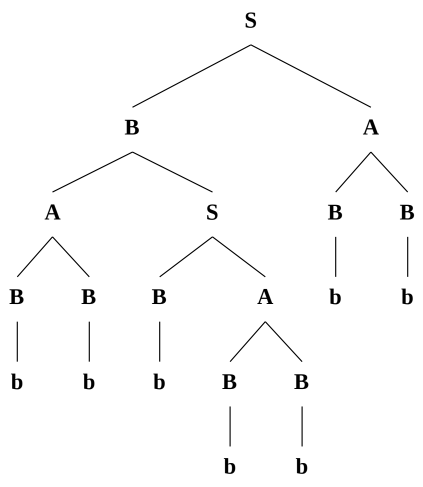
For instance, the path S, B, A, B, b in Figure 12 is the acyclic path S, B, b with the minimal cycle B, A, B inserted into it. The corresponding left-corner derivation in G (shown below) is the second initial tree in Figure 13 with the second auxiliary tree in Figure 14 adjoined on the node labeled B.



Consider 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 adjunction and substitution. This resulting set of trees is the tree set of G.

To see this, consider that every derivation in G can be decomposed into left-corner derivations in G that are combined with substitution and vice versa. In particular, every terminal node that is the leftmost child of its parent is the left-corner of a separate left-corner derivation.

For instance, the derived tree shown below is composed of 7 left-corner derivations. From left to right along the fringe, these are the second initial tree with the second auxiliary tree adjoined into it, and the fourth, second, third, fourth, third, and fourth initial trees respectively.



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 η just to the right of the foot. If this node is a terminal do nothing. Otherwise, remove T from the set of auxiliary trees. Then, add every auxiliary tree that can be constructed by substituting a compatible initial tree from Step 2 for η in T.

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, T would correspond to a derivation $X\overset{*}{\Rightarrow}X$ in G and G would be infinitely ambiguous.

In the case of our continuing example, Step 4 results in the auxiliary trees in Figure 15.

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.

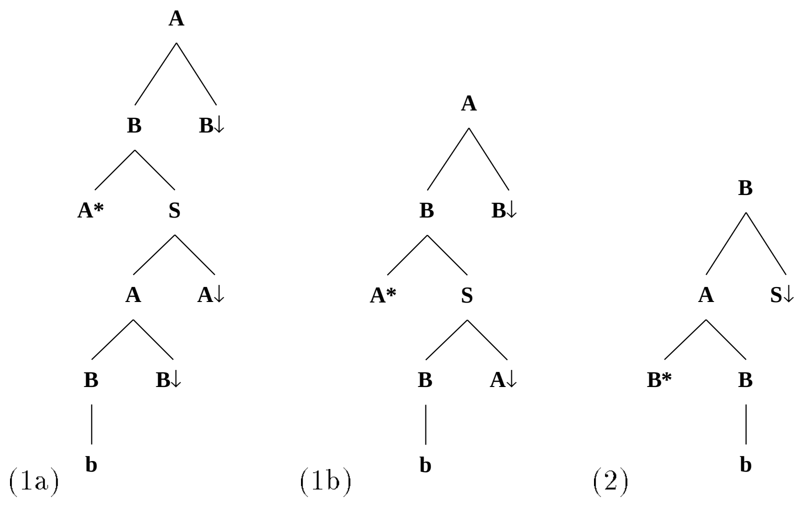


Figure 15: Auxiliary trees created by Step 4.

The change in the auxiliary trees caused by Step 4 does not alter the set of trees that can be produced. The only action of Step 4 was to eliminate some nodes where substitution was previously possible. Whenever this was done, new trees were created corresponding to every substitution that could ever be done at these nodes.

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.

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 auxiliary assumption that G does not contain empty rules. This assumption can be dispensed with in two steps as follows.

First, any finitely ambiguous CFG that does not derive the empty string can be converted into a TSG that generates the same trees and does not contain any empty trees. (An *empty tree* is a tree where every element of the frontier is labeled with ε.) Second, the lexicalization procedure above can straightforwardly be extended to lexicalize any finitely ambiguous TSG that does not generate the empty string and does not contain any empty trees.

Suppose G contains one or more empty rules. One first constructs an equivalent TSG G', by converting every rule $A \rightarrow \alpha$ into a one-level tree. In this tree, the root is labeled A and the elements of α become the labels of its children. Any non-terminal children are marked for substitution. This conversion does not change the trees that can be generated in any way. There is at least one empty tree in G'.

The empty trees in G' are eliminated by substituting them wherever possible as follows. Suppose T is an empty tree in G' whose root is labeled A. (Note that since G does not derive the empty string, the root of T is not labeled S.) For every substitutable node η in G' labeled A: Make a copy of the tree containing η with T substituted for η. If there is no tree other than T that can be substituted for η, discard the original tree containing η. Note that this step might generate additional empty trees, but it cannot generate a new empty tree whose root is labeled A. (If it did, that would mean that there was previously a tree whose root is labeled A and all the non-empty frontier elements are labeled A. Given that there is also an empty tree whose root is labeled A, this implies that G is infinitely ambiguous in the way it derives the empty trees.) The process above is iterated until no empty trees remain.

The grammar G'' which results from the elimination of the empty trees in G' still generates exactly the same trees. The only change is that every empty subtree that can be created by G has been created and substituted everywhere it can appear.

With minor modifications, the lexicalization procedure above can now be used to lexicalize G''. The key change is to modify LG to take account of trees instead of rules and trees whose left corner is ε. The restated algorithm is as follows.

> **Step 1'** LG contains a node for every terminal and non-terminal symbol and an arc for every initial tree as follows. For each terminal and non-terminal symbol X in the grammar create a node in LG labeled with X. For each initial tree T, let X be the label of the root of T and Y ($Y \in \Sigma \cup NT$) be the label of the leftmost frontier node that is not labeled with ε. Create a directed arc labeled with T from the node in LG labeled with X to the node labeled Y.

There is a one-to-one correspondence between the modified LG and *leftmost-nonempty derivations*. A leftmost-nonempty derivation in a CFG is a (partial) derivation starting from a non-terminal where every expanded node (other than the root and nodes that head empty subtrees) is the leftmost child of its parent that does not head an empty subtree; and the leftmost non-empty node on the frontier is a terminal. With left-corner derivations replaced by leftmost-nonempty derivations, the rest of the procedure goes through as before.

> **Step 2'** For each acyclic path in LG from X to y, construct an initial tree T as follows. Start with the tree corresponding to the first arc in the path. Substitute the trees in the rest of the path one after another, always substituting for the leftmost non-empty node of T. While doing this, leave all the other substitution nodes in T unchanged.
>
> **Step 3'** For each minimal cycle in LG from a node X to X, construct an auxiliary tree T by starting with the tree corresponding to the first arc and repeatedly substituting for the leftmost non-empty frontier node using the trees in the rest of the path as in Step 2'. When all the trees in the path have been used, the leftmost non-empty frontier node in T will be labeled X. Change this from a substitution node to one marked as the foot of T. While doing the above, leave all the other substitution nodes in T unchanged.
>
> **Step 4'** For each auxiliary tree T built in step 3', consider the first non-empty frontier node η to the right of the foot. If this node is a terminal do nothing. Otherwise, remove T from the set of auxiliary trees. Then, add every auxiliary tree that can be constructed by substituting a compatible initial tree from Step 2' for η in T.

□

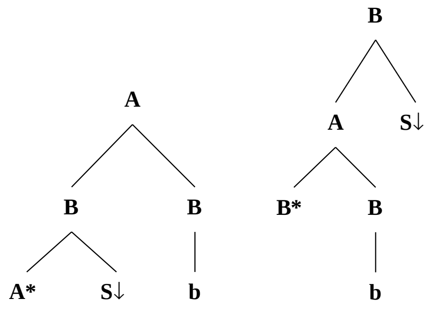
4.1 Efficiency Improvements To the Lexicalization Procedure

In the interest of simplifying the proof of the theorem, the presentation of the lexicalization procedure above was optimized for clarity, rather than for producing the best LCFG possible. When implementing the procedure steps should be made to minimize the size of the grammar produced and ensure that unnecessary ambiguity is not introduced into the grammar.

In the worst case, the number of initial and auxiliary trees in G_L can be very much greater than the number of production rules in G. In particular, the number of elementary trees in G_L is related to the number of acyclic and minimal cycle paths in LG. In the worst case, this number can rise very fast as a function of the number of arcs in LG (that is to say, as a function of the number of rules in P). (A fully connected graph of n^2 arcs between n nodes has $n!$ acyclic paths and $n!$ minimal cycles.)

Fortunately, a truly exponential explosion is unlikely in the typical case; and in the typical case, a lot can be done to reduce the number of elementary trees required for lexicalization.

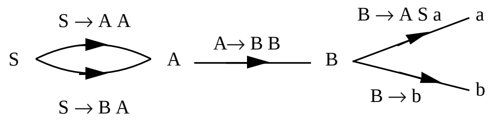
In particular, there are several places in the algorithm where alternative choices for how to proceed are possible. By picking the best of these choices, a smaller LCFG can be produced. For instance, when lexicalizing the auxiliary trees created in step 2, you need not do anything if there is any frontier node that is a terminal. Further, if a frontier node must be expanded, you can choose the node that corresponds to the smallest number of initial trees, rather than the leftmost. Using this approach, one can lexicalize the grammar in Figure 11 using just the two auxiliary trees shown below.



Alternatively, everywhere in the procedure, the word 'right' can be replaced by 'left' and vice versa. This results in the creation of a set of right anchored initial trees and left auxiliary trees. This can be of interest when the LG corresponding to right-corner derivations has fewer cycles than the LG corresponding to left-corner derivations. For instance, consider the following example:

$$\begin{array}{l} S \rightarrow A\ A \\ S \rightarrow B\ A \\ A \rightarrow B\ B \\ B \rightarrow A\ S\ a \\ B \rightarrow b \end{array}$$

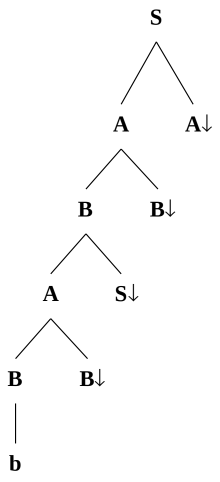
The LG corresponding to left-corner derivations for this grammar is the same as in Figure 12, however, the LG for right-corner derivations has no cycles:



Using the LG for right-corner derivations, the grammar above can be lexicalized using an LCFG that does not require any auxiliary trees and is therefore merely a TSG.

Going beyond this, one is not limited to using either all leftmost or all rightmost arcs in LG. When adding the arc for a rule $A \rightarrow \alpha$ one is free to connect the arc from A to any symbol (other than the empty string) in α. The only requirement is that every minimal cycle in the resulting LG must consist solely of either leftmost or rightmost derivation arcs and when when a minimal cycle is nested in another minimal cycle, the two cycles must have the same kind of arcs. Arcs that do not participate in any cycles do not need to be either leftmost or rightmost, because they will not contribute to any auxiliary trees. Using the full freedom of choice indicated above, one can often create an LG that has many fewer acyclic paths and minimal cycles than the LG corresponding to left- (or right-) corner derivations.

In the simple form presented above, the lexicalization procedure can result in an LCFG G_L that is ambiguous in the way it creates the derivation trees of G. This matters because it adds to the time required to parse with G_L. As an illustration of this problem, consider the following left-corner derivation in the CFG in Figure 11.



This can be created by starting from the first initial tree in Figure 13 and adjoining the first auxiliary tree in Figure 14 at A. However, it can also be created by starting from the first initial tree in Figure 13 and adjoining the second auxiliary tree in Figure 14 at B.

In general, the longer the minimal cycles in L, the greater ambiguity there will be in the way G_L derives trees. Fortunately, by introducing constraints on what auxiliary trees are allowed to adjoin on what nodes in which initial trees, it is possible to greatly reduce and often completely eliminate this ambiguity.

5 LCFG is Cubic-Time Parsable

Since LCFG is a restricted case of tree-adjoining grammar (TAG), standard $O(n^6)$-time TAG parsers [15, 8, 6] 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 proves that LCFG is parsable in $O(n^3)$-time by exhibiting a simple bottom-up recognizer for LCFG that is in the style of the CKY parser for CFG. The next section describes a much more efficient (and more elaborate) recognizer for LCFG that achieves the same worst case bounds.

5.1 Terminology

Suppose that $G = (\Sigma, NT, I, A, S)$ is an LCFG and that $a_1 \cdots a_n$ is an input string. 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).

Label(η) $\in \Sigma \cup NT \cup \varepsilon$ is the label of the node. The predicate IsInitialRoot(η) is true if and only if η is the root of an initial tree. Parent(η) is the node that is the parent of η or $\perp$ if η has no parent. FirstChild(η) is the node that is the leftmost child of η or $\perp$ if η has no children. Sibling(η) is the node that is the next child of the parent of η (in left to right order) or $\perp$ if there is no such node. The predicate Substitutable(ρ, η) is true if and only if η is marked for substitution and ρ is the root of an initial tree that can be substituted for η.

If ρ is the root of an auxiliary tree, then Foot(ρ) is the foot of the tree, otherwise Foot(ρ) $= \perp$. Conversely if ϕ is the foot of an auxiliary tree then Root(ϕ) is the root of the tree, otherwise Root(ϕ) $= \perp$. The predicate Adjoinable(ρ, η) is true if and only if ρ is the root of an elementary auxiliary tree that can adjoin on η. The predicate Radjoinable(ρ, η) is true if and only if ρ is the root of an elementary right auxiliary tree that can adjoin on η. The predicate Ladjoinable(ρ, η) is true if and only if ρ is the root of an elementary left auxiliary tree that can adjoin on η.

Figure 16 shows a tree generated by G. The bottommost node labeled A is an instance of a node η in some elementary tree T. The bottommost node labeled B is the corresponding instance of the root ρ of T. The middle node labeled A is an instance of the root η_L of a left auxiliary tree that was adjoined on η. The topmost node labeled A is an instance of the root η_R of a right auxiliary tree that was adjoined on η. Similarly, the topmost node labeled B is

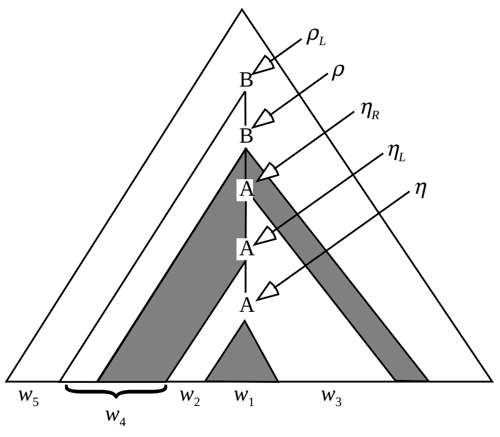


Figure 16: Segments of the fringe of a generated tree.

an instance of the root ρ_L of a left auxiliary tree adjoined on ρ. In the following these nodes will be referred to as $A{\lhd\!-}\eta$, $B{\lhd\!-}\rho$, $A{\lhd\!-}\eta_L$, $A{\lhd\!-}\eta_R$, and $B{\lhd\!-}\rho_L$, respectively.

It is useful to have names for various segments of the fringe of the tree generated by G. In the order of the numbered substrings along the bottom of Figure 16, the segments of interest are: The *span* w_1 of $A{\lhd\!-}\eta$ is the fringe of the subtree rooted at $A{\lhd\!-}\eta$. The *left concatenated string* w_2 of $A{\lhd\!-}\eta$ is the portion of the span of $A{\lhd\!-}\eta_L$ that is left of the span of $A{\lhd\!-}\eta$. The *right concatenated string* w_3 of $A{\lhd\!-}\eta$ is the portion of the span of $A{\lhd\!-}\eta_R$ that is right of the span of $A{\lhd\!-}\eta$. The *basic left context* w_4 of $A{\lhd\!-}\eta$ is the left concatenated string of $B{\lhd\!-}\rho$ plus the portion of the span of $B{\lhd\!-}\rho$ that is left of the left concatenated string of $A{\lhd\!-}\eta$ and the span of $A{\lhd\!-}\eta$. The *far left context* w_5 of $A{\lhd\!-}\eta$ is everything from the extreme left corner of the whole generated tree up to but not including the basic left context of $A{\lhd\!-}\eta$.

If there is no left auxiliary tree adjoined on η, then the left concatenated string of $A{\lhd\!-}\eta$ is empty. If there is no right auxiliary tree adjoined on η, then the right concatenated string of $A{\lhd\!-}\eta$ is empty. If η is the root of an elementary tree, then η and ρ and therefore $A{\lhd\!-}\eta$ and $B{\lhd\!-}\rho$ are the same node, and the basic left context of $A{\lhd\!-}\eta$ is empty.

If η is on the spine of an auxiliary tree T, then the *coverage* of $A{\lhd\!-}\eta$ is the span of $A{\lhd\!-}\eta$ with the span of the node filling the foot of T removed. If η is not a spine node, then the coverage of $A{\lhd\!-}\eta$ is the same as the span of $A{\lhd\!-}\eta$. (Since LCFG does not allow wrapping auxiliary trees, the coverage of an instance a node is always a single contiguous string. There can be tree structure on both sides of a foot, but the frontier of one of these sides must consist solely of nodes labeled with ε.)

Since substitution nodes and foot nodes ϕ are replaced during derivations, they do not appear directly in derived trees. However, a node that replaces such a node ϕ is taken to be an instance of ϕ (as well as an instance of the node it is a copy of). For example, the root corresponding to $A{\lhd\!-}\eta_L$ was replaced by $A{\lhd\!-}\eta$. The span of the instance of this root is therefore the span of $A{\lhd\!-}\eta$, while the coverage of the instance of this root is the empty string.

5.2 A Simple CKY-Style Parsing Algorithm

We can assume without loss of generality that every node in $I \cup A$ has at most two children. (By adding new nodes, any LCFG can be transformed into an equivalent LCFG satisfying this condition. This transformation can be readily reversed after one or more parses have been found.) The more efficient parser in Section 6 directly supports nodes with more than two children.

The algorithm stores pairs of the form $[\eta, \mathit{code}]$ in an n by n array C. In a pair, *code* is a set over the universe L (for left adjunction) and R (for right adjunction). The fact that $[\eta, \mathit{code}] \in C[i,k]$ means that there is some derivation of some string in which an instance of η accounts for the substring $a_{i+1} \cdots a_k$. More precisely, for every node η in every elementary tree in G, the algorithm guarantees that:

- $[\eta, \emptyset] \in C[i,k]$ if and only if there is some derivation in G where $a_{i+1} \cdots a_k$ is the coverage of an instance of η.
- $[\eta, \{L\}] \in C[i,k]$ if and only if there is some derivation in G where $a_{i+1} \cdots a_k$ is the left concatenated string followed by the coverage of an instance of η.
- $[\eta, \{R\}] \in C[i,k]$ if and only if there is some derivation in G where $a_{i+1} \cdots a_k$ is the coverage followed by the right concatenated string of an instance of η.

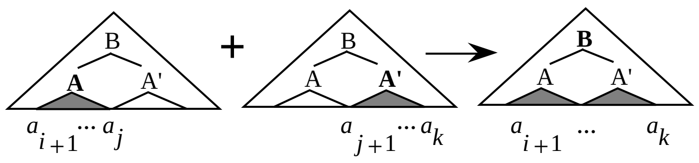


Figure 17: Sibling concatenation.

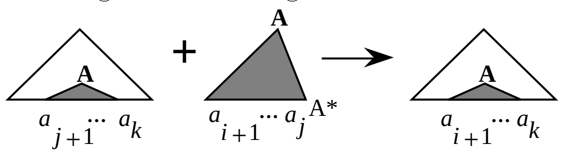


Figure 18: left concatenation.

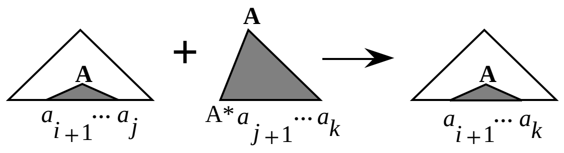


Figure 19: right concatenation.

- $[\eta, \{L, R\}] \in C[i,k]$ if and only if there is some derivation in G where $a_{i+1} \cdots a_k$ is the left concatenated string followed by the coverage followed by the right concatenated string of an instance of η.

The process starts by placing each foot node and each frontier node that is labeled with the empty string in every cell $C[i,i]$. This signifies that they each cover the empty string at all positions. The initialization also puts 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 matched substrings into longer matched substrings—it fills the upper diagonal portion of the array $C[i,k]$ $(0 \leq i \leq k \leq n)$ for increasing values of $k - i$.

Two observations are central to the efficiency of this process. Since every auxiliary tree (elementary and derived) in LCFG is either a left or right auxiliary tree, the substring matched by a tree is always a contiguous string. Further, when matched substrings are combined, one only has to consider adjacent substrings. (In LTAG, a tree with a foot can match a pair of strings that are not contiguous, one left of the foot and one right of the foot.)

There are three situations where combination of matched substrings is possible: sibling concatenation, left concatenation, and right concatenation.

As illustrated in Figure 17, sibling concatenation combines the substrings matched by two sibling nodes into a substring matched by their parent. In particular, suppose that there is a node η_0 (labeled B in Figure 17) with two children η_1 (labeled A) and η_2 (labeled A'). If $[\eta_1, \{L, R\}] \in C[i,j]$ and $[\eta_2, \{L, R\}] \in C[j,k]$ then $[\eta_0, \emptyset] \in C[i,k]$.

left concatenation (see Figure 18) combines the substring matched by a left auxiliary tree with the substring matched by a node the auxiliary tree can adjoin on. Right concatenation (see Figure 19) is analogous.

The recognizer (see Figure 20) is written in two parts: a main procedure and a subprocedure Add$(\eta, \mathit{code}, i, k)$ which adds the pair $[\eta, \mathit{code}]$ into $C[i,k]$.

The main procedure repeatedly scans the array C, building up longer and longer matched substrings until it determines whether any S-rooted derived tree matches the entire input. The

```
Procedure Recognize
begin
  ;; foot node and empty node initialization ($C[i,i]$)
  for $i = 0$ to $n$
    for all foot nodes $\phi$ in $A$; Add($\phi, \emptyset, i, i$)
    for all nodes $\eta$ in $A \cup I$ where Label($\eta$) = $\varepsilon$; Add($\eta, \emptyset, i, i$)
  ;; terminal node initialization ($C[i, i+1]$)
  for $i = 0$ to $n-1$
    for all nodes $\eta$ in $A \cup I$ where Label($\eta$) = $a_{i+1}$; Add($\eta, \emptyset, i, i+1$)
  ;; induction ($C[i,k] = C[i,j] + C[j,k]$)
  for $d = 2$ to $n$
    for $i = 0$ to $n - d$
      set $k = i + d$
      for $j = i$ to $k$
        ;; sibling concatenation
        if $[\eta, \{L,R\}] \in C[i,j]$ and [Sibling($\eta$), $\{L,R\}] \in C[j,k]$; Add(Parent($\eta$), $\emptyset, i, k$)
        ;; left concatenation
        if $[\rho, \{L,R\}] \in C[i,j]$ and $[\eta, code] \in C[j,k]$ and $L \notin code$ and Ladjoinable($\rho, \eta$);
          Add($\eta, L \cup code, i, k$)
        ;; right concatenation
        if $[\eta, code] \in C[i,j]$ and $[\rho, \{L,R\}] \in C[j,k]$ and $R \notin code$ and Radjoinable($\rho, \eta$);
          Add($\eta, R \cup code, i, k$)
for each node $\rho$ such that IsInitialRoot($\rho$) and Label($\rho$) = $S$;
  If $[\rho, \{L,R\}] \in C[0,n]$; return acceptance
end

Procedure Add($\eta, code, i, k$)
begin
  if $[\eta, code] \notin C[i,k]$;
    $C[i,k] := C[i,k] \cup [\eta, code]$
    ;; single child parent propagation
    if $code = \{L,R\}$ and FirstChild(Parent($\eta$)) = $\eta$ and Sibling($\eta$) = $\perp$; Add(Parent($\eta$), $\emptyset, i, k$)
    ;; substitution
    if $code = \{L,R\}$; for each node $\phi$ such that Substitutable($\eta, \phi$); Add($\phi, \emptyset, i, k$)
    ;; allowing skipping left adjunction
    if $L \notin code$; Add($\eta, L \cup code, i, k$)
    ;; allowing skipping right adjunction
    if $R \notin code$; Add($\eta, R \cup code, i, k$)
end
```

Figure 20: A CKY-style recognizer for LCFG.

purpose of the codes ($\{L, R\}$ etc.) is to insure that left and right adjunction can each be applied at most once 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 [12].

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 situations where information is propagated without increasing the length of the matched string—i.e., when a node is the only child of its parent, when adjunction is not performed, and when substitution occurs.

The $O(n^3)$ complexity of the recognizer follows from the three nested induction loops on d, 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.)

The algorithm does not depend on the fact that LCFG is lexicalized—it would work equally well if were not lexicalized.

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

6 An Earley-Style Cubic-Time Parser For LCFG

By combining top-down prediction as in Earley's algorithm for parsing CFGs [2] with bottom-up recognition as in the last section, one can obtain a more efficient left-to-right parsing algorithm for LCFG that maintains the valid prefix property and requires $O(n^3)$ time in the worst case. The algorithm is a general recognizer for LCFGs, which requires no condition on the grammar. It is particularly interesting in light of the fact that the best known parser for LTAG that maintains the valid prefix property requires $O(n^9)$-time in the worst case [8] versus only $O(n^6)$ for LTAG parsers that do not maintain the valid prefix property.

The algorithm collects states into a set called the chart $\mathcal{C}$. A *state* is a 4-tuple, $[\eta, pos, i, j]$, where: η is a node in an elementary tree; $pos \in \{la, lb, ra, rb\}$ is a position as described below; and $0 \leq i \leq j \leq n$ are integers ranging over positions in the input string.

During parsing, elementary trees are traversed in a top-down, left-to-right manner that visits the frontier nodes in left-to-right order. The position codes are used to represent the state of this traversal. In diagrams the positions are indicated by placing a dot either: left and above la, left and below lb, right and below rb, or right and above ra the node. As illustrated in Figure 21, the traversal begins left and above the root node and ends right and above the root node.

In a manner analogous to dotted rules for CFG as defined by Earley [1], being at a particular position with regard to a particular node, divides the tree containing the node into two parts: a *left context* consisting of nodes that have been already been traversed and a *right context* that still needs to be traversed. The situation where one is la a node is illustrated in Figure 22.

The indices i, j record the portion of the input string which is the left context. (The fact that LCFG forbids wrapping auxiliary trees guarantees that a pair of indices is always sufficient for representing the left context.) As traversal proceeds, the left context grows larger and larger. In particular, for every node η in every elementary tree in G, the algorithm guarantees the following.

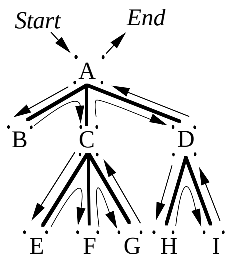


Figure 21: left to right tree traversal

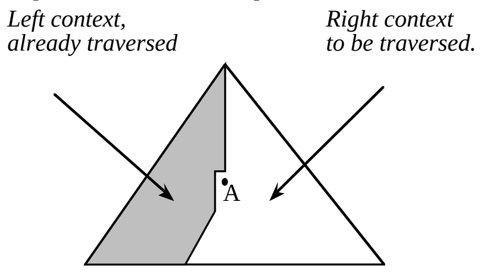
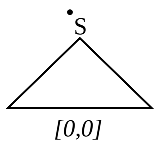


Figure 22: Left and right contexts of a node and position.

- $[\eta, la, i, j] \in \mathcal{C}$ if and only if there is some derivation in G where $a_1 \cdots a_i$ is the far left context of an instance of η and $a_{i+1} \cdots a_j$ is the basic left context of the instance of η (see Figure 16).
- $[\eta, lb, i, j] \in \mathcal{C}$ if and only if there is some derivation in G where $a_1 \cdots a_i$ is the far left context of an instance of η and $a_{i+1} \cdots a_j$ is the basic left context followed by the left concatenated string of the instance of η.
- $[\eta, rb, i, j] \in \mathcal{C}$ if and only if there is some derivation in G where $a_1 \cdots a_i$ is the far left context of an instance of η and $a_{i+1} \cdots a_j$ is the basic left context followed by the left concatenated string followed by the coverage of the instance of η.
- $[\eta, ra, i, j] \in \mathcal{C}$ if and only if there is some derivation in G where $a_1 \cdots a_i$ is the far left context of an instance of η and $a_{i+1} \cdots a_j$ is the basic left context followed by the left concatenated string followed by the coverage followed by the right concatenated string of the instance of η.

Figure 23 depicts the recognition algorithm as a set of production rules. The first rule (1) initializes the chart by adding all states of the form $[\eta, la, 0, 0]$, where η is the root of an initial tree and Label$(\eta) = S$. As illustrated below, the initial states encode the fact that any valid derivation must start from an S-type initial tree.

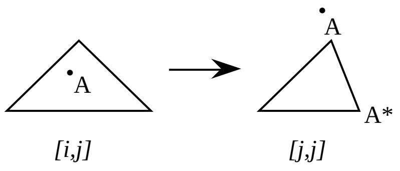


The addition of a new state to the chart $\mathcal{C}$, can trigger the addition of other states as specified by the productions in Figure 23. Computation proceeds with the introduction of more and more states until no more rules can be fired. The last rule (16) specifies that the input is recognized if and only if the final chart contains a state of the form $[\eta, ra, 0, n]$, where η is the root of an initial tree and Label$(\eta) = S$.

The traversal rules in Figure 23 control the left-to-right traversal of elementary trees. The scanning, and substitution rules recognize substitutions of trees and are similar to the steps found in Earley's parser for CFGs [2]. The left and right adjunction rules recognize the adjunction of left and right auxiliary trees. These 14 rules are discussed in detail below.

The traversal rules (2–4) move from node to node in initial trees as indicated by the arrows in Figure 21.

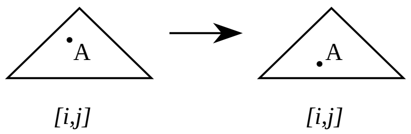
The left adjunction rules recognize left adjunction and move from position *la* to *lb*. As indicated pictorially below, rule (5) predicts the presence of left auxiliary trees. It does this top down taking account of the left context.



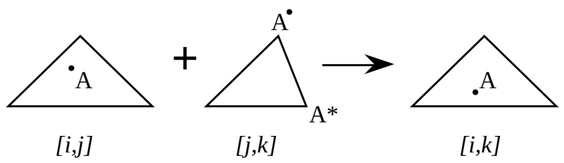
	<u>Initialization</u>		
(1)	$\text{IsInitialRoot}(\eta) \wedge \text{Label}(\eta) = S$	$\rightarrow$	$[\eta, la, 0, 0] \in \mathcal{C}$
	<u>Traversal</u>		
(2)	$[\eta, lb, i, j] \in \mathcal{C} \wedge \text{FirstChild}(\eta) \neq \perp$	$\rightarrow$	$[\text{FirstChild}(\eta), la, i, j] \in \mathcal{C}$
(3)	$[\eta, ra, i, j] \in \mathcal{C} \wedge \text{Sibling}(\eta) \neq \perp$	$\rightarrow$	$[\text{Sibling}(\eta), la, i, j] \in \mathcal{C}$
(4)	$[\eta, ra, i, j] \in \mathcal{C} \wedge \text{Sibling}(\eta) = \perp \wedge \text{Parent}(\eta) \neq \perp$	$\rightarrow$	$[\text{Parent}(\eta), rb, i, j] \in \mathcal{C}$
	<u>Left Adjunction</u>		
(5)	$[\eta, la, i, j] \in \mathcal{C} \wedge \text{Ladjoinable}(\rho, \eta)$	$\rightarrow$	$[\rho, la, j, j] \in \mathcal{C}$
(6)	$[\eta, la, i, j] \in \mathcal{C}$	$\rightarrow$	$[\eta, lb, i, j] \in \mathcal{C}$
(7)	$[\eta, la, i, j] \in \mathcal{C} \wedge [\rho, ra, j, k] \in \mathcal{C} \wedge \text{Ladjoinable}(\rho, \eta)$	$\rightarrow$	$[\eta, lb, i, k] \in \mathcal{C}$
	<u>Scanning</u>		
(8)	$[\eta, lb, i, j] \in \mathcal{C} \wedge \text{Label}(\eta) = a_{j+1}$	$\rightarrow$	$[\eta, rb, i, j+1] \in \mathcal{C}$
(9)	$[\eta, lb, i, j] \in \mathcal{C} \wedge \text{Label}(\eta) = \epsilon$	$\rightarrow$	$[\eta, rb, i, j] \in \mathcal{C}$
(10)	$[\text{Foot}(\rho), lb, i, j] \in \mathcal{C}$	$\rightarrow$	$[\text{Foot}(\rho), rb, i, j] \in \mathcal{C}$
	<u>Substitution</u>		
(11)	$[\eta, lb, i, j] \in \mathcal{C} \wedge \text{Substitutable}(\rho, \eta)$	$\rightarrow$	$[\rho, la, j, j] \in \mathcal{C}$
(12)	$[\eta, lb, i, j] \in \mathcal{C} \wedge [\rho, ra, j, k] \in \mathcal{C} \wedge \text{Substitutable}(\rho, \eta)$	$\rightarrow$	$[\eta, rb, i, k] \in \mathcal{C}$
	<u>Right Adjunction</u>		
(13)	$[\eta, rb, i, j] \in \mathcal{C} \wedge \text{Radjoinable}(\rho, \eta)$	$\rightarrow$	$[\rho, la, j, j] \in \mathcal{C}$
(14)	$[\eta, rb, i, j] \in \mathcal{C}$	$\rightarrow$	$[\eta, ra, i, j] \in \mathcal{C}$
(15)	$[\eta, rb, i, j] \in \mathcal{C} \wedge [\rho, ra, j, k] \in \mathcal{C} \wedge \text{Radjoinable}(\rho, \eta)$	$\rightarrow$	$[\eta, ra, i, k] \in \mathcal{C}$
	<u>Final Recognition</u>		
(16)	$[\eta, ra, 0, n] \in \mathcal{C} \wedge \text{IsInitialRoot}(\eta)$	$\rightarrow$	Acceptance

Figure 23: An Earley-style recognizer for LCFG, expressed using production rules.

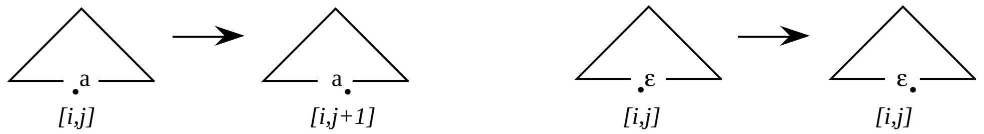
Rule (6) considers the possibility that left adjunction does not occur.



Rule (7) recognizes the adjunction of a left auxiliary tree. It is a bottom-up step, which combines a fully recognized left auxiliary tree with a partially recognized tree.

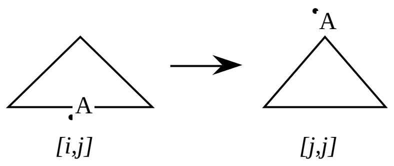


The scanning rules recognize the coverage of nodes and move from position *lb* to *rb*. The first two rules (8 & 9) match terminal nodes against elements of the input string.

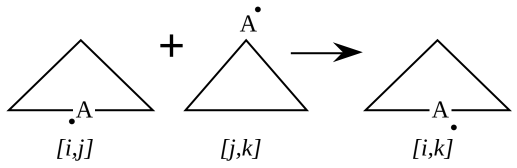


Rule (10) encodes the fact that one can move directly from the position *lb* a foot to *rb* the foot, since by the definition of cover, the foot of an auxiliary tree never covers anything.

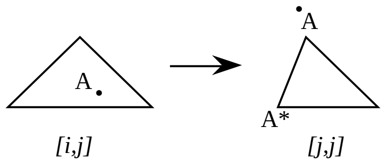
The substitution rules recognize substitution and also move from position *lb* to *rb*. The substitution prediction rule (11) predicts in a top-down fashion initial trees to be substituted. It predicts a substitution only if an appropriate left context has been scanned.



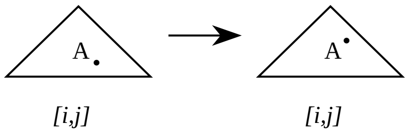
The substitution completion rule (12) recognizes a completed substitution. It is a bottom-up step that concatenates the boundaries of a fully recognized initial tree with a partially recognized tree.



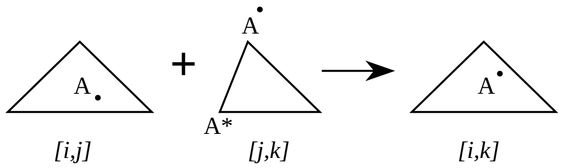
The right adjunction rules are analogous to the left adjunction rules. They recognize right adjunction and move from position *rb* to *ra*. Rule (13) predicts right adjunction.



Rule (14) considers the possibility of no right adjunction.



Rule (15) recognizes right adjunction.



The algorithm takes in the worst case $O(n^3)$ time and $O(n^2)$ space. The worst case complexity comes from the left and right completion steps. An intuition of the validity of this result

$[\eta, la, i, j] \equiv [\eta, lb, i, j]$	by rule (6) if	$\neg\exists\rho \mid \text{Ladjoinable}(\rho, \eta)$
$[\eta, lb, i, j] \equiv [\eta, rb, i, j]$	by rule (9) if	$\text{Label}(\eta) = \epsilon$
$[\phi, lb, i, j] \equiv [\phi, rb, i, j]$	by rule (10) if	$\exists\rho \mid \text{Foot}(\rho) = \phi \neq \perp$
$[\eta, lb, i, j] \equiv [\mu, la, i, j]$	by rule (2) if	$\text{FirstChild}(\eta) = \mu \neq \perp$
$[\eta, rb, i, j] \equiv [\eta, ra, i, j]$	by rule (14) if	$\neg\exists\rho \mid \text{Radjoinable}(\rho, \eta)$
$[\eta, ra, i, j] \equiv [\mu, la, i, j]$	by rule (3) if	$\text{Sibling}(\eta) = \mu \neq \perp$
$[\eta, ra, i, j] \equiv [\mu, rb, i, j]$	by rule (4) if	$\text{Sibling}(\eta) = \perp \wedge \text{Parent}(\eta) = \mu \neq \perp$

Figure 24: Equivalent states.

can be obtained by observing that these steps may be applied at most n^3 times since there are at most n^3 instances of states (corresponding to the possible ranges of the indices i, j, k).

As stated in Figure 23, the algorithm is merely a recognizer. It can be converted to a parser by keeping track of the reasons why states are added to the chart $\mathcal{C}$. In this regard, it should be noted that the algorithm always looks for left adjunction before looking for right adjunction. This is unavoidable due to the left-to-right nature of the algorithm. However, both orders of doing adjunction can be recovered when retrieving the derivations corresponding to a completed chart.

For the sake of simplicity, it was assumed above that there are no constraints on adjunction. However, the algorithm can easily be extended to handle such constraints. It can also be extended to handle the alternative definition of TAG derivation found in [12].

6.1 Efficiency Improvements To the Earley-Style Parser

When implementing the algorithm above, two steps should be taken to improve the efficiency of the result: grammar pruning and equivalent state merging.

The top-down prediction part of the algorithm does not take full advantage of the fact that the grammar is lexicalized. For instance, it creates initial states for every initial tree whose root is labeled S whether or not the anchor of the rule appears in the input. For efficiency sake, the prediction of an elementary tree at position j (by rules 5, 11, & 13) should be prevented if the anchor of the ruled does not appear in $a_{j+1} \cdots a_n$[10].

A number of the rules in Figure 23 (in particular, 2, 3, 4, 9, and 10) merely move from state to state without changing the left context i, j. These rules reflect facts about the grammar and the traversal and do not refer to the input. These rules can be precompiled out of the algorithm by viewing certain states as equivalent. During parsing one then skips directly from the first to the last state in a set of equivalent states without going through the normal rule application process and enters only the last state in the chart.

Figure 24 indicates which states are equivalent. Merging these states eliminates approximately 3/4 of the possible states in the chart $\mathcal{C}$. Eliminating the rules that are no longer needed eliminates 1/3 of the rules in Figure 23.

Each of the equivalences in Figure 24 equates adjacent states. Occasionally, long chains of equivalent states are possible. Of particular interest are the following. If ρ is the root of a right auxiliary tree then $[\rho, la, i, j] \equiv [\text{Foot}(\rho), ra, i, j]$. This is true because the foot does not cover anything, and if there is any structure left of the foot, it must all match the empty string and therefore must match against any input. Further, adjunction cannot apply anywhere on or left of the foot.

If ϕ is the foot of a left auxiliary tree then $[\phi, la, i, j] \equiv [\text{Root}(\phi), ra, i, j]$. This is true because the foot does not cover anything, and if there is any structure right of the foot, it must all match the empty string and therefore must match against any input. Further, adjunction cannot apply anywhere on or right of the foot.

7 Conclusion

The preceding discussion presents lexicalized context-free grammar (LCFG) from the perspective of its ability to lexicalize CFG while preserving the trees produced. The importance of this is that heretofore there was no $O(n^3)$-time parsable formalism that was known to lexicalize CFG.

It is interesting to consider whether there are formalisms that are even more limited than LCFG that still lexicalize CFG. For instance, even if LCFG were limited to allowing only right (or only left) auxiliary trees, lexicalization would still be possible. It is an open question whether some even greater limitation might be possible.

The preceding also presents a constructive procedure for converting CFGs into LCFGs. If extended as discussed in Section 4.1, this procedure can produce a compact LCFG corresponding to a CFG. If you are starting from a CFG G that you feel is fully satisfactory and wish to lexicalize G, the lexicalization procedure should be very effective.

The last section presents an Earley-style parser for LCFG that achieves $O(n^3)$ worst-case time bounds. Using this parser, it is possible to parse LCFGs very efficiently. In particular, it should often be possible to parse using an LCFG derived by lexicalizing a CFG significantly more quickly than one can parse using the original CFG.

It is interesting to note that there is an entirely different perspective from which to view LCFG. LCFG is a restricted form of lexicalized tree adjoining grammar (LTAG). A primary virtue of LTAG is that it supports linguistic analyses that are significantly more elegant than those that can be supported by CFG. However, this elegance comes at a significant price–LTAG requires $O(n^6)$-time to parse.

The restrictions on LCFG allow LCFG to be parsed in $O(n^3)$-time. However, rather surprisingly, *in practice* the restrictions do not appear to significantly limit the elegance of analysis being obtained by means of LTAG. In fact, the original motivation behind the development of LCFG was the observation that the natural-language grammars being currently developed use LTAG only use some of the capabilities provided by LTAG. The initial design goal of LCFG was to support just the capabilities that people were using without incurring computational overhead for capabilities that were not being used.

To take advantage of the features of LCFG that go beyond merely lexicalizing CFG, one has to hand craft an LCFG grammar. In particular, it is important to realize that the goal of the lexicalization procedure is to lexicalize a CFG. The procedure does not improve the elegance of the analyses provided by a CFG—it does not change them in any way. Rather, it allows you to obtain the same analyses faster.

8 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, Esther König, Fernando Pereira, Stuart Shieber, B. Srinivas, and the members of Mitsubishi Electric Research Laboratories for valuable discussions.

References

[1] Jay C. Earley. *An Efficient Context-Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1968.

[2] Jay C. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[3] S. A. Greibach. A new normal-form theorem for context-free phrase-structure grammars. *J. ACM*, 12:42–52, 1965.

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

[5] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

[6] Bernard Lang. 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 1990.

[7] Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* PhD thesis, University of Pennsylvania, Philadelphia, PA, August 1990. Available as technical report (MS-CIS-90-48, LINC LAB179) from the Department of Computer Science.

[8] Yves Schabes. 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 1991.

[9] Yves Schabes, Anne Abeillé, and Aravind K. Joshi. 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 1988.

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

[11] Yves Schabes and Waters R.C. Lexicalized context-free grammars. In 21^{st} *Meeting of the Association for Computational Linguistics (ACL'93)*, Columbus, Ohio, June 1993.

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

[13] Mark Steedman. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5:403–439, 1987.

[14] J. W. Thatcher. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396, 1971.

[15] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1987.