MITSUBISHI ELECTRIC RESEARCH LABORATORIES
CAMBRIDGE RESEARCH CENTER

Tree Insertion Grammar:
A Cubic-Time Parsable Formalism That Lexicalizes Context-Free Grammar Without Changing the Trees Produced

Yves Schabes and Richard C. Waters

TR-94-13 June 1994

Abstract

Tree insertion grammar (TIG) is a tree-based formalism that makes use of tree substitution and tree adjunction. TIG is related to tree adjoining grammar. However, the adjunction permitted in TIG is sufficiently restricted that TIGs only derive context free languages and TIGs have the same cubic-time worst-case complexity bounds for recognition and parsing as context free grammars. An efficient Earley-style parser for TIGs is presented.

Any context free grammar (CFG) can be converted into a lexicalized tree insertion grammar (LTIG) that generates the same trees. A constructive procedure is presented for converting a CFG into a left anchored (i.e., word initial) LTIG that preserves ambiguity and generates the same trees. The LTIG created can be represented very compactly by taking advantage of sharing between the elementary trees in it. Other methods for converting CFGs into a left anchored form, e.g., the methods of Greibach and Rosenkrantz, do not preserve the trees produced and result in very large output grammars.

For the purpose of experimental evaluation, the LTIG lexicalization procedure was applied to eight different CFGs for subsets of English. The LTIGs created were smaller than the original CFGs. Using an implementation of the Earley-style TIG parser that was specialized for left anchored LTIGs, it was possible to parse more quickly with these LTIGs than with the original CFGs.

Contents

1 Introduction

Most current linguistic theories give lexical accounts of several phenomena that used to be considered purely syntactic.[1] The information put in the lexicon is thereby increased in both amount and complexity.

In this paper, we study the problem of lexicalizing context-free grammars and show that it enables faster processing. In previous attempts to take advantage of lexicalization, a variety of lexicalization procedures have been developed that convert context free grammars (CFGs) into equivalent lexicalized grammars. However, these procedures typically suffer from one or more of the following problems.

- Lexicalization procedures such as those developed by Greibach [9] and Rosenkrantz [19] often produce very large output grammars—so large that they can be awkward or even impossible to parse with.
- Procedures that convert CFGs into lexicalized CFGs provide only a *weak* lexicalization, because while they preserve the strings derived, they do not preserve the trees derived. Parsing with the resulting grammar can be fast, but it does not produce the right trees.
- Strong lexicalization that preserves the trees derived is possible using context sensitive formalisms such as tree adjoining grammar (TAG) [13, 20]. However, these context sensitive formalisms entail much larger computation costs than CFGs—$O(n^6)$-time in the case of TAG, instead of $O(n^3)$ for CFG.

Tree insertion grammar (TIG) is a compromise between CFG and TAG that combines the efficiency of the former with the strong lexicalizing power of the latter. As discussed in Section 2, TIG is the same as TAG except that adjunction is restricted so that it is no longer a context sensitive operation.

Like CFG, TIG can be parsed in $O(|G|n^3)$-time. Section 3 presents an Earley-style parser for TIG that maintains the valid prefix property.

Section 4 presents a procedure that converts CFGs into lexicalized tree insertion grammars (LTIGs) generating the same trees. The procedure produces a left anchored LTIG—one where for each elementary tree, the first element that must be matched against the input is a lexical item.

Section 5 presents a number of experiments evaluating TIG. Section 5.1 shows that the grammars generated by the LTIG procedure can be represented very compactly. In the experiments performed, the LTIG grammars are smaller than the CFGs they are generated from. Section 5.2 investigates the practical value of the grammars created by the LTIG procedure as a vehicle for parsing CFGs. It reports a number of experiments comparing a standard Earley-style parser for CFGs with the Earley-style TIG parser of Section 3, adapted to take advantage of the left anchored nature of the grammars created by the LTIG procedure. In these experiments parsing using LTIG is typically 5 to 10 times faster.

The original motivation behind the development of TIG was the intuition that the natural-language grammars currently being developed using TAG do not make full use of the capabilities provided by TAG. This suggests a different use for TIG—as a (partial) substitute for TAG, see Section 6.

[1]Some of the linguistic formalisms illustrating the increased use of lexical information are, lexical rules in LFG [14], GPSG [7], HPSG [17], Combinatory Categorial Grammars [28], Karttunen's version of Categorial Grammar [15], some versions of GB theory [4], and Lexicon-Grammars [10].

2 Tree Insertion Grammar

Tree insertion grammar (TIG) is a tree generating system that is a restricted variant of tree-adjoining grammar (TAG) [13, 20]. As in TAG, a TIG grammar consists of two sets of trees: initial trees, which are combined by substitution and auxiliary trees, which are combined which each other and the initial trees by adjunction. However, both the auxiliary trees and the adjunction allowed are different than in TAG.

Definition 1 (TIG) A *tree insertion grammar* (TIG) is a five-tuple (Σ, NT, I, A, S), where Σ is a set of terminal symbols, NT is a set of nonterminal symbols, I is a finite set of finite initial trees, A is a finite set of finite auxiliary trees, and S is a distinguished nonterminal symbol. The set $I \cup A$ is referred to as the elementary trees.

In each initial tree the root and interior—i.e., non-root, non-leaf—nodes are labeled by nonterminal symbols. The nodes on the frontier are labeled with terminal symbols, nonterminal symbols, and the empty string (ε). The nonterminal symbols on the frontier are marked for substitution. By convention, substitutability is indicated in diagrams by using a down arrow ($\downarrow$). The root of at least one elementary initial tree must be labeled S.

In each auxiliary tree the root and interior nodes are labeled by nonterminal symbols. The nodes on the frontier are labeled with terminal symbols, nonterminal symbols, and the empty string (ε). The nonterminal symbols on the frontier of an auxiliary tree are marked for substitution, except that exactly one nonterminal frontier node 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 nonempty frontier node is to the left of the foot are called *left* auxiliary trees. Similarly, auxiliary trees in which every nonempty frontier node is to the right of the foot are called *right* auxiliary trees. Other auxiliary trees are called *wrapping* auxiliary trees.[2]

The root of each elementary tree must have at least one child. Frontier nodes labeled with ε are referred to as *empty*. If all the frontier nodes of an initial tree are empty, the tree is referred to as *empty*. If all the frontier nodes other than the foot of an auxiliary tree are empty, the tree is referred to as *empty*.

The operations of substitution and adjunction are discussed in detail below. Substitution replaces a node marked for substitution with an initial tree. Adjunction replaces a node with an auxiliary tree.

To this point, the definition of a TIG is essentially identical to the definition of a TAG. However, the following differs from the definition of TAG.

TIG does not allow there to be any elementary wrapping auxiliary trees or elementary empty auxiliary trees. This ensures that every elementary auxiliary tree will be uniquely either a left auxiliary tree or a right auxiliary tree. Wrapping auxiliary trees are neither. Empty auxiliary trees are both and cause infinite ambiguity.

TIG does not allow a left (right) auxiliary tree to be adjoined on any node that is on the spine of a right (left) auxiliary tree. Further, no adjunction whatever is permitted on a node μ 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 μ must be labeled with ε.

[2]In [23] these three kinds of auxiliary trees are referred to differently as right recursive, left recursive, and centrally recursive, respectively.

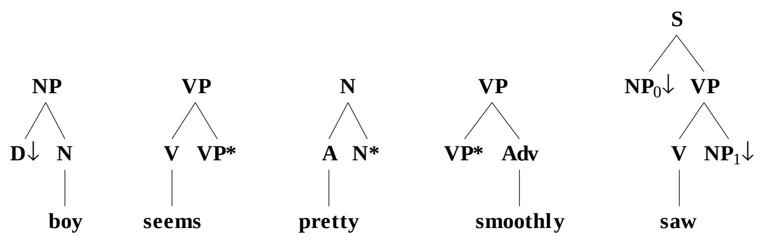


Figure 1: Example TIG trees.

TIG allows arbitrarily many simultaneous adjunctions on a single node in a manner similar to the alternative TAG derivation defined in [26]. This is specified in terms of two sequences, one of left auxiliary trees and the other of right auxiliary trees, which specify the order the strings corresponding to the trees are combined.

A TIG derivation starts with an initial tree rooted at S. This tree is repeatedly extended using substitution and adjunction. A derivation is complete when every frontier node in the tree(s) derived is labeled with a terminal symbol. By means of adjunction, complete derivations can be extended to bigger complete derivations.

As in TAG, but in contrast to CFG, there is an important difference in TIG between a derivation and the tree derived. By means of simultaneous adjunction, there can be several trees created by a single derivation. In addition, there can be several different derivations for the same tree.

To eliminate useless ambiguity in derivations, TIG prohibits adjunction: at nodes marked for substitution, because the same trees can be created by adjoining on the roots of the trees substituted at these nodes; at foot nodes of auxiliary trees, because the same trees can be created by simultaneous adjunction on the nodes the auxiliary trees are adjoined on; and at the roots of auxiliary trees, because the same trees can be created by simultaneous adjunction on the nodes the auxiliary trees are adjoined on.

Figure 1, shows five elementary trees that might appear in a TIG for English. The trees containing 'boy' and 'saw' are initial trees. The remainder are auxiliary trees.

As illustrated in Figure 2, substitution inserts an initial tree T in place of a frontier node μ that has the same label as the root of T and is marked for substitution.

Adjunction inserts a copy of an auxiliary tree T into another tree at a node μ that has the same label as the root (and therefore foot) of T. In particular, μ is replaced by a copy of T and the foot of the copy of T is replaced by the subtree rooted at μ.

The adjunction of a left auxiliary tree is referred to as left adjunction. This is illustrated in Figure 3. The adjunction of a right auxiliary tree is referred to as right adjunction (see Figure 4).

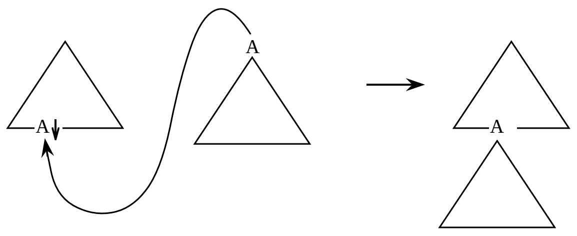


Figure 2: Substitution.

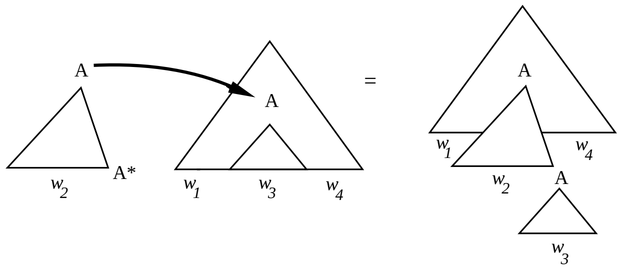


Figure 3: Left adjunction.

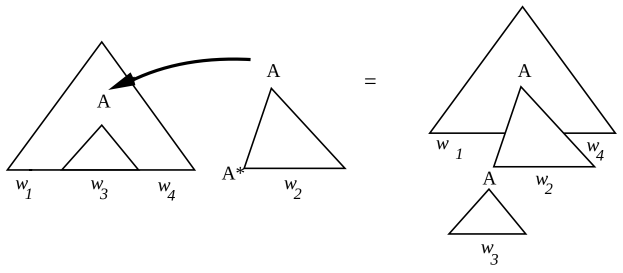


Figure 4: Right adjunction.

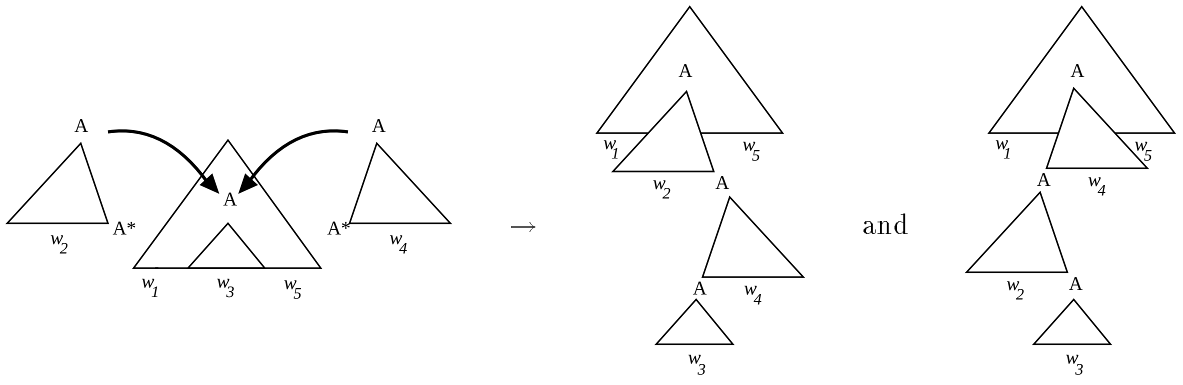


Figure 5: Simultaneous left and right adjunction.

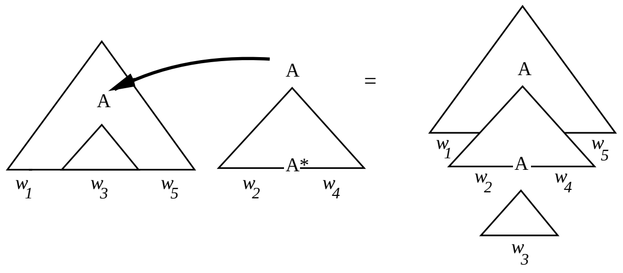


Figure 6: Wrapping adjunction.

Simultaneous adjunction is fundamentally ambiguous in nature and typically results in the creation of several different trees. The order in the sequences of left and right auxiliary trees fixes the order of the strings being combined. However, unless one of the sequences is empty, variability is possible in the trees that can be produced. The TIG formalism specifies that every tree is produced that is consistent with the specified order.

Figure 5 illustrates the simultaneous adjunction of one left and one right auxiliary tree on a node. The string corresponding to the left auxiliary tree must precede the node and the string corresponding to the right auxiliary tree must follow it. However, two different trees can be derived—one where the left auxiliary tree is on top and one where the right auxiliary tree is on top. Similarly, the simultaneous adjunction of two left and two right auxiliary trees leads to 6 derived trees.

The adjunction of a wrapping auxiliary tree is referred to as wrapping adjunction. This is illustrated in Figure 6. The key force of the restrictions applied to TIG in comparison with TAG are that they prevent wrapping adjunction from occurring, by preventing the creation of wrapping auxiliary trees.[3]

Wrapping adjunction is context sensitive in nature because two strings that are mutually constrained by being in the same auxiliary tree are wrapped around another string. In contrast, every operation allowed by a TIG inserts a string into another string. (Simultaneous adjunction merely specifies multiple independent insertions. Simultaneous left and right adjunction is not an instance of wrapping, because TIG does not allow there to be any constraints between the adjoinability of the trees in question.)

There are many ways that the TIG formalism could be extended. First, adjoining constraints could be used to prohibit the adjunction of particular auxiliary trees (or all auxiliary trees) at a node.

Second, one can easily imagine variants of TIG where simultaneous adjunction is more limited. One could allow only one canonical derived tree. One could allow at most one left auxiliary tree and one right auxiliary tree as we did in [23]. One could forbid multiple adjunction altogether. We have chosen unlimited simultaneous adjunction here primarily because it allows more efficient parsing.

Third, one can introduce stochastic parameters controlling the probabilities with which particular substitutions and adjunctions occur (see [24]).

Fourth, and of particular importance in the current paper, one can require that a TIG be lexicalized.

Definition 2 (LTIG) A *lexicalized tree insertion grammar* (LTIG)[4] (Σ, NT, I, A, S) is a TIG where every elementary tree in $I \cup A$ is lexicalized. A tree is lexicalized if at least one frontier node is labeled with a terminal symbol.

An LTIG is said to be *left anchored* if every elementary tree is left anchored. An elementary TIG tree is left anchored if the first nonempty frontier element other than the foot, if any, is a lexical item. All the trees in Figure 1 are lexicalized; however, only the ones containing *seems*, *pretty*, and *smoothly* are left anchored.

[3]Using a simple case by case analysis, one can show that given a TIG, it is not possible to create wrapping auxiliary trees. A proof of this fact is presented in Appendix A.

[4]In [23] a formalism almost identical to LTIG is referred to as lexicalized context free grammar (LCFG). A different name is used here to highlight the importance of the non-lexicalized formalism, which was not given a name in [23].

2.1 CFG, TIG and TAG

In this section we briefly compare CFG, TIG and TAG, noting that TIG shares a number of properties with CFG on one hand and TAG on the other.

To start with, note that any CFG can be trivially converted into a TIG that derives the same trees by converting each rule R into a single-level initial tree. If the right hand side of R is empty, the initial tree created has a single frontier element labeled with ε. Otherwise, the elements of the right hand side of R become the labels on the frontier of the initial tree, with the nonterminals marked for substitution.

Similarly, any TIG that does not make use of adjoining constraints can be easily converted into a TAG that derives the same trees; however, adjoining constraints may have to be used in the TAG. The trivial nature of the conversion can be seen by considering the three differences between TIG and TAG.

First, TIG prohibits elementary wrapping auxiliary trees. From the perspective of this difference, a TIG is trivially a TAG without the need for any alterations.

Second, TIG prohibits adjunction on the roots of auxiliary trees and allows simultaneous adjunction while TAG allows adjunction on the roots of auxiliary trees and prohibits simultaneous adjunction. From the perspective of this difference in approach, a TIG is also trivially a TAG without alteration. To see this, consider the following. Suppose that there are a set of auxiliary trees T that are allowed to adjoin on a node μ in a TIG. Simultaneous adjunction in TIG allows these auxiliary trees to be chained together in every possible way root-to-foot on μ. The same is true in a TAG where the trees in T are allowed to adjoin on each other's roots.

Third, TIG imposes a number of detailed restrictions on the interaction of left and right auxiliary trees. To convert a TIG into a TAG deriving the same trees and no more, one has to capture these restrictions. In general, this requires the use of adjoining constraints to prohibit the forbidden adjunctions.

It should be noted that if a TIG makes use of adjoining constraints, then the conversion of the TIG to a TAG can become significantly more complex or even impossible, depending on the details of exactly how the adjoining constraints are allowed to act in the TIG and TAG.

TIG generates context-free languages. Like CFG, TIG generates context-free languages. In contrast, TAG generates so called tree adjoining languages (TALs) [12].

The fact that any context-free language can be generated by a TIG follows from the fact that any CFG can be converted into a TIG. The fact that TIGs can only generate context-free languages follows from the fact that any TIG can be converted into a CFG generating the same language as shown in the following theorem.

Theorem 1 If $G = (\Sigma, NT, I, A, S)$ is a TIG then there is a CFG[5] $G' = (\Sigma, NT', P, S)$ that generates the same string set.

Proof: The key step in converting a TIG into a CFG is eliminating the auxiliary trees. Given only initial trees, the final conversion to a CFG is trivial.

Step 1: For each nonterminal A_i in NT, add two more nonterminals Y_i and Z_i. This yields the new nonterminal set NT'.

Step 2: For each nonterminal A_i, include the following rules in P: $Y_i \rightarrow \varepsilon$ and $Z_i \rightarrow \varepsilon$.

[5] As usual, a *context free grammar* (CFG) G is a four-tuple (Σ, NT, P, S) where Σ is a set of terminal symbols, NT is a set of nonterminal symbols, P is a finite set of finite production rules that rewrite nonterminal symbols to, possibly empty, strings of terminal and nonterminal symbols, and S is a distinguished nonterminal symbol that is the start symbol of any derivation.

Step 3: Alter every node μ in every elementary tree in I and A as follows: Let A_i be the label of μ. If and only if left adjunction is possible at μ, add a new leftmost child of μ labeled Y_i and mark it for substitution. If and only if right adjunction is possible at μ, add a new rightmost child of μ labeled Z_i and mark it for substitution.

Step 4: Convert every auxiliary tree t in A into an initial tree as follows: Let A_i be the label of the root μ of t. If t is a left auxiliary tree, add a new root labeled Y_i with two children: μ on the left and on the right, a node labeled Y_i and marked for substitution. Otherwise add a new root labeled Z_i with two children: μ on the left and on the right, a node labeled Z_i and marked for substitution. Relabel the foot of t with ε, turning t into an initial tree.

Step 5: Every elementary tree t is now an initial tree. Each one is converted into a rule in P as follows. The label of the root of t becomes the left hand side of R. The labels on the frontier of t with any instances of ε omitted become the right hand side of R.

Every derivation in G maps directly to a derivation in G' that generates the same string. Substitution steps map directly. Adjunctions are converted into substitutions via the new non-terminals Y_i and Z_i. The new roots and their rightmost children labeled Y_i and Z_i created in Step 3 allow arbitrarily many simultaneous adjunctions at a node. The right linear ordering inherent in these structures encodes the ordering information specified for a simultaneous adjunction. □

It should be noted that while G' generates the same strings as G, it does not generate the same trees. It is interesting to consider two examples of this.

First, simultaneous adjunction can lead to the creation of many different trees. In contrast, the rules in Step 2 generate only one tree for a simultaneous adjunction. This works because all the trees created by simultaneous adjunction correspond to the same string, and the rules in Step 2 generate this same string.

Second, the substitutions in G' that correspond to adjunctions in G create trees that are very different from the trees generated by G. For instance, if a left auxiliary tree T has structure to the right of its spine, this structure ends up on the left rather than the right of the node "adjoined on" in G'. However, this does not alter the strings that are generated, because by the definition of TIG the structure to the right of the spine of T must be entirely empty.

The construction in the theorem above does not work to convert TAGs into CFGs, because the construction involving Y_i and Z_i does not work for wrapping auxiliary trees. The reason for this is that a wrapping auxiliary tree has nonempty structure on both the left and the right of its spine.

TIG generates context-free path sets. 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 $\Sigma \cup NT \cup \{\varepsilon\}\}$. CFGs have path sets that are regular languages (RLs) [29]. In contrast, TAGs have path sets that are context-free languages (CFLs) [33].

The fact that the path sets generated by a TIG cannot be more complex than context-free languages follows from the fact that TIGs can be converted into TAGs generating the same trees. The fact that TIGs can generate path sets more complex than regular languages is shown by the following example.

Consider the TIG in Figure 7. The path set L generated by this grammar contains a variety of paths including Sx (from the elementary initial tree), $SASBAx$ and SAa (from adjoining the elementary auxiliary tree once on the initial tree), and so on. By relying on the fact that the intersection of two regular languages must be regular, it is easy to show that L is not a

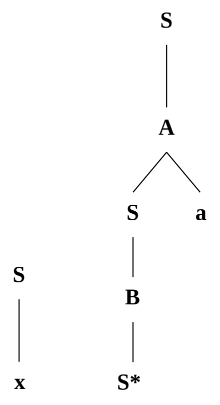


Figure 7: A TIG with a context-free path set.

regular language. In particular, consider:

$$L \cap \{SA\}^*S\{BS\}^*x = \{SA\}^nS\{BS\}^nx$$

This intersection corresponds to all the paths from root to x in the trees that are generated by recursively embedding the elementary auxiliary tree in Figure 7 into the middle of its spine. Since this intersection is not a regular language, L cannot be a regular language.

3 An Earley-Style Cubic-Time Parser For TIG

Since TIG is a restricted case of tree-adjoining grammar (TAG), standard $O(n^6)$-time TAG parsers [16, 21, 31, 32] can be used for parsing TIG. Further, they can be easily optimized to require at most $O(n^4)$-time when applied to a TIG. However, this still does not take full advantage of the context-freeness of TIG.

A simple $O(n^3)$-time bottom-up recognizer for TIG in the style of the CKY parser for CFG can be straightforwardly constructed following the approach shown in [23].

As shown below, one can obtain a more efficient left-to-right parsing algorithm for TIG that maintains the valid prefix property and requires $O(n^3)$ time in the worst case, by combining top-down prediction as in Earley's algorithm for parsing CFGs [6] with bottom-up recognition. The algorithm is a general recognizer for TIGs, which requires no condition on the grammar. This parser is the more remarkable because for TAG the best parser known that maintains the valid prefix property requires in the worst case more time than parsers that do not maintain the valid prefix property ($O(n^9)$-time versus $O(n^6)$) [21].

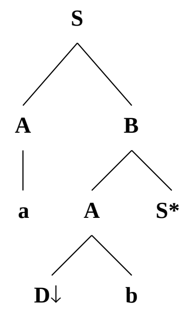
Notation. Suppose that $G = (\Sigma, NT, I, A, S)$ is a TIG and that $a_1 \cdots a_n$ is an input string. The Greek letters μ, ν, and ρ are used to designate nodes in elementary trees. Subscripts are used to indicate the label on a node, e.g., μ_X. Superscripts are sometimes used to distinguish between nodes.

A layer of an elementary tree is represented textually in a style similar to a production rule, e.g., $\mu_X \rightarrow \nu_Y\, \rho_Z$. For instance, the tree in Figure 8 is represented in terms of four layer productions as shown on the right of the figure.

The predicate $\text{Init}(\mu_X)$ is true if and only if μ_X is the root of an initial tree. The predicate $\text{LeftAux}(\rho_X)$ is true if and only if ρ_X is the root of an elementary left auxiliary tree. The predicate $\text{RightAux}(\rho_X)$ is true if and only if ρ_X is the root of an elementary right auxiliary tree. The predicate $\text{Subst}(\mu_X)$ is true if and only if μ_X is marked for substitution. The predicate $\text{Foot}(\mu_X)$ is true if and only if μ_X is the foot of an auxiliary tree.

Chart states. The Earley-style TIG parser collects states into a set called the *chart*, $\mathcal{C}$. A *state* is a 3-tuple, $[p, i, j]$ where: p is a position in an elementary tree as described below; and $0 \leq i \leq j \leq n$ are integers indicating a span of 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, see Figure 9. Positions, which are depicted as dots in Figure 9, are used to represent the state of this traversal.



$\mu_S^1 \rightarrow \mu_A^2\, \mu_B^4$
$\mu_A^2 \rightarrow \mu_a^3$
$\mu_B^4 \rightarrow \mu_A^5\, \mu_S^8$
$\mu_A^5 \rightarrow \mu_D^6\, \mu_b^7$
$\text{LeftAux}(\mu_S^1)$
$\text{Subst}(\mu_D^6)$
$\text{Foot}(\mu_S^8)$

Figure 8: An auxiliary tree and its textual representation.

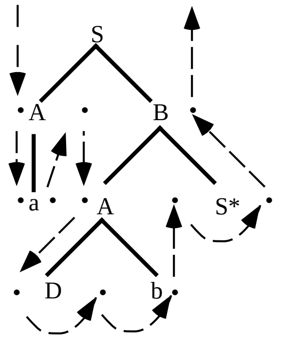


Figure 9: Left-to-right tree traversal

In a manner analogous to dotted rules for CFG as defined by Earley [5], being at a particular position with regard to a particular node, divides the subtree rooted at the node into two parts: a *left context* consisting of children that have been already been matched and a *right context* that still needs to be matched.

Positions are represented by placing a dot in the production for the corresponding layer. For example, the fourth position reached in Figure 9 is represented as $\mu_S^1 \rightarrow \mu_A^2 \bullet \mu_B^4$. In dotted layer productions, the Greek letters α, β, and γ are used to represent sequences of zero or more nodes.

The indices i, j record the portion of the input string that is spanned by the left context. The fact that TIG forbids wrapping auxiliary trees guarantees that a pair of indices is always sufficient for representing a left context. As traversal proceeds, the left context grows larger and larger.

Correctness condition. Given an input string $a_1 \cdots a_n$, for every node μ_X in every elementary tree in G, the Earley-style TIG parsing algorithm guarantees that:

- $[\mu_X \rightarrow \alpha \bullet \beta, i, j] \in \mathcal{C}$ if and only if there is some derivation in G of some string beginning with $a_1 \cdots a_j$ where $a_{i+1} \cdots a_j$ is spanned by:
  - A sequence of zero or more left auxiliary trees simultaneously adjoined on μ_X plus
  - The children of μ_X corresponding to α plus
  - if $\beta = \varepsilon$, zero or more right auxiliary trees simultaneously adjoined on μ_X.

The algorithm. Figure 10 depicts the Earley-style TIG parsing algorithm as a set of inference rules. Using the deductive parser developed by Shieber, Schabes, and Pereira [27], we were able to experiment with the TIG parser represented directly in this form (see Section 5).

The first rule (1) initializes the chart by adding all states of the form $[\mu_S \rightarrow \bullet \alpha, 0, 0]$, where μ_S is the root of an initial tree. The initial states encode the fact that any valid derivation must start from an initial tree whose root is labeled S.

The addition of a new state to the chart can trigger the addition of other states as specified by the inference rules in Figure 10. Computation proceeds with the introduction of more and more states until no more inferences are possible. The last rule (13) specifies that the input is recognized if and only if the final chart contains a state of the form $[\mu_S \rightarrow \alpha \bullet, 0, n]$, where μ_S is the root of an initial tree.

$$
\begin{array}{rcll}
 & \underline{\text{Initialization}} & & \\
\text{Init}(\mu_S) & \vdash & [\mu_S\rightarrow\bullet\alpha,0,0] & (1) \\
 & \underline{\text{Left Adjunction}} & & \\
[\mu_A\rightarrow\bullet\alpha,i,j] \wedge \text{LeftAux}(\rho_A) & \vdash & [\rho_A\rightarrow\bullet\gamma,j,j] & (2) \\
[\mu_A\rightarrow\bullet\alpha,i,j] \wedge [\rho_A\rightarrow\gamma\bullet,j,k] \wedge \text{LeftAux}(\rho_A) & \vdash & [\mu_A\rightarrow\bullet\alpha,i,k] & (3) \\
 & \underline{\text{Scanning}} & & \\
[\mu_A\rightarrow\alpha\bullet\nu_a\,\beta,i,j] \wedge a=a_{j+1} & \vdash & [\mu_A\rightarrow\alpha\,\nu_a\bullet\beta,i,j{+}1] & (4) \\
[\mu_A\rightarrow\alpha\bullet\nu_a\,\beta,i,j] \wedge a=\varepsilon & \vdash & [\mu_A\rightarrow\alpha\,\nu_a\bullet\beta,i,j] & (5) \\
[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j] \wedge \text{Foot}(\nu_B) & \vdash & [\mu_A\rightarrow\alpha\,\nu_B\bullet\beta,i,j] & (6) \\
 & \underline{\text{Substitution}} & & \\
[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j] \wedge \text{Subst}(\nu_B) \wedge \text{Init}(\rho_B) & \vdash & [\rho_B\rightarrow\bullet\gamma,j,j] & (7) \\
[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j] \wedge [\rho_B\rightarrow\gamma\bullet,j,k] \wedge \text{Subst}(\nu_B) \wedge \text{Init}(\rho_B) & \vdash & [\mu_A\rightarrow\alpha\,\nu_B\bullet\beta,i,k] & (8) \\
 & \underline{\text{Subtree Traversal}} & & \\
[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j] & \vdash & [\nu_B\rightarrow\bullet\gamma,j,j] & (9) \\
[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j] \wedge [\nu_B\rightarrow\gamma\bullet,j,k] & \vdash & [\mu_A\rightarrow\alpha\,\nu_B\bullet\beta,i,k] & (10) \\
 & \underline{\text{Right Adjunction}} & & \\
[\mu_A\rightarrow\alpha\bullet,i,j] \wedge \text{RightAux}(\rho_A) & \vdash & [\rho_A\rightarrow\bullet\gamma,j,j] & (11) \\
[\mu_A\rightarrow\alpha\bullet,i,j] \wedge [\rho_A\rightarrow\gamma\bullet,j,k] \wedge \text{RightAux}(\rho_A) & \vdash & [\mu_A\rightarrow\alpha\bullet,i,k] & (12) \\
 & \underline{\text{Final Recognition}} & & \\
[\mu_S\rightarrow\alpha\bullet,0,n] \wedge \text{Init}(\mu_S) & \vdash & \text{Acceptance} & (13)
\end{array}
$$

Figure 10: An Earley-style recognizer for TIG, expressed using inference rules.

The scanning, and substitution rules recognize terminal symbols and substitutions of trees. They are similar to the steps found in Earley's parser for CFGs [6].

The scanning rules match fringe nodes against the input string. Rule 4, recognizes the presence of a terminal symbol in the input string. Rules (5 & 6) encode the fact that one can skip over nodes label with ε and foot nodes without having to match anything.

The substitution rules are triggered by states of the form $[\mu_A\rightarrow\alpha\bullet\nu_B\,\beta,i,j]$ where ν_B is a node at which substitution can occur. Rule (7) predicts a substitution. It does this top down only if an appropriate prefix string has been found. Rule (8) 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 subtree traversal rules control the recognition of subtrees. Rule (9) predicts a subtree if and only if the previous siblings have already been recognized. Rule (10) completes the recognition of a subtree.

Rules (9 & 10) are closely analogous to rules (7 & 8). They can be looked at as recognizing a subtree that is required to be substituted at a particular spot as opposed to a subtree that may be substituted at a particular spot.

The left and right adjunction rules recognize the adjunction of left and right auxiliary trees. The left adjunction rules are triggered by states of the form $[\mu_A\rightarrow\bullet\alpha,i,j]$. Rule (2) predicts the presence of a left auxiliary tree, if and only if a node that the auxiliary tree can adjoin on has already been predicted. Rule (3) supports the bottom-up recognition of the adjunction of a left auxiliary tree.

The fact that left adjunction can occur any number of times (including zero) is captured by the fact that states of the form $[\mu_A \rightarrow \bullet\alpha, i, j]$ represent both situations where left adjunction can occur and situations where it has occurred.

The right adjunction Rules (11 & 12) are analogous to the left adjunction rules, but are triggered by states of the form $[\mu_A \rightarrow \alpha\bullet, i, j]$.

As written in Figure 10, the algorithm is a recognizer. However, it can be straightforwardly converted to a parser by keeping track of the reasons why states are added to the chart. Derivations (and therefore trees) can then be retrieved from the chart in linear time.

For the sake of simplicity, it was assumed in the discussion above that there are no adjunction constraints. However, the algorithm can easily be extended to handle such constraints.

Computational bounds. The algorithm in Figure 10 requires space $O(|G|n^2)$ in the worst case. In this equation n is the length of the input string and $|G|$ is the *size* of the grammar G. For the TIG parser, $|G|$ is computed as the sum over all the non-leaf nodes μ in all the elementary trees in G of: one plus the number of children of μ. The correctness of this space bound can be seen by observing that there are only $|G|n^2$ possible chart states $[\mu_X \rightarrow \alpha\bullet\beta, i, j]$.

The algorithm takes $O(|G|^2n^3)$ time in the worst case. This can informally be seen by noting that the worst case complexity is due to the completion rules (3, 8, 10, & 12) because they apply to a pair of states, rather than just one state. Since each of the completion rules requires that the chart states be adjacent in the string, each can apply at most $O(|G|^2n^3)$ times since there are at most n^3 possibilities for $0 \leq i \leq j \leq k \leq n$.

3.1 Improving the Efficiency of the TIG Parser

As presented in Figure 10, the TIG parser is optimized for clarity rather than speed. There are several ways that the efficiency of the TIG parser can be improved.

Parsing that is linear in the grammar size. The time complexity of the parser can be reduced from $O(|G|^2n^3)$ to $O(|G|n^3)$ by using the techniques described in [8]. This improvement is very important, because as a practical matter $|G|$ is typically much larger than n. The speedup can be achieved by altering the parser in two ways.

The prediction rules (2, 7, 9, & 11) can apply $O(|G|^2n^2)$ times, because they are triggered by a chart state and grammar node ρ; and for each of $O(|G|n^2)$ possible values of the former there can be $O(|G|)$ values of the latter. However, the new chart state produced by the prediction rules does not depend on the identity of the node in the triggering chart element nor on the value of i, but rather only on whether there is any chart element ending at j that makes the relevant prediction. Therefore, the parser can be changed so that a prediction rule is triggered at most once for any j and ρ. This reduces the prediction rules to a time complexity of only $O(|G|n)$.

The completion rules (3, 8, 10, & 12) can apply $O(|G|^2n^3)$ times, because they are triggered by pairs of chart states; and there can be $O(|G|)$ possibilities for each element of the pair for each $i < j < k$. However, the new chart state produced by the completion rules does not depend on the identity of the node ρ in the second chart element, but rather only on whether there is any appropriate chart element from j to k. Therefore, the parser can be changed so that a completion rule is triggered at most once for any possible first chart state and k. This reduces the completion rules to a time complexity of $O(|G|n^3)$.

Eliminating equivalent states. Rules (5 & 6) merely move from state to state without changing the span i, j. These rules reflect facts about the grammar and the traversal that do not depend on the input. These rules can be largely precompiled out of the algorithm by noting that the following states are equivalent.

$$\begin{array}{rl} [\mu_A \rightarrow \bullet \nu_X\ \alpha, i, j] \equiv [\mu_A \rightarrow \nu_X \bullet \alpha, i, j] & \text{if} \quad (X = \varepsilon \vee \text{Foot}(\nu_X)) \wedge \neg\exists\rho_A\ \text{LeftAux}(\rho_A) \\ [\mu_A \rightarrow \alpha \bullet \nu_X\ \beta, i, j] \equiv [\mu_A \rightarrow \alpha\, \nu_X \bullet \beta, i, j] & \text{if} \quad (X = \varepsilon \vee \text{Foot}(\nu_X)) \end{array}$$

To take advantage of equivalent states during parsing, one skips directly from the first to the last state in a set of equivalent states. This avoids going through the normal rule application process and has the effect of reducing the grammar size.

For a state $[\mu_A \rightarrow \bullet \nu_X\ \alpha, i, j]$ to be equivalent to $[\mu_A \rightarrow \nu_X \bullet \alpha, i, j]$, it is not sufficient that the first child of ν_X be empty or a foot node. It must also be the case that left adjunction is not possible on μ_A. If left adjunction is possible on μ_A, the state $[\mu_A \rightarrow \bullet \nu_X\ \alpha, i, j]$ must be independently retained in order to trigger left adjunction when appropriate.

Sharing nodes in a TIG. An important feature of the parser in Figure 10 is that the nth child of a node need not be unique and a subtree need not have only one parent. This indicates that a subtree or a supertree appears several different places in the grammar. The only requirement when sharing nodes is that every possible way of constructing a tree that is consistent with the parent-child relationships must be a valid elementary tree in the grammar.

For example consider the trees in Figure 11. They can be represented individually as follows:

$$\mu_S^1 \rightarrow \mu_A^2\, \mu_B^4,\ \mu_A^2 \rightarrow \mu_a^3,\ \mu_B^4 \rightarrow \mu_A^5\, \mu_S^8,\ \mu_A^5 \rightarrow \mu_D^6\, \mu_b^7,\ \text{LeftAux}(\mu_S^1),\ \text{Subst}(\mu_D^6),\ \text{Foot}(\mu_S^8),$$
$$\nu_S^1 \rightarrow \nu_A^2\, \nu_B^4,\ \nu_A^2 \rightarrow \nu_a^3,\ \nu_B^4 \rightarrow \nu_A^5\, \nu_S^7,\ \nu_A^5 \rightarrow \nu_a^6,\ \text{LeftAux}(\nu_S^1),\ \text{Foot}(\nu_S^7)$$

However, taking maximum advantage of sharing within and between the trees, they can be represented much more compactly as:

$$\mu_S^1 \rightarrow \mu_A^2\, \mu_B^4,\ \mu_A^2 \rightarrow \mu_a^3,\ \mu_B^4 \rightarrow \{\mu_A^5 | \mu_A^2\}\, \mu_S^8,\ \mu_A^5 \rightarrow \mu_D^6\, \mu_b^7,\ \text{LeftAux}(\mu_S^1), \text{Subst}(\mu_D^6),\ \text{Foot}(\mu_S^8)$$

In the above, two kinds of sharing are apparent. Subtrees are shared by using the same node (for example μ_A) on the right-hand side of more than one layer production. Supertrees are shared by explicitly recording the fact that there are multiple alternatives for the nth child of a some node. This is represented textually above using curly braces.

In the case of Figure 11, sharing reduces the grammar size $|G|$ from 21 to 11. Depending on the amount of sharing present in a grammar, an exponential decrease in the grammar size is possible.

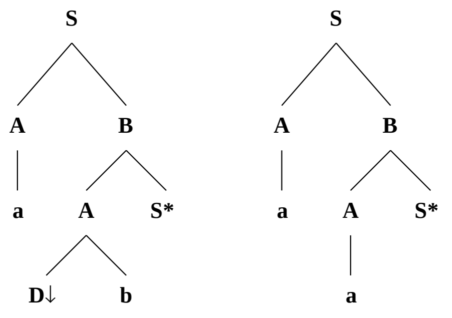


Figure 11: A pair of TIG trees.

Parsing left anchored LTIGs. The algorithm above can be extended to take advantage of the fact that the elementary trees in an LTIG are lexicalized. This does not change the worst case complexity, but is a dramatic improvement in typical situations, because it has the effect of dramatically reducing the size of the grammar that has to be considered when parsing a particular input string.

Space does not permit a discussion of all the ways lexical sensitivity can be introduced into the TIG parser. However, one way of doing this is particularly important in the context of this paper. The LTIG lexicalization procedure presented in Section 4 produces grammars that have no left auxiliary trees and are left anchored—ones where for each elementary tree, the first element that must be matched against the input is a lexical item. By means of two simple changes in the prediction rules, the TIG parser can benefit greatly from this kind of lexicalization.

First, whenever considering a node μ_B for prediction at position j, it should only be predicted if its anchor is equal to the next input item a_{j+1}. Other predictions cannot lead to successful matches.

If sharing is being used, then one chart state can correspond to a number of different positions in different trees. As a result, even though every tree has a unique left anchor, a given chart state can correspond to a set of such trees and therefore a set of such anchors. A prediction should be made if any of these anchors is the next element of the input.

Second, when predicting a node μ_B whose first child is a terminal symbol, it is known from the above that this child must match the next input element. Therefore, there is no need to create the state $[\mu_B \rightarrow \bullet \nu_a\, \alpha, j, j]$. One can instead skip directly to the state $[\mu_B \rightarrow \nu_a \bullet \alpha, j, j+1]$.

Both of the changes above depend critically on the fact that there are no left auxiliary trees. In particular, if there is a left auxiliary tree ρ_B that can be adjoined on μ_B, then the next input item may be matched by ρ_B rather than μ_B; and neither of the shortcuts above can be applied.

4 TIG Strongly Lexicalizes CFG

In the following, we say that a grammar is *lexicalized* [20, 22] if every elementary structure contains a terminal symbol called the *anchor*. A CFG is lexicalized if every production rule contains a terminal. Similarly, a TIG is lexicalized if every tree contains a terminal symbol.

A formalism F' is said to *lexicalize* [13] another formalism F, if for every 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 string set.

F' is said to *strongly lexicalize* 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 string set and tree set.

The restrictions on the form of G in the definitions above are motivated by two key properties of lexicalized grammars [13]. First, lexicalized grammars cannot derive the empty string, because every structure introduces at least one lexical item. Thus, if a CFG is to be lexicalized, it must not be the case that $S \overset{*}{\Rightarrow} \varepsilon$.

Second, 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 strongly 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 shown by Greibach [9] and Rosenkrantz [19], any CFG grammar that does not generate the empty string can be converted into a lexicalized CFG. Moreover, this grammar can be left anchored—one where the first element of the right hand side of each rule is a terminal symbol. However, this is only a weak lexicalization, because the trees generated by the lexicalized grammar are not the same as those generated by the original CFG.

CFGs can also be lexicalized by converting them into categorial grammars [2]. However, these are again only weak lexicalizations because the trees produced are not preserved.

Strong lexicalization can be obtained using TAG, but only at the cost of $O(n^6)$ parsing [13, 20]. A key virtue of TIG is that it is both $O(n^3)$ parsable and strongly lexicalizes CFG.

4.1 A Strong Lexicalization Procedure

In the following, we give a constructive proof of the fact that TIG strongly lexicalizes CFG. This can be done using a lexicalization procedure related to the lexicalization procedure used to create Greibach normal form (GNF) as presented in [11].

Our procedure relies on the following four lemmas. The first lemma converts CFGs into a very restricted form of TIG. The next three lemmas describe ways that TIGs can be transformed without changing the trees produced.

Lemma 1 Any finitely ambiguous CFG $G = (\Sigma, NT, P, S)$ can be converted into a TIG $G' = (\Sigma, NT, I, \{\}, S)$ such that: (i) there are no auxiliary trees; (ii) no initial tree contains any interior nodes; (iii) G' generates the same trees and therefore the same strings as G; (iv) there is only one way to derive a given tree in G'.

Proof: We assume without loss of generality that G does not contain any useless production.

The set I of initial trees in G' is constructed by converting each rule R in P into a one-level tree t whose root is labeled with the left-hand side of R. If R has $n > 0$ elements on its right-hand side, then t is given n children, each labeled with the corresponding right-hand-side element. Each child labeled with a nonterminal is marked for substitution. If the right-hand side of R is empty, t is given one child labeled with ε.

By construction, there are no auxiliary trees and no interior nodes in any initial tree. There is an exact one-to-one correspondence between derivations in G and derivations using the initial trees. Each rule substitution in G becomes a tree substitution in G'. As a result, exactly the same trees are generated in both cases, and there is only one way to generate each tree in G', because there cannot be two ways to derive the same tree in a CFG. □

Lemma 2 Let $G = (\Sigma, NT, I, A, S)$ be a TIG. Let $t \in I \cup A$ be an elementary tree whose root is labeled Y and let μ be a frontier element of t that is labeled X and marked for substitution. Further suppose, that if t is an initial tree, $X \neq Y$. Let T' be the set of every tree t' that can be created by substituting an X-rooted tree $u \in I$ for μ. Define $G' = (\Sigma, NT, I', A', S)$ where I' and A' are created as follows.

$$\text{If } t \in I \text{ then } I' = (I - \{t\}) \cup T' \text{ and } A' = A.$$
$$\text{If } t \in A \text{ then } I' = I \text{ and } A' = (A - \{t\}) \cup T'.$$

Then, G' generates exactly the same trees as G. Further, if there is only one way to generate each tree generated by G, then there is only one way to generate each tree generated by G'.

Proof: The transformation specified by this lemma closes over substitution into μ and then discards t. Since t cannot be substituted into μ, this only generates a finite number of additional trees.

Any complete derivation in G can be converted into exactly one derivation in G' as follows. A derivation consists of elementary trees and operations between them. Every use of t in a complete derivation in G has to be associated with a substitution of some $u \in I$ for μ. Taken as a group, the two trees t and u, along with the substitution operation between them, can be replaced by the appropriate new tree $t' \in T'$ that was added in the construction of G'.

Since TIGs do not treat the roots of initial trees in any special way, there is no problem converting any operation applied to the root of u into an operation on the corresponding interior node of t'. Further, since it cannot be the case that $t = u$, there is no ambiguity in the mapping defined above.

Any derivation in G' can be converted into exactly one derivation in G by doing the reverse of the conversion above. Each instance t' of one of the new trees introduced is replaced by an instance of t with the appropriate initial tree $u \in I$ being combined with it by substitution.

Again since TIGs do not treat the roots of initial trees in any special way, there is no problem converting any operation applied to an interior node of t' that corresponds to the root of u into an operation on the root of u.

Further, if there is only one way to derive a given tree in G, there is no ambiguity in the mapping from derivations in G' to G, because there is no ambiguity in the mapping of T' to trees in G. The tree t' must be different from the other trees generated when creating T', because t' contains complete information about the trees it was created from. The tree t' must not be in $I \cup A$. If it were, there would be multiple derivations for some tree in G, one involving t' and one involving t and u. Finally, t' must be different from t, because it must be larger than t.

If there is only one way to derive a given tree in G, the mappings between derivations in G' and G are one-to-one and there is therefore only one way to derive a given tree in G'. □

Lemma 3 Let $G = (\Sigma, NT, I, A, S)$ be a TIG. Let $t \in I$ be an elementary initial tree whose root is labeled with $X \neq S$. Further suppose that none of the substitution nodes, if any, on the fringe of t are labeled X. Let U' be the set of every initial tree that can be created by substituting t for one or more frontier nodes in an initial tree $u \in I$ that are labeled X and marked for substitution. Let V' be the set of every auxiliary tree that can be created by substituting t for one or more frontier nodes in an auxiliary tree $v \in A$ that are labeled X and marked for substitution. Define $G' = (\Sigma, NT, I', A', S)$ where $I' = (I - \{t\}) \cup U'$ and $A' = A \cup V'$.

Then, G' generates exactly the same trees as G. Further, if there is only one way to generate each tree generated by G, then there is only one way to generate each tree generated by G'.

Proof: The transformation specified by this lemma closes over substitution of t and then discards t. Since t cannot be substituted into itself, this only generates a finite number of additional trees. Since the root of t is not labeled S, t is not required for any purpose other than substitution.

Any complete derivation in G can be converted into exactly one derivation in G' as follows. Since the root of t is not labeled S, every use of t in a complete derivation in G has to be substituted into some frontier node μ of some $u \in I \cup A$. Taken as a group, the two trees u and t, along with any other copies of t substituted into other frontier nodes of u and the substitution operations between them, can be replaced by the appropriate new tree $u' \in U' \cup V'$ that was added in the construction of G'.

Since TIGs do not treat the roots of initial trees in any special way, there is no problem converting any operation applied to the root of t into an operation on the corresponding interior node of u'. Further, since it cannot be the case that $t = u$, there is no ambiguity in the mapping defined above.

Any derivation in G' can be converted into a derivation in G by doing the reverse of the conversion above. Each instance u' of one of the new trees introduced is replaced by one or more instances of t substituted into the appropriate tree $u \in I \cup A$.

Again since TIGs do not treat the roots of initial trees in any special way, there is no problem converting any operation applied to the interior node of u' that corresponds to the root of t into an operation on the root of t.

Further, if there is only one way to derive a given tree in G, there is no ambiguity in the mapping from derivations in G' to G, because there is no ambiguity in the mapping of u' to trees in G. The tree u' must be different from the trees that are generated by substituting t in other trees u, because u' contains complete information about the trees it was created from. The tree u' must not be in $I \cup A$. If it were, there would be multiple derivations for some tree in G, one involving u' and one involving u and t. Finally, u' must be different from t, because it must be larger than t.

If there is only one way to derive a given tree in G, the mappings between derivations in G' and G are one-to-one and there is therefore only one way to derive a given tree in G'. □

Lemma 4 Let $G = (\Sigma, NT, I, A, S)$ be a TIG and $X \in NT$ be a nonterminal. Let $T \subset I$ be the set of every elementary initial tree t such that the root of t and the leftmost nonempty frontier node of t are both labeled X. Suppose that every node labeled X where adjunction can occur is the root of an initial tree in I. Suppose also that there is no tree in A whose root is labeled X. Let T' be the set of right auxiliary trees created by marking the first nonempty frontier node of each element of T as a foot rather than for substitution. Define $G' = (\Sigma, NT, I - T, A \cup T', S)$.

Then, G' generates exactly the same trees as G. Further, if there is only one way to generate each tree generated by G, then there is only one way to generate each tree generated by G'.

Proof: Note that when converting the trees in T into trees in T', every initial tree is converted into a different auxiliary tree. Therefore there is a one-to-one mapping between trees in T and T'. Further, since there are no X-rooted trees in A, $A \cap T' = \{\}$.

Since in G, every node labeled X where adjunction can occur is the root of an initial tree in I, it must be the case that in G', every node labeled X where adjunction can occur is the root of an initial tree in I', because the construction of T' did not create any new nodes labeled X where adjunction can occur. Therefore, the only way that any element of T' can be used in a derivation in G' is by adjoining it on the root of an initial tree u. The effect of this adjunction is exactly the same as substituting the corresponding $t \in I$ in place of u and then substituting u for the first nonempty frontier node of t.

Any complete derivation in G can be converted into exactly one derivation in G' as follows. Every instance of a tree in T has to occur in a substitution chain as follows. The chain consists of some number of instances $t_1, t_2, \ldots, t_m$ of trees in T with each tree substituted for the leftmost nonempty frontier node of the next. The top of the chain t_m is either not substituted anywhere (i.e., only if $X = S$) or substituted at a node that is not the leftmost nonempty node of a tree in T. The bottom tree in the chain t_1 has some tree $u \notin T$ substituted for its leftmost nonempty frontier node. Since there are no X-rooted trees in A, there cannot be any adjunction on the root of u or on the roots of any of the trees in the chain. The chain as a whole can be replaced by the simultaneous adjunction of the corresponding trees $t'_1, t'_2, \ldots, t'_m$ in T' on the root of u, with u used in the same way that t_m was used.

Any derivation in G' can be converted into a derivation in G by doing the reverse of the conversion above. Each use of a tree in T' must occur as part of the simultaneous adjunction of 1 or more auxiliary trees on the root of some initial tree u, because there are no other nodes at which this tree can be adjoined. Since the trees in T' are the only X-rooted trees in A', all the trees being simultaneously adjoined must be instances of trees in T'. The simultaneous adjunction can be replaced with a substitution chain combining the corresponding trees in T, with u substituted into the tree at the bottom of the chain and the top of the chain used however u was used.

Further, if there is only one way to derive a given tree in G, there is no ambiguity in the mapping from derivations in G' to G, because there is no ambiguity in the mapping of the t'_i to trees in G. If there is only one way to derive a given tree in G, the mappings between derivations in G' and G are one-to-one and there is therefore only one way to derive a given tree in G'. □

After an application of Lemmas 2–4, a TIG may no longer be in reduced form; however, it can be brought back to reduced form by discarding any unnecessary elementary trees. For instance, in Lemma 2, if μ is the only substitution node labeled X and $X \neq S$, then when t is discarded, every X-rooted initial tree can be discarded as well.

Using the above lemmas, an LTIG corresponding to a CFG can be constructed as follows.

Theorem 2 If $G = (\Sigma, NT, P, S)$ is a finitely ambiguous CFG that does not generate the empty string, then there is an LTIG $G' = (\Sigma, NT, I', A', S)$ generating the same language and tree set as G with each tree derivable in only one way. Furthermore, G' can be chosen so that all the auxiliary trees are right auxiliary trees and every elementary tree is left anchored.

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

The proof proceeds in four steps. At each step none of the modifications made to the grammar change the tree set produced nor introduce more than one way to derive any tree. Therefore, the degree of ambiguity of each string is preserved by the constructed LTIG.

An ordering $\{A_1, \ldots, A_m\}$ of the nonterminals NT is assumed.

Step 1: Using Lemma 1, we first convert G into an equivalent TIG $(\Sigma, NT, I, \{\}, S)$, generating the same trees. Because G does not contain any empty rules, the set of initial trees created does not contain any empty trees.

Step 2: In this step, we modify the grammar of Step 1 so that every initial tree $t \in I$ satisfies the following property Ω. Let the label of the root of t be A_i. The tree t must either:

(i) be left anchored, i.e., have a terminal as its first nonempty frontier node; or
(ii) have a first nonempty frontier node labeled A_j where $i < j$.

We modify the grammar to satisfy Ω inductively for increasing values of i.

Consider the A_1-rooted initial trees that do not satisfy Ω. Such trees must have their first nonempty frontier node labeled by A_1. These initial trees are converted into right auxiliary trees as specified by Lemma 4. The applicability of Lemma 4 in this case is guaranteed since, after Step 1, there are no auxiliary trees, no interior nodes, and TIG prohibits adjunction at frontier nodes.

We now assume inductively that Ω holds for every A_i rooted initial tree t where $i < k$.

- Step 2a: Consider the A_k-rooted initial trees that fail to satisfy Ω. Each one must have a first nonempty frontier node μ labeled with A_j where $j \leq k$. For those where $j < k$, we generate a new set of initial trees by substituting other initial trees for μ in accordance with Lemma 2.

  By the inductive hypothesis, the substitutions specified by Lemma 2 result in trees that are either left anchored, or have first nonempty frontier nodes labeled with A_l where $l > j$. For those trees where $l < k$, substitution as specified by Lemma 2 is applied again.

  After at most $k-1$ rounds of substitution, we reach a situation where every A_k-rooted initial tree that fails to satisfy Ω has a first nonempty frontier node labeled by A_k.

- Step 2b: The A_k-rooted initial trees where the first nonempty frontier node is labeled with A_k are then converted into right auxiliary trees as specified by Lemma 4. The applicability of Lemma 4 in this situation is guaranteed by the following. First, there cannot have previously been any A_k-rooted auxiliary trees, because there were none after Step 1, and every auxiliary tree previously introduced in this induction has a root labeled A_i for some $i < k$. Second, there cannot be any internal nodes in any elementary tree labeled A_k, because there were none after Step 1, and all subsequent substitutions have been at nodes labeled A_i where $i < k$.

Steps 2a and 2b are applied iteratively for each i, $1 < i \leq m$ until every initial tree satisfies Ω.

Step 3: In this step, we modify the set of initial trees further until every one is left anchored. We modify the grammar to satisfy this property inductively for decreasing values of i.

According to property Ω, every A_m-rooted initial tree is left anchored, because there are no higher indexed nonterminals.

We now assume inductively that every A_i rooted initial tree t where $i > k$, t is left anchored.

The A_k rooted initial-trees must be left anchored, or have leftmost nonempty frontier nodes labeled with A_j, where $j > k$. When the label is A_j, we generate new initial trees using Lemma 2. These new rules are all left anchored, because by the induction hypothesis, all the trees u substituted by Lemma 2 are left anchored.

The above is repeated for each i until $i = 1$ is reached.

Step 4: Finally, consider the auxiliary trees created above. Each is a right auxiliary tree. If an auxiliary tree t is not left anchored, then the first nonempty frontier element after the foot is labeled with some nonterminal A_i. There must be some nonempty frontier element after the foot of t because G is not infinitely ambiguous. We can use Lemma 2 yet again to replace t with a set of left anchored right auxiliary trees. All the trees produced must be left anchored because all the initial trees resulting from Step 3 are left anchored.

Empty rules: The auxiliary assumption that G does not contain empty rules can be dispensed with as follows.

If G contains empty rules, then the TIG created in Step 1 will contain empty trees. These trees can be eliminated by repeated application of Lemma 3 as follows.

Let t be an empty tree. Since G does not derive the empty string, the label of the root of t is not S. The tree t can be eliminated by applying Lemma 3. This can lead to the creation of new empty trees. However, these can be eliminated in turn using Lemma 3. This process must terminate because G is finitely ambiguous.

Mark all the interior nodes in all the initial trees created by Lemma 3 as nodes where adjunction cannot occur. With the inclusion of these adjoining constraints, the procedure above works just as before. □

In the worst case, the number of elementary trees created by the LTIG procedure above can be exponentially greater than the number of production rules in G. This explosion in numbers comes from the compounding of repeated substitutions in Steps 2 & 3.

However, as noted at the end of Section 3, counting the number of elementary trees is not an appropriate measure of the size of an LTIG. The compounding of substitutions in the LTIG procedure causes there to be a large amount of sharing between the elementary trees. Taking advantage of this sharing can counteract the exponential growth in the number of rules completely. In particular, if the CFG does not have any empty rules or sets of mutually left recursive rules involving more than one nonterminal, then the size of the LTIG created by the procedure of Theorem 2 will be smaller than the size of the original CFG.

On the other hand, if a grammar has many sets of mutually left recursive rules involving more than one nonterminal, even taking advantage of sharing cannot stop an exponential explosion in the size of the LTIG. In the worst case, a grammar with m nonterminals can have $m!$ sets of mutually left recursive rules, and the result LTIG will is enormous.

Example. Figure 12 illustrates the operation of the LTIG procedure. Step 1 of the procedure converts the CFG at the top of the figure to the TIG on the shown on the second line.

CFG

$$\begin{array}{lcl} A_1 & \rightarrow & A_2A_2 \\ A_2 & \rightarrow & A_1A_2 \mid A_2A_1 \mid a \end{array}$$

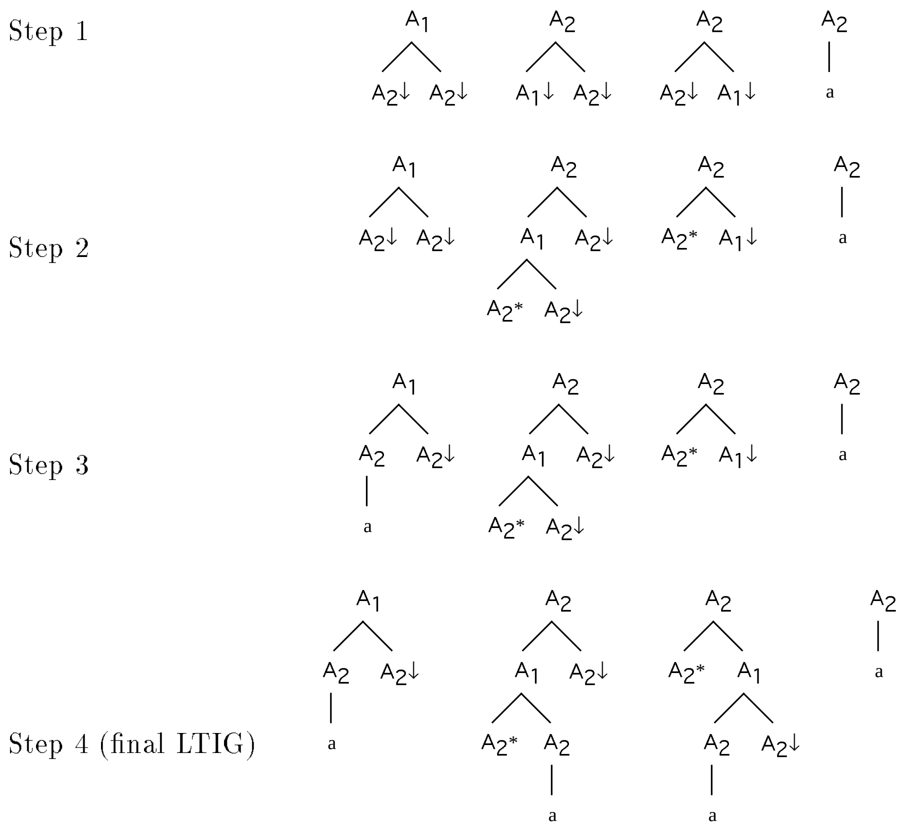


Figure 12: Example of the operation of the LTIG procedure.

In Step 2, no change is necessary in the A_1 initial tree. However, the first A_2 initial tree has the A_1 initial tree substituted into it. After that, the first two A_2 initial trees are converted into auxiliary trees as shown on the third line of Figure 12.

In step 3, the A_1 initial tree is lexicalized by substituting the remaining A_2 initial tree into it. Step 4 creates the final LTIG by lexicalizing the auxiliary trees. The A_1 initial tree is retained under the assumption that A_1 is the start symbol of the grammar.

TIG Strongly Lexicalizes LTIG. It has been shown [13, 20] that TAG extended with adjoining constraints not only strongly lexicalizes CFG, but itself as well. We conjecture that our construction can be extended so that given any TIG as input, an LTIG generating the same trees could be produced. As in the case for TAG, adjoining constraints forbidding the adjunction of specific auxiliary trees on specific nodes can be required in the resulting LTIG.

4.2 Comparison of the LTIG, Greibach, and Rosenkrantz Procedures

The LTIG procedure of Theorem 2 is related to the procedure traditionally used to create GNF (see e.g., [11]). In particular, the main part of the GNF procedure operates in three steps that are similar to Steps 2, 3, & 4. However, there are three important differences between the procedures.

CFG
$$\begin{array}{lcl} A_1 & \rightarrow & A_2A_2 \\ A_2 & \rightarrow & A_1A_2 \mid A_2A_1 \mid a \end{array}$$

Step 2
$$\begin{array}{lcl} A_1 & \rightarrow & A_2A_2 \\ A_2 & \rightarrow & aZ_2 \mid a \\ Z_2 & \rightarrow & A_1 \mid A_2A_2 \mid A_2A_2Z_2 \mid A_1Z_2 \end{array}$$

Step 3
$$\begin{array}{lcl} A_1 & \rightarrow & aA_2 \mid aZ_2A_2 \\ A_2 & \rightarrow & aZ_2 \mid a \\ Z_2 & \rightarrow & A_2A_2 \mid A_1 \mid A_2A_2Z_2 \mid A_1Z_2 \end{array}$$

GNF
$$\begin{array}{lcl} A_1 & \rightarrow & aA_2 \mid aZ_2A_2 \\ A_2 & \rightarrow & aZ_2 \mid a \\ Z_2 & \rightarrow & aA_2 \mid aA_2Z_2 \mid aZ_2A_2 \mid aZ_2A_2Z_2 \end{array}$$

Figure 13: Example of the operation of the GNF procedure.

First, the LTIG procedure maintains both the ambiguity of the original grammar and the exact trees produced. In contrast, the GNF procedure can reduce the ambiguity of the grammar and in general, introduces radical changes in the trees produced.

Second, in lieu of Step 1, the GNF procedure converts the input into Chomsky normal form. This eliminates infinite ambiguity and empty rules, and puts the input grammar in a very specific form.

The elimination of infinite ambiguity is essential, because the GNF procedure will not operate if infinite ambiguity is present. The elimination of empty rules is also essential, because empty rules in the input to the rest of the GNF procedure lead to empty rules in the output.

However, the remaining changes caused by putting the input in Chomsky normal form are irrelevant to the basic goal of creating a left anchored output. A more compact left anchored grammar can typically be produced by eliminating infinite ambiguity and empty rules without making the other changes necessary to put the input in Chomsky normal form. In the following discussion we assume a modified version of the GNF procedure that takes this approach.

The third important difference between the LTIG and GNF procedures is the way they handle left recursive rules. The LTIG procedure converts them into right auxiliary trees. In contrast, the GNF procedure converts them into right recursive rules. That is to say, the GNF procedure converts rules of the form $A_k \rightarrow A_k\alpha \mid \beta$ into rules of the form $A_k \rightarrow \beta|\beta Z_k$ and $Z_k \rightarrow \alpha|\alpha Z_k$.

Figure 13 illustrates the operation of the GNF procedure when applied to the same CFG as in Figure 12. Since the input grammar is finitely ambiguous and has no empty rules, it can be operated on as is.

The step of the GNF procedure corresponding to Step 2 of the LTIG procedure converts the CFG at the top of Figure 13 into the rules shown in the second part of the figure. No change is necessary in the A_1 rule. However, the first A_2 rule has the A_1 rule substituted into it. After that, the left recursive A_2 rules are converted into right recursive rules utilizing a new non-terminal Z_2.

The step of the GNF procedure corresponding to Step 3 of the LTIG lexicalizes the A_1 rule by substituting the A_2 rules into it.

The final step of the GNF procedure lexicalizes the Z_2 rules as shown at the bottom of Figure 13. An important thing to notice is that this lexicalization only results in 4 Z_2 rules,

$$\begin{array}{lcl} A_1 & \rightarrow & aZ_2A_2 \\ A_2 & \rightarrow & aZ_2 \\ Z_2 & \rightarrow & aZ_2A_2Z_2 \mid \varepsilon \end{array}$$

Figure 14: The LTIG of Figure 12 converted into a CFG.

rather than the 8 one would expect, because the 8 possible ways of substituting an A_1 or A_2 rule into the first position of a Z_2 rule only yield 4 distinct rules. For example, substituting $A_1 \rightarrow aA_2$ into $Z_2 \rightarrow A_1$ yields the same result as substituting $A_2 \rightarrow a$ into $Z_2 \rightarrow A_2A_2$. This kind of collapsing of identical rules derived in different ways is the reason why the GNF procedure can reduce the ambiguity of a grammar.

It is interesting to note that if the LTIG created in Figure 12 is converted into a CFG as specified in Theorem 1, this results in the rules in Figure 14. Ambiguity is lost in this transformation, because both auxiliary trees turn into the same rule. If the empty rule in Figure 14 is eliminated by substitution, the exact same grammar as at the bottom of Figure 13 results.

We conjecture that there is, in general, an exact correspondence between the output of the LTIG procedure and the GNF procedure. In particular, if (a) the LTIG procedure is applied to a CFG in Chomsky normal form, (b) the LTIG is converted into a CFG as specified by Theorem 1, and (c) any resulting empty rules are eliminated by substitution, the result is always the same CFG as the GNF procedure produces. Eliminating empty rules by substitution can yield an exponential increase in the number of rules. Therefore, the output of the LTIG procedure can have exponentially fewer elementary trees than there are rules in the corresponding GNF.

The LTIG procedure can be looked at as making essentially the same transformation as the GNF procuedures, but doing so in a way that (a) records exactly where each new rule comes from so that ambiguity is preserved and the original parse trees can be recovered, and (b) represents the new rules in a way that allows them to be compactly represented by taking full advantage of their very repetitive structure.

The Rosenkrantz procedure. Another interesting point of comparison with the LTIG procedure is the CFG lexicalization procedure of Rosenkrantz [19]. This operates in a completely different way than Greibach's procedure—simultaneously eliminating all leftmost derivation paths of length greater than one, rather than shortening derivation paths one step at a time via substitution and eliminating left recursive rules one nonterminal at a time.

One consequence of the simultaneous nature of the Rosenkrantz procedure is that one needs not select an order of the nonterminals. This contrasts with the Greibach and LTIG procedures where the order chosen can have a significant impact on the number of elementary structures in the result.

As with the GNF procedure, ambiguity can be reduced and the trees derived are changed. However, the trees are changed even more radically than is the case with the GNF procedure.

Also like the GNF procedure, one typically begins the Rosenkrantz procedure by converting the input to Chomsky normal form. This is necessary to remove infinite ambiguity and empty rules. However, it is also needed to remove chain rules, which would otherwise lead to non-lexicalized rules in the output. The conversion to Chomsky normal form makes a lot of other changes as well, which are largely counterproductive if one wants to construct a left anchored grammar.

A key advantage of the Rosenkrantz procedure is that unlike the Greibach and LTIG procedures, the output it produces cannot be exponentially larger than the input. In particular, the growth in the number of rules is at worst $O(m^5)$, where m is the number of nonterminals. However, as a practical matter, the Rosenkrantz procedure produces grammars that are much less compact than those created by the LTIG procedure, see Section 5.1.

It would be develop a formalism and procedure that bares the same relationship to the Rosenkrantz procedure that TIG and the LTIG procedure bare to the GNF procedure. Given the fundamental advantages of the Rosenkrantz over the GNF procedure, this might lead to a result that was superior to the LTIG procedure.

4.3 Variants of the LTIG Procedure

The LTIG procedure above creates a left anchored LTIG that uses only right auxiliary trees. As shown in Section 5.2, this is a quite advantageous form. However, other forms might be more advantageous in some situations. Many variants of the LTIG procedure are possible.

For example, everywhere in the procedure, the word 'right' can be replaced by 'left' and vice versa. This results in the creation of a right anchored LTIG that uses only left auxiliary trees. This could be valuable when processing a language with a fundamentally left recursive structure.

A variety of steps can be taken to reduce the number of elementary trees produced by the LTIG procedure. To start with, the choice of an ordering $\{A_1, \ldots, A_m\}$ for the nonterminals is significant. In the presence of sets of mutually left recursive rules involving more than one nonterminal (i.e., sets of rules of the form $\{A \rightarrow B\beta,\ B \rightarrow A\alpha\}$), choosing the best ordering of the relevant nonterminals can greatly reduce the number of trees produced.

If one abandons the requirement that the grammar must be left anchored, one can sometimes reduce the number of elementary trees produced dramatically. The reason for this is that instead of being forced to lexicalize each rule in G at the first position on its right hand side, one is free to choose the position that minimizes the total number of elementary trees eventually produced. However one must be careful to meet the requirements imposed by TIG while doing this. In particular, one must create only left and right auxiliary trees as opposed to wrapping auxiliary trees. The search space of possible alternatives is so large that it is not practical to find an optimal LTIG; however, by means of simple heuristics and hill climbing, significant reductions in the number of elementary trees can be obtained.

Finally, one can abandon the requirement that there be only one way to derive each tree in the LTIG. This approach is discussed in [25]. In the presence of sets of mutually left recursive rules involving more than one nonterminal, allowing increased ambiguity can yield significant reduction in the number of trees.

Exploring ways to create LTIGs with small numbers of elementary trees is interesting; however, it may not be of practical significance, because there are so many opportunities for sharing between the elementary trees in the LTIGs created by the LTIG procedure, that the grammar size $|G|$ is often exponentially smaller than the number of elementary trees. Further, if an increased number of elementary trees is accompanied by increased sharing, this can lead to a decrease in the grammar size, rather than an increase.

5 Experimental Results

The experiments below use eight grammars for fragments of English as test cases, see Figure 15. The first four grammars are the test CFGs used by Tomita [30]. The next three grammars are derived from the Treebank corpus [3] of hand-parsed sentences from the Wall Street Journal. Each "Treebank n" grammar corresponds to the n most commonly occurring tree levels in the corpus that form a CFG with no useless productions. The eighth grammar is a CFG grammar used in the natural language processing component of a simple interactive computer environment. It supports conversation with an animated robot called Mike [18].

The grammars are all finitely ambiguous and none generates the empty string. The Tomita III grammar contains an empty rule. The relative size and complexity of the grammars is indicated at the top of Figure 15. The size $|G|$ is computed as appropriate for a Earley-style CFG parser—i.e., as the number of possible dotted rules, which is the sum over all the rules of: one plus the number of elements on the right-hand side of the rule.

The bottom of Figure 15 summarizes the left and right recursive structure of the test grammars. The grammars have very few sets of mutually left recursive rules involving more than one nonterminal. In contrast, all but the smallest grammars have many sets of mutually right recursive rules involving significant numbers of different nonterminals. This reflects the fact that English is primarily right recursive in nature.

Due to the unbalanced recursive nature of the test grammars, left anchored lexicalizations are much more compact than right anchored ones. For languages that are primarily left recursive in nature, the situation would be reversed.

The experiments below are based on parsing a corpus of randomly generated sentences. For each test grammar, four sentences were generated of each possible length from 1–25. The top of Figure 16 shows the average number of parses of these sentences versus sentence length. The

	Nonterminals	Terminals	Rules	Size
Tomita I	5	4	8	22
Tomita II	13	9	43	133
Tomita III	38	54	224	679
Tomita IV	45	32	394	1,478
Treebank 200	11	31	200	689
Treebank 500	14	36	500	1,833
Treebank 1000	16	36	1,000	3,919
Mike	25	102	145	470

	Left Cycles of Length			Right Cycles of Length		
	1	2	> 2	1	2-9	> 9
Tomita I	2	0	0	0	1	0
Tomita II	7	0	0	8	3	0
Tomita III	10	0	0	11	2,260	12,595
Tomita IV	13	0	0	11	3,453	5,964
Treebank 200	5	0	0	5	15	0
Treebank 500	9	1	0	9	945	44
Treebank 1000	11	2	0	10	14,195	5,624
Mike	0	0	0	1	1	0

Figure 15: Properties of the Grammars used as test cases.

	1-5	6-10	11-15	16-20	21-25
Tomita I	1	4	25	174	3,696
Tomita II	1	2	3	50	46
Tomita III	1	2	6	66	58
Tomita IV	1	11	25	140	624
Treebank 200	1	1	3	8	36
Treebank 500	1	4	20	218	1,721
Treebank 1000	2	36	1,376	23,106	279,656
Mike	1	1	1	1	1

	1-5	6-10	11-15	16-20	21-25
Tomita I	23	51	88	135	205
Tomita II	145	308	461	698	898
Tomita III	304	577	1,026	1,370	1,788
Tomita IV	827	1,436	2,311	3,192	4,146
Treebank 200	526	1,054	1,500	2,171	2,717
Treebank 500	1,193	2,762	4,401	6,712	8,566
Treebank 1000	3,795	8,301	15,404	23,689	32,633
Mike	124	163	264	334	435

Figure 16: Properties of the sentences used as test cases versus sentence length. Top: average ambiguity. Bottom: average chart size.

ambiguity varies by 5 orders of magnitude across the test corpus.

The bottom of Figure 16 shows the average number of chart states created when parsing the test sentences using a standard Earley-style CFG parser. As is to be expected, the number of chart states rises strongly with the complexity of the grammars varying by two orders of magnitude. The number of chart states also grows with the length of the sentences, but not much faster than linearly.

5.1 The Size of LTIG Grammars

The top of Figure 17 shows the number of elementary initial and auxiliary trees in grammars created by the LTIG procedure given the various test grammars. Because most of the test grammars do not have sets of mutually left recursive rules involving more than one nonterminal, the order chosen for the nonterminals typically has no effect on the output. However, for the grammars where there is an effect, the ordering that lead to the smallest number of elementary trees was automatically chosen.

The middle portion of the table summarizes the left anchored LTIGs created by the procedure of Theorem 2. The rightmost portion summarizes an unconstrained LTIGs created by a hill climbing algorithm that attempts to minimize the number of elementary trees produced. It can be seen that the left anchored LTIG corresponding to a CFG can have many more elementary trees than an unconstrained LTIG.

The bottom of Figure 17 shows the sizes of the various LTIGS. The sizes are very much smaller than the numbers of trees, because there is an extremely large amount of sharing between the elementary structures in the LTIGs. In fact, there is so much sharing that the LTIGs are smaller than the corresponding CFGs.

	CFG	Left LTIG		LTIG	
	Rules	Initial	Auxiliary	Initial	Auxiliary
Tomita I	8	6	2	5	1
Tomita II	43	905	7	87	8
Tomita III	224	1,790	45	522	51
Tomita IV	394	40,788	469	1,456	201
Treebank 200	200	648	77	284	76
Treebank 500	500	9,558	4,497	794	698
Treebank 1000	1,000	1,050,343	667,972	2,792	3,306
Mike	145	626	0	267	0

	CFG	Left LTIG	LTIG
Tomita I	22	16	21
Tomita II	133	115	125
Tomita III	679	528	665
Tomita IV	1,478	1,263	1,438
Treebank 200	689	517	677
Treebank 500	1,833	1,427	1,801
Treebank 1000	3,919	3,146	3,839
Mike	470	356	470

Figure 17: Properties of LTIGS corresponding to the test grammars.
Top: numbers of elementary trees. Bottom: grammar size $|G|$.

It is interesting to note that the left anchored LTIGs are even smaller than the unconstrained LTIGs. This is possible because of the small number of sets of mutually left recursive rules involving more than one nonterminal in the test grammars. If there were many such sets, the left anchored LTIGs could be much larger than the unconstrained ones; and it might be fruitful to consider using a right anchored LTIG. If there were many sets of mutually left recursive rules and many sets of mutually right recursive rules, then every LTIG might be large. Therefore, the practical utility of LTIG is limited to languages that are fundamentally left recursive or right recursive, but not both.

The GNF and Rosenkrantz procedures. As a basis for comparison with the LTIG procedure, the GNF and Rosenkrantz procedures were implemented as well. To minimize the size of the grammars produced by these latter procedures, the input grammars were not converted to Chomsky normal form, but rather only modified to the minimal extent required by the procedures, see Section 4.2. This yielded savings that were almost always significant and sometimes dramatic. In the case of the GNF procedure, the order of nonterminals was chosen so as to minimize the number of rules produced.

The top of Figure 18 compares the grammars produced by the three procedures in terms of the number of elementary structures. Except for Treebank 200, the Rosenkrantz procedure created fewer rules than the GNF procedure and on the larger grammars dramatically fewer. The LTIG procedure created somewhat fewer elementary structures than the Rosenkrantz procedure, except that for Treebank 1000, the LTIG has 13 times more elementary structures than the Rosenkrantz grammar. Assumedly, the large size of the LTIG for Treebank 1000 reflects the fundamentally exponential behavior of the LTIG procedure in comparison to the polynomial

	CFG	Left LTIG	Rosenkrantz	GNF
Tomita I	8	8	16	19
Tomita II	43	912	861	10,848
Tomita III	224	1,835	3,961	4,931
Tomita IV	394	41,257	45,834	243,374
Treebank 200	200	725	2,462	1,723
Treebank 500	500	14,055	20,896	149,432
Treebank 1000	1,000	1,718,315	133,170	$> 10^8$
Mike	145	626	656	843

	CFG	Left LTIG	Rosenkrantz	GNF
Tomita I	22	16	54	68
Tomita II	133	115	3,807	100,306
Tomita III	679	528	16,208	29,622
Tomita IV	1,478	1,263	257,206	2,461,556
Treebank 200	689	517	11,104	9,546
Treebank 500	1,833	1,427	106,812	1,591,364
Treebank 1000	3,919	3,146	766,728	$> 10^9$
Mike	470	356	2,439	4,384

Figure 18: Comparison of the LTIG, Rosenkrantz, and GNF procedures.
Top: number of elementary structures. Bottom: grammar size.

behavior of the Rosenkrantz procedure.

The bottom of Figure 18 takes sharing into account and compares the sizes of the various grammars. It reveals that the LTIGs are very much more compact than the other grammars, particularly for the larger test grammars.

The entries in Figure 18 for the Treebank 1000 GNF grammar are only approximate, because this grammar is too large to be practically computed given the facilities available to the authors. We had to estimate the number of rules based on the number of substitutions called for by the GNF procedure.

5.2 Parsing with LTIG

To evaluate parsing with LTIG, three experimental parsers were implemented using the deductive engine developed by Shieber, Schabes, and Pereira [27]. The test grammars were parsed using a standard Earley-style CFG parser. The grammars created by the Greibach and Rosenkrantz procedures, were parsed using an Earley-style CFG parser adapted to take full advantage of left anchored CFG grammars. The grammars produced by the LTIG procedure were parsed with the parser of Section 3 extended in all the ways discussed in Section 3.1 so that it takes full advantage of sharing and the left anchored nature of these LTIGs. Every effort was extended to make the three parsers as identical as possible, so that any differences in parsing would be due to the grammars used, rather than the parsers.

The top of Figure 19 compares the number of chart states required when parsing using the various grammars. The numbers are averages over all the test sentences of the ratio of the number of chart states created using various grammars to the chart states created when parsing using the original CFG.

	CFG	Left LTIG	Rosenkrantz	GNF
Tomita I	1.00	0.69	0.94	0.95
Tomita II	1.00	0.31	0.39	2.14
Tomita III	1.00	0.09	0.08	0.13
Tomita IV	1.00	0.14	0.28	
Treebank 200	1.00	0.12	0.15	0.53
Treebank 500	1.00	0.13	0.27	
Treebank 1000	1.00	0.19		
Mike	1.00	0.21	0.17	0.19

	CFG	Left LTIG	Rosenkrantz	GNF
Tomita I	1.0	1.0	1.0	1.0
Tomita II	1.0	1.0	1.0	1.0
Tomita III	1.0	1.0	0.7	0.7
Tomita IV	1.0	1.0	0.8	
Treebank 200	1.0	1.0	1.0	0.9
Treebank 500	1.0	1.0	0.8	
Treebank 1000	1.0	1.0		
Mike	1.0	1.0	1.0	1.0

Figure 19: Parsing properties of LTIG, Rosenkrantz, and GNF grammars.
Top: relative chart sizes. Bottom: relative ambiguity.

Chart states are used as a basis for comparison instead of parsing times, because they can be more reliably and repeatably obtained than parsing times and because they allow the easy comparison of parsers implemented using different technologies. Chart states should be a particularly accurate basis for comparison in this case, because the overhead per chart element is essentially identical for the three parsers being compared.

The third column in the table at the top of Figure 19 shows that in all cases, parsing with LTIG requires fewer chart states than parsing with the original CFG. Except for the Tomita I grammar, which is a toy example, the reduction is by a factor of at least 3 and typically in the range of 5–10. This benefit is obtained without changing the trees produced and without increasing the grammar size. The benefit is as great or greater for large grammars like Tomita IV and Treebank 1000 as for small ones like Tomita II and Mike.

As a general matter, the grammars generated by the Rosenkrantz and GNF procedures also yield reductions in the number of chart states. However, the reduction is not as great as for the LTIG, and is only obtained at the cost of changing the trees produced and significantly increasing the grammar size.

With the Rosenkrantz and GNF procedures, the size of the grammar can be a significant problem in two ways. First, it can be so large that even with left anchored parsing an unreasonable large number of chart states is created. In Figure 19, this happens with the Greibach grammar for Tomita II. Second, the grammar can be too large to parse with at all. Several of the entries in Figure 19 are left blank, because using our experimental deduction-based parser, it was not possible for us to parse with grammars larger than one hundred thousand or so. It is not clear whether any practical parser could handle the grammar that the GNF procedure creates for Treebank 1000.

The bottom of Figure 19 shows the average relative ambiguity of the grammars produced by

	1-5	6-10	11-15	16-20	21-25
Tomita I	0.43	0.60	0.69	0.76	0.86
Tomita II	0.28	0.30	0.30	0.34	0.35
Tomita III	0.06	0.08	0.10	0.10	0.11
Tomita IV	0.11	0.14	0.15	0.15	0.17
Treebank 200	0.08	0.11	0.12	0.14	0.14
Treebank 500	0.08	0.11	0.13	0.16	0.16
Treebank 1000	0.10	0.15	0.21	0.25	0.33
Mike	0.14	0.23	0.21	0.22	0.21

	1	2-10	11-100	101-1000	> 1000
Tomita I	0.44	0.61	0.73	0.80	0.90
Tomita II	0.28	0.32	0.36		
Tomita III	0.06	0.09	0.13		
Tomita IV	0.11	0.13	0.16	0.18	
Treebank 200	0.09	0.13	0.15		
Treebank 500	0.07	0.12	0.15	0.18	0.20
Treebank 1000	0.08	0.13	0.17	0.22	0.30
Mike	0.20				

Figure 20: Ratio of Left LTIG to CFG chart states.
Top: versus sentence length. Bottom: versus sentence ambiguity.

the three procedures when applied to the test sentences. Each number is the average ambiguity of the sentences under the grammar in question divided by their ambiguity under the original CFG. The LTIG always has the same ambiguity as the CFG. The other procedures often create grammars with significantly less ambiguity.

The tables in Figure 20 provide a more detailed analysis of the reduction in chart states obtained via the LTIG procedure. As in the top of Figure 19, the numbers are ratios of the number of chart states created by the LTIG parser to the number of chart states created by the CFG parser, for sentences with the indicated properties.

The top of Figure 20 shows that the benefit obtained by using LTIG declines with longer sentences, but continues to be significant. The bottom of Figure 20 shows that the benefit obtained by using LTIG also declines with higher ambiguity, but not dramatically. The missing entries in the table stem from the fact that some of the grammars do not generate significant numbers of high ambiguity sentences.

6 A Future Direction

In the preceding, TIG is primarily presented as an alternative to CFG. Another perspective on TIG is as an alternative to TAG. To explore the possibilities in this regard, we investigated the extent to which the lexicalized tree adjoining grammar (LTAG) for English being developed at the University of Pennsylvania [1] is consistent with LTIG.

The current English LTAG consists of 392,001 elementary trees. These trees are all lexicalized and contain a total of 54,777 different words. At first glance, it might seem impractical to parse using such an enormous grammar expressed in any formalism. However, because the elementary trees are lexicalized and there are so many terminal symbols, only a small fraction of the elementary trees needs to be considered when parsing any one sentence. In particular, there are on average only 7 elementary trees for each word. Therefore, only on the order of 100 elementary trees need be considered when parsing any one 10-20 word sentence.

In the context of this paper, the most striking aspect of the current English LTAG is that it is very nearly an LTIG (see Figure 21). In particular, the current English LTAG contains almost 100,000 elementary left and right auxiliary trees but only 109 elementary wrapping auxiliary trees. Further, the vast majority of the ways the auxiliary trees can be used are also consistent with the restrictions imposed by TIG. The only exceptions are the small number of situations where an elementary wrapping auxiliary tree can be adjoined and the even smaller number of situations where an elementary left auxiliary tree can be adjoined on the spine of an elementary right auxiliary tree and vice versa.

Figure 21 is suggestive, but it has several shortcomings. To start with, by counting things, the figure implicitly assumes that every elementary tree and every interaction between them is equally important. It is entirely possible that some of the non-LTIG adjunctions occur very frequently and/or are linguistically essential.

More importantly, the figure considers only simple, unconstrained adjunction. However, the current English LTAG makes use of adjoining constraints and the propagation of attributes during parsing. These mechanisms would both have to be modified if the LTAG were converted into an LTIG, due to the need to switch from adjoining on the roots of auxiliary trees to multiple simultaneous adjunction. It is argued in [26] that multiple adjunction is linguistically well motivated in a number of situations. However, there are other situations where multiple adjunction presents significant difficulties.

Given the above, there is no reason to believe that it would be easy to convert the current English LTAG entirely into an LTIG. However, there is every reason to believe that it would be worthwhile to try. Given that no effort was expended to date and yet the grammar is close to an LTIG, the grammar could probably be brought much closer to an LTIG. If complete conversion is not possible, one could consider implementing a combined parser for TIG and TAG that would apply TIG parsing to the TIG subset of a TAG and full TAG parsing to the rest. For a grammar that was mostly a TIG, such a parser should be almost as fast as a TIG parser.

	Number	Incompatible With LTIG	
initial trees	294,568	0	0%
auxiliary trees	97,433	109	.11%
possible adjunctions	45,962,478,485	49,840,130	.11%

Figure 21: Most of the current LTAG for English is consistent with LTIG.

7 Conclusion

A variety of lexicalization procedures for CFG have previously been developed. However, they all have significant disadvantages. The approaches of Greibach and Rosenkrantz, which produce a CFG in Greibach Normal Form, are only weak lexicalization procedures since they do not guarantee that the same trees are produced. In addition, these approaches often produce very large output grammars. TAG allows strong lexicalization that preserves the trees derived; however, because it uses a context sensitive operation, TAG entails much larger computation costs than CFGs.

Tree insertion grammar (TIG) is a restricted form of tree adjoining grammar (TAG) that is $O(n^3)$-time parsable, generates context-free languages, and yet allows the strong lexicalization of CFG. The main results of this paper are an efficient Earley-style parser for TIG and a procedure that converts any CFG into a left anchored lexicalized TIG (LTIG) that produces the same trees with the same degree of ambiguity. By taking advantage of the sharing between trees, these LTIGs can be represented very compactly.

Experiments with grammars for subsets of English show that the corresponding LTIGs are often even smaller than the original CFG. Most importantly, by taking advantage of the left anchored nature of the LTIG, it is possible to avoid on the order of 80% of the chart states required when parsing with the original CFG. Given that the per-chart-state cost of TIG and CFG parsers are essentially identical, this should translate directly into an 80% increase in parsing speed.

A possible future use of TIG is as an alternative for TAG. TIG is not as powerful as TAG, but it includes a number of the features of TAG. Further, at least in the current English LTAG, the features of TAG that are included in TIG are used more often that the features that are not included in TIG. As a result, it may be possible to use TIG instead of TAG in some situations, thereby gaining $O(n^3)$ parsability.

The uses for TIG discussed in this paper all involve starting with an existing grammar and converting it into a TIG. An important area for further investigation is using TIG as the original formalism for constructing grammars, because TIG allows greater derivational freedom than CFG. For instance, one can require that the grammar be lexicalized, without placing any limits the parse trees produced. This can result in grammars that are better motivated linguistically and/or faster to parse.

8 Acknowledgments

We thank John Coleman, Aravind Joshi, Esther König, Fernando Pereira, Stuart Shieber, B. Srinivas, and the members of Mitsubishi Electric Research Laboratories for valuable discussions. We thank Masaru Tomita for making his test grammars available to us.

9 References

[1] Anne Abeillé, Kathleen M. Bishop, Sharon Cote, Aravind K. Joshi, and Yves Schabes. Lexicalized tags, parsing and lexicons. In *DARPA Speech and Natural Language Workshop*, Philadelphia, PA, February 1989.

[2] Yoshua Bar-Hillel. On categorial and phrase structure grammars. In *Language and Information*, chapter 5, pages 99–115. Addison-Wesley, 1964. First appeared in *the Bulletin of the Research Council of Israel*,vol. 9F (1960), pp.1-16.

[3] Eric Brill, David Magerman, Mitchell Marcus, and Beatrice Santorini. Deducing linguistic structure from the statistics of large corpora. In *DARPA Speech and Natural Language Workshop*. Morgan Kaufmann, Hidden Valley, Pennsylvania, June 1990.

[4] N. Chomsky. *Lectures on Government and Binding*. Foris, Dordrecht, 1981.

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

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

[7] G. Gazdar, E. Klein, G. K. Pullum, and I. A. Sag. *Generalized Phrase Structure Grammars*. Blackwell Publishing, Oxford, 1985. Also published by Harvard University Press, Cambridge, MA.

[8] S.L. Graham, M.A. Harrison, and W.L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

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

[10] Maurice Gross. Lexicon-grammar and the syntactic analysis of french. In *Proceedings of the* 10^{th} *International Conference on Computational Linguistics (COLING'84)*, Stanford, 2-6 July 1984.

[11] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA, 1978.

[12] Aravind K. Joshi. How much context-sensitivity is necessary for characterizing structural descriptions—Tree Adjoining Grammars. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Processing—Theoretical, Computational and Psychological Perspectives*. Cambridge University Press, New York, 1985.

[13] 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.

[14] R. Kaplan and J. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge MA, 1983.

[15] Lauri Karttunen. Radical lexicalism. Technical Report CSLI-86-68, CSLI, Stanford University, 1986. Also in *Alternative Conceptions of Phrase Structure*, University of Chicago Press, Baltin, M. and Kroch A., Chicago, 1989.

[16] 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.

[17] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol 1: Fundamentals*. CSLI, 1987.

[18] Charles Rich, Richard C. Waters, Carol Strohecker, Yves Schabes, William T. Freeman, Mark C. Torrance, Andrew R. Golding, and Michael Roth. A prototype interactive environment for collaboration and learning. Technical Report TR94-06, Mitsubishi Electric Research Laboratories, 1994.

[19] Daniel J. Rosenkrantz. Matrix equations and normal forms for context-free grammars. *Journal of the Association for Computing Machinery*, 14(3):501–507, 1967.

[20] 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.

[21] 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*, pages 21–30, Cancun, Mexico, February 1991.

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

[23] Yves Schabes and Waters R.C. Lexicalized context-free grammars. In 21st *Meeting of the Association for Computational Linguistics (ACL'93)*, pages 121–129, Columbus, Ohio, June 1993.

[24] Yves Schabes and Waters R.C. Stochastic lexicalized context-free grammars. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 257–266, Tilburg (the Netherlans), Durbuy (Belgium), August 1993.

[25] Yves Schabes and Waters R.C. Stochastic lexicalized context-free grammars. Technical Report 93-12, Mitsubishi Electric Research Laboratories, 201 Broadway. Cambridge MA 02139, 1993.

[26] Yves Schabes and Stuart Shieber. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20(1):91–124, March 1994.

[27] Stuart M. Shieber, Yves Schabes, and Fernando C.N. Pereira. Principles and implementation of deductive parsing. Technical Report TR-11-94, Harvard University, 1994.

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

[29] 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.

[30] Masaru Tomita. *Efficient Parsing for Natural Language, A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1985.

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

[32] K. Vijay-Shanker and David Weir. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636, 1993.

[33] David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

A No wrapping trees can be built in TIG

In this appendix, we give a proof that given a TIG (Σ, NT, I, A, S), it is not possible to create wrapping auxiliary trees.

Proof: The only elementary trees allowed are left auxiliary trees, right auxiliary trees and initial trees. A case by case analysis reveals that every possible combination of these kinds of trees yields a new tree in one of the three categories. Therefore, no derivation can ever create a wrapping auxiliary tree.

Substitution of an initial tree in an initial tree yields an initial tree.

Adjunction of a left or right auxiliary tree in an initial tree yields an initial tree.

Substitution of an initial tree in a left (right) auxiliary tree yields a left (right) auxiliary tree, because by definition the node marked for substitution must be left (right) of the foot and therefore all the new frontier nodes must be added left (right) of the foot.

Adjunction of a left (right) auxiliary tree S in a right (left) auxiliary tree T yields a right (left) auxiliary tree, because by definition the node adjoined upon must be to the right (left) of the spine of T and therefore all the new frontier nodes must be added right (left) of the foot of T.

Adjunction of a left (right) auxiliary tree S in a left (right) auxiliary tree T yields a left (right) auxiliary tree, for the same basic reason as above except that the node adjoined upon can be on the spine of T. However, since all the nonempty structure in S is left (right) of the spine of S, even in this case, all the new nonempty frontier nodes are added to the left (right) of the foot of T. □