The Use of Shared Forests in Tree Adjoining Grammar Parsing*

K. Vijay-Shanker
Department of Computer &
Information Sciences
University of Delaware
Newark, DE 19716
USA
vijay@udel.edu

David J. Weir
School of Cognitive &
Computing Sciences
University of Sussex
Falmer, Brighton BN1 9QH
UK
davidw@cogs.sussex.ac.uk

Abstract

We study parsing of tree adjoining grammars with particular emphasis on the use of shared forests to represent all the parse trees deriving a well-formed string. We show that there are two distinct ways of representing the parse forest one of which involves the use of linear indexed grammars and the other the use of context-free grammars. The work presented in this paper is intended to give a general framework for studying tag parsing. The schemes using lig and cfg to represent parses can be seen to underly most of the existing tag parsing algorithms.

1 Introduction

We study parsing of tree adjoining grammars (tag) with particular emphasis on the use of shared forests to represent all the parse trees deriving a well-formed string. Following Billot and Lang [1989] and Lang [1992] we use grammars as a means of recording all parses. Billot and Lang used context-free grammars (cfg) for representing all parses in a cfg parser demonstrating that a shared forest grammar can be viewed as a specialization of the grammar for the given input string. Lang [1992] extended this approach considering both the recognition problem as well as the representation of all parses and suggests how this can be applied to tag.

This paper examines this approach to tag parsing in greater detail. In particular, we show that there are two distinct ways of representing the parse forest. One possibility is to use linear indexed grammars (lig), a formalism that is equivalent to tag [Vijay-Shanker and Weir, in pressa]. The use of lig is not surprising in that we would expect to be able to represent parses of a formalism in an equivalent formalism. However, we also show that there is a second way of representing parses that makes use of a cfg.

The work presented in this paper is intended to give a general framework for studying tag parsing. The schemes using lig and cfg to represent parses can be seen to underly most of the existing tag parsing algorithms.

We begin with brief definitions of the tag and lig formalisms. This is followed by a discussion of the methods for cfg recognition and the representation of parses trees that were described in [Billot and Lang, 1989; Lang, 1992]. In the remainder of the paper we examine how this approach can be applied to tag. We first consider the representation of parses using a cfg and give the space and time complexity of recognition and extraction of parses using this representation. We then consider the same issues where lig is used as the formalism for representing parses. We conclude by comparing these results with those for existing tag parsing algorithms.

2 Tree Adjoining Grammars

Tag is a tree generating formalism introduced in [Joshi *et al.*, 1975]. A tag is defined by a finite set of *elementary* trees that are composed by means of the operations of tree adjunction and substitution. In this paper, we only consider the use of the adjunction operation.

*We are very grateful to Bernard Lang for helpful discussions.

Definition 2.1 A tag, G, is denoted

$$G = (V_N, V_T, S, I, A)$$

where
V_N is a finite set of nonterminals symbols,
V_T is a finite set of terminal symbols,
$S \in V_N$ is the start symbol,
I is a finite set of initial trees,
A is a finite set of auxiliary trees.

An **initial** tree is a tree with root labeled by S and internal nodes and leaf nodes labeled by nonterminal and terminal symbols, respectively. An **auxiliary** tree is a tree that has a leaf node (the **foot** node) that is labeled by the same nonterminal that labels the root node. The remaining leaf nodes are labeled by terminal symbols and all internal nodes are labeled by nonterminals. The path from the root node to the foot node of an auxiliary tree is called the **spine** of the auxiliary tree. An **elementary** tree is either an initial tree or an auxiliary tree. We use α to refer to initial trees and β for auxiliary trees.

A node of an elementary tree is called an **elementary node** and is named with an **elementary node address**. An elementary node address is a pair comprising of the name of the elementary tree to which the node belongs and the address of the node within that tree. We will assume the standard addressing scheme: the root node has an address ϵ; if a node with address μ has k children then the k children (in left to right order) have addresses $\mu \cdot 1, \ldots, \mu \cdot k$. Thus, for each address μ we have $\mu \in \mathcal{N}^*$ where $\mathcal{N}$ is the set of natural numbers. In this section we use μ to refer to addresses and η to refer to elementary node addresses. In general, we can write $\eta =< \gamma, \mu >$ where γ is an elementary tree and $\mu \in \mathsf{dom}(\gamma)$ and $\mathsf{dom}(\gamma)$ is the set of addresses of the nodes in γ.

Let γ be a tree with internal node labeled by a nonterminal A. Let β be an auxiliary tree with root and foot node labeled by the same nonterminal A. The tree, γ', that results from the **adjunction** of β at the node in γ labeled A is formed by removing the subtree of γ rooted at this node, inserting β in its place, and substituting it at the foot node of β.

Each elementary node is associated with a **selective adjoining** (SA) constraint that determines the set of auxiliary trees that can be adjoined at that node. In addition when adjunction is mandatory at a node it is said to have an **obligatory adjoining** (OA) constraint. Whether β can be adjoined at the node (labeled by A) in γ is determined by the SA constraint of the node. In γ' the nodes contributed by β have the same constraints as those associated with the corresponding nodes in β. The remaining nodes in γ' have the constraints of the corresponding nodes in γ.

Given $\mu \in \mathsf{dom}(\gamma)$, by $\mathsf{lbl}\langle\gamma, \mu\rangle$ we refer to the label of the node addressed μ in γ. Similarly, we will use $\mathsf{sa}\langle\gamma, \mu\rangle$ and $\mathsf{oa}\langle\gamma, \mu\rangle$ to refer to the SA and OA constraints of a node addressed μ in a tree γ. Finally, we will use $\mathsf{ft}(\beta)$ to refer to the address of the foot node of an auxiliary tree β.

$\mathsf{adj}(\gamma, \mu, \beta)$ denotes the tree that results from the adjunction of β at the node in γ with address μ. This is defined when $\beta \in \mathsf{sa}\langle\gamma, \mu\rangle$. If $\mathsf{adj}(\gamma, \mu, \beta) = \gamma'$ then the nodes in γ' are defined as follows.

- $\mathsf{dom}(\gamma') =$

$$\begin{array}{ll} & \{\mu_1 \mid \ \mu_1 \in \mathsf{dom}(\gamma) \text{ and } \\ & \qquad \mu_1 \neq \mu \cdot \mu_2 \text{ for some } \mu_2 \in \mathcal{N}^*\} \\ \cup & \{\mu \cdot \mu_1 \mid \ \mu_1 \in \mathsf{dom}(\beta)\} \\ \cup & \{\mu \cdot \mathsf{ft}(\beta) \cdot \mu_1 \mid \ \mu \cdot \mu_1 \in \mathsf{dom}(\gamma) \text{ and } \\ & \qquad \mu_1 \neq \epsilon\} \end{array}$$

- if $\mu_1 \in \mathsf{dom}(\gamma)$ such that $\mu_1 \neq \mu \cdot \mu_1$ for some $\mu_1 \in \mathcal{N}^*$, (i.e., the node in γ with address μ_1 is not equal to or dominated by the node addressed μ in γ) then
  - $\mathsf{lbl}\langle\gamma', \mu_1\rangle = \mathsf{lbl}\langle\gamma, \mu_1\rangle$,
  - $\mathsf{sa}\langle\gamma', \mu_1\rangle = \mathsf{sa}\langle\gamma, \mu_1\rangle$,
  - $\mathsf{oa}\langle\gamma', \mu_1\rangle = \mathsf{oa}\langle\gamma, \mu_1\rangle$,
- if $\mu \cdot \mu_1 \in \mathsf{dom}(\gamma')$ such that $\mu_1 \in \mathsf{dom}(\beta)$ then
  - $\mathsf{lbl}\langle\gamma', \mu \cdot \mu_1\rangle = \mathsf{lbl}\langle\beta, \mu_1\rangle$,
  - $\mathsf{sa}\langle\gamma', \mu \cdot \mu_1\rangle = \mathsf{sa}\langle\beta, \mu_1\rangle$,
  - $\mathsf{oa}\langle\gamma', \mu \cdot \mu_1\rangle = \mathsf{oa}\langle\beta, \mu_1\rangle$,
- if $\mu \cdot \mathsf{ft}(\beta) \cdot \mu_1 \in \mathsf{dom}(\gamma')$ such that $\mu \cdot \mu_1 \in \mathsf{dom}(\gamma)$ then
  - $\mathsf{lbl}\langle\gamma', \mu \cdot \mathsf{ft}(\beta) \cdot \mu_1\rangle = \mathsf{lbl}\langle\gamma, \mu \cdot \mu_1\rangle$,
  - $\mathsf{sa}\langle\gamma', \mu \cdot \mathsf{ft}(\beta) \cdot \mu_1\rangle = \mathsf{sa}\langle\gamma, \mu \cdot \mu_1\rangle$,
  - $\mathsf{oa}\langle\gamma', \mu \cdot \mathsf{ft}(\beta) \cdot \mu_1\rangle = \mathsf{oa}\langle\gamma, \mu \cdot \mu_1\rangle$,

In general, if μ is the address of a node in γ then $< \gamma, \mu >$ denotes the elementary node address of the node that contributes to its presence, and hence its label and constraints.

The tree language, $T(G)$, generated by a TAG, G, is the set of trees derived starting from an initial tree such that no node in the resulting tree has an OA constraint. The (string) language, $L(G)$, generated by a TAG, G, is the set of strings that appear on the frontier of trees in $T(G)$.

Example 2.1 Figure 1 gives a TAG, G, which generates the language $\{wcw \mid w \in \{a, b\}^*\}$. The constraints associated with the root and foot of β specify that no auxiliary trees can be adjoined at these nodes. This is indicated in Figure 1 by associating the empty set, ϕ, with these nodes. An example derivation of the strings *aca* and *abcab* is shown in Figure 2.

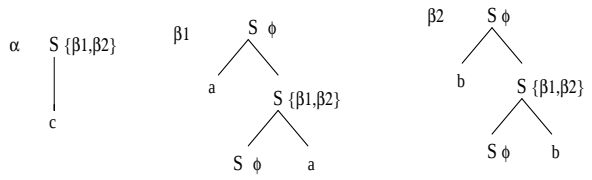

Figure 1: Example of a TAG G

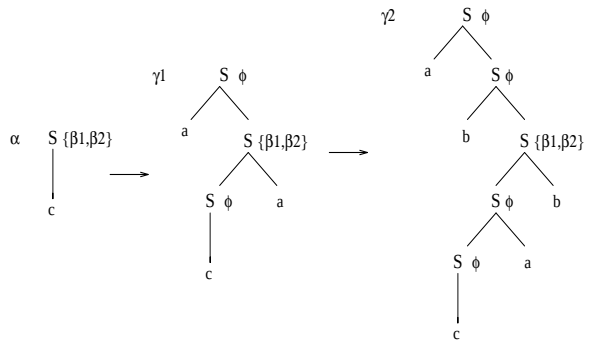

Figure 2: Sample derivations in G

3 Linear Indexed Grammars

An indexed grammar [Aho, 1968] can be viewed as a cfg in which objects are nonterminals with an associated stack of symbols. In addition to rewriting nonterminals, the rules of the grammar can have the effect of pushing or popping symbols on top of the stacks that are associated with each nonterminal. In [Gazdar, 1988] a restricted form of indexed grammars was discussed in which the stack associated with the nonterminal on the left of each production can only be associated with one of the occurrences of nonterminals on the right of the production. Stacks of bounded size are associated with other occurrences of nonterminals on the right of the production. We call this linear indexed grammars (lig). Lig generate the same class of languages as tag [Vijay-Shanker and Weir, in pressa].

Definition 3.1 A LIG, G, is denoted

$$G = (V_N, V_T, V_I, S, P)$$

where
V_N is a finite set of nonterminals,
V_T is a finite set of terminals,
V_I is a finite set of indices (stack symbols),
$S \in V_N$ is the start symbol, and
P is a finite set of productions.

Given a lig, $G = (V_N, V_T, V_I, S, P)$, we define the set of objects of G as

$$V_C(G) = \{ A[\alpha] \mid A \in V_N \text{ and } \alpha \in V_I^* \}$$

We use $A[\circ\circ\, \alpha]$ to denote the nonterminal A associated with an arbitrary stack with the string α on top and $A[]$ to denote that an empty stack is associated with A. We use Υ to denote strings in $(V_C(G) \cup V_T)^*$.

The general form of a lig production is:

$$A[\circ\circ\, \alpha] \rightarrow \Upsilon B[\circ\circ\, \alpha']\Upsilon'$$

where $A, B \in V_N$, $\alpha, \alpha' \in V_I^*$ and $\Upsilon, \Upsilon' \in (V_C(G) \cup V_T)^*$.

Given a grammar, $G = (V_N, V_T, V_I, S, P)$, the derivation relation, $\underset{G}{\Longrightarrow}$, is defined such that if

$$A[\circ\circ\, \alpha] \rightarrow \Upsilon B[\circ\circ\, \alpha']\Upsilon' \in P$$

then for every $\beta \in V_I^*$ and $\Upsilon_1, \Upsilon_2 \in (V_C(G) \cup V_T)^*$:

$$\Upsilon_1 A[\beta\alpha]\Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 \Upsilon B[\beta\alpha']\Upsilon'\Upsilon_2$$

As a result of the *linearity* in the rules, the stack $\beta\alpha$ associated with the object in the left-hand side of the derivation and the stack $\beta\alpha'$ associated with one of the objects in the right-hand side have the initial part β in common. In the derivation above, we say that the object $B[\beta\alpha']$ is the **distinguished child** of $A[\beta\alpha]$. Given a derivation, the **distinguished descendant** relation is the reflexive, transitive closure of the distinguished child relation.

The language generated by a lig, G is:

$$L(G) = \left\{ w \;\middle|\; S[\,] \underset{G}{\overset{*}{\Longrightarrow}} w \right\}$$

where $\underset{G}{\overset{*}{\Longrightarrow}}$ denotes the reflexive, transitive closure of $\underset{G}{\Longrightarrow}$.

Example 3.1 The language

$$\{ wcw \mid w \in \{a, b\}^* \}$$

is generated by the lig

$$G = (\{ S, T \}, \{ a, b, c \}, \{ \gamma_a, \gamma_b \}, S, P)$$

where P contains the following productions.

$$\begin{array}{ll} S[\circ\circ\,] \rightarrow aS[\circ\circ\, \gamma_a] & S[\circ\circ\,] \rightarrow bS[\circ\circ\, \gamma_b] \\ S[\circ\circ\,] \rightarrow T[\circ\circ\,] & T[\circ\circ\, \gamma_a] \rightarrow T[\circ\circ\,]a \\ T[\circ\circ\, \gamma_b] \rightarrow T[\circ\circ\,]b & T[\,] \rightarrow c \end{array}$$

This grammar generates the string *abcab* as follows.

$$\begin{aligned} S[\,] &\underset{G}{\Longrightarrow} aS[\gamma_a] \\ &\underset{G}{\Longrightarrow} abS[\gamma_a\gamma_b] \\ &\underset{G}{\Longrightarrow} abT[\gamma_a\gamma_b] \\ &\underset{G}{\Longrightarrow} abT[\gamma_a]b \\ &\underset{G}{\Longrightarrow} abT[\,]ab \\ &\underset{G}{\Longrightarrow} abcab \end{aligned}$$

4 Parsing as Intersection with Regular Languages

In the case of cfg parsing, [Billot and Lang, 1989; Lang, 1992] show that a cfg can be used to encode all of the parses for a given string. For example, let G_o be a grammar and let the string $w = a_1 \dots a_n$ be in $L(G_o)$. All parses for the string w can be represented by the shared forest grammar G_w. The nonterminals in G_w are of the form (A, i, j) where A is a nonterminal of G_o and $0 \leq i < j \leq n$. The construction of G_w is such that any derivation from (A, i, j) encodes a derivation

$$A \underset{G_o}{\overset{*}{\Longrightarrow}} a_{i+1} \dots a_j$$

For instance, suppose $A \rightarrow BC$ is a production in G_o that is used in the first step of a derivation of the substring $a_{i+1} \dots a_j$ from A. Corresponding to this production, G_w contains a production

$$(A, i, j) \rightarrow (B, i, k)(C, k, j)$$

for each $0 \leq i < k < j \leq n$. This can be used to encode all parses of $a_{i+1} \dots a_j$ from A where

$$B \overset{*}{\Longrightarrow} a_{i+1} \dots a_k \text{ and } C \overset{*}{\Longrightarrow} a_{k+1} \dots a_j$$

In general, corresponding to a production

$$A \rightarrow X_1 \dots X_r$$

in G_o the grammar G_w contains a production

$$(A, i_1, j_r) \rightarrow (X_1, i_1, j_1) \dots (X_r, i_r, j_r)$$

for every $i_1, j_1, \dots, i_r, j_r \in \{1, \dots, n\}$ such that for each $1 \leq k < r$ if $X_k \in V_T$ then $i_k + 1 = j_k$, otherwise $i_k + 1 \leq j_k$. Additionally, G_w includes the production

$$(a_k, k, k+1) \rightarrow a_k$$

for each $1 \leq k \leq n$.

Note that the number of nonterminals in the shared forest grammar, G_w, is $O(n^2)$ and the number of productions is $O(n^{m+1})$ where $|w| = n$ and m is the maximum number of nonterminals in the right-hand-side of a production in G_o. Therefore, if the object grammar were in Chomsky normal form, the number of productions is $O(n^3)$.

Lang [1992] extended this by showing that parsing a string w according to a grammar G can be viewed as intersecting the language $L(G)$ with the regular language $\{w\}$. Suppose we have an object context-free grammar G_o and some deterministic finite state automaton M. For the sake of simplicity, let us assume that G_o is in Chomsky normal form. The standard proof that context-free languages are closed under intersection with regular languages, constructs a context-free grammar for $L(G_o) \cap L(M)$ with a production

$$(A, p, q) \rightarrow (B, p, r)(C, r, q)$$

for each production $A \rightarrow BC$ of G_o and states p, q, r of M. Also for each terminal a the production $(a, p, q) \rightarrow a$ will be included if and only if $\delta(p, a) = q$ where δ is the transition function of M.

Lang [1992] applied this to cfg recognition as follows. Given an input, $w = a_1 \dots a_n$, define the dfa M_w such that $L(M_w) = \{w\}$. The state set of M_w is $\{0, 1, \dots, n\}$; the transition function δ is such that $\delta(i, a_{i+1}) = i + 1$ for each $0 \leq i < n$; 0 is the initial state; and n is the final state. The shared forest grammar G_w is obtained when the standard intersection construction described above is applied to G_o and M_w. Furthermore, since $L(G_w) = L(G_o) \cap L(M_w)$ and $L(M_w) = \{w\}$, we have $w \in L(G_o)$ if and only if $L(G_w)$ is not the empty set. That is, the original recognition problem can be turned into one of generating the shared forest grammar, G_w, and deciding whether the start nonterminal, $(S, 0, n)$, of G_w is an *useful* symbol, i.e., whether there is some terminal string x such that

$$(S, 0, n) \underset{G_w}{\overset{*}{\Longrightarrow}} x$$

Here S has been taken to be the start nonterminal of G_o. Note that G_w can be constructed in $O(n^3)$ time and "recognition" can also be accomplished within this time bound.

One advantage that arises from viewing parsing as intersection with regular languages is that exactly the same algorithm can be given a word net (a regular language that is not a singleton) rather than a single word as input. This could be useful if we wish to deal with ill-formed inputs.

5 Derivation versus Derived Trees in TAG

For grammar formalisms involving the derivation of trees, a tree is called a **derived** tree with respect to a given grammar if it can be derived using the rewriting rules of the grammar. A **derivation** tree of the grammar, on the other hand, is a tree that encodes the sequence of rewritings used in deriving a derived tree. In the case of cfg, a tree that is derived contains all the information about its derivation and there is no need to distinguish between derivation trees and derived trees. This is not always the case. In particular, for a tree-rewriting system like tag we need to distinguish between derived and derivation trees. In fact there are at least two ways one can encode tag derivation trees. The first (see [Vijay-Shanker, 1987]) captures the fact that derivations in tag are *context-free*, i.e., the trees that can be adjoined at a node can be determined a priori and are not dependent on the derivation history. We capture this context-freeness by giving a cfg to represent the set of all possible derivation sequences in a tag. An alternate scheme uses a tag or a lig (see [Vijay-Shanker

and Weir, in pressb]) to represent the set of all possible derivations.

We briefly consider the first scheme to show how given a tag, G_o and a string, w, context-free grammar can be used to represent shared forests. In later sections we will study the second scheme using lig for shared forests.

6 Using CFG for Shared Forests

Given a TAG,

$$G_o = (V_N, V_T, S, I, A)$$

and a string $w = a_1 \ldots a_n$ we construct a context-free grammar, G_w such that $L(G_w) \neq \phi$ if and only if $w \in L(G_o)$. Let M_w be the dfa for w described in Section 4.

Consider a tree β that has been derived from some auxiliary tree in A. Let the string on the frontier of β that is to the left of the foot node be u_l and the string to the right of the foot node be u_r. Consider the tree that results from the adjunction of β at a node in with elementary node address[1] η where v is the string on the frontier of the subtree rooted at η. After adjunction the strings u_l and u_r will appear to the left and right (respectively) of v.

Suppose that in a derivation of the string w by the grammar G_o the strings u_l and u_r form two continuous substrings w: i.e., $u_l = a_{i+1} \ldots a_p$ and $u_r = a_{q+1} \ldots a_j$ for some $0 \leq i < p \leq q < j \leq n$. Thus, according to the definition of M_w we would have $\delta(i, u_l) = p$ and $\delta(q, u_r) = j$. Hence, we can use the four states i, j, p and q of M_w to account for which parts of w are spanned by the frontier of β.

Since the string appearing at the subtree rooted at η is v then if $\delta(p, v) = q$ we have $\delta(i, u_l v u_r) = j$ and p and q identify the substring of w that is spanned by the subtree rooted at η. However, the node η may be on the spine of some auxiliary tree, i.e., on the path from the root to the foot node. In that case we will have to view the frontier of the subtree rooted at η as comprising two substrings, say v_l and v_r to the left and right of the foot node, respectively. The two states p, q of M_w are do not fully characterize the frontier of subtree rooted at η. We need four states, say p, q, r, s, where $\delta(p, v_l) = r$ and $\delta(s, v_r) = s$. Note that the four states in question only characterize the frontier of subtree rooted at η *before* the adjunction of β takes place. The four states i, j, r, s characterize the situation after adjunction of β since $\delta(i, u_l) = p$, $\delta(p, v_l) = r$ (therefore $\delta(i, u_l v_l) = p$) and $\delta(s, v_r u_r) = \delta(q, u_r) = j$.

In the shared forest cfg G_w the derivation of the string at frontier of tree rooted at η before adjunction will be captured by the use of a nonterminal of the form $(\bot, \eta, p, q, r, s)$ and the situation after adjunction will be characterized by $(\top, \eta, i, j, r, s)$. We use the symbols $\top$ and $\bot$ to capture the fact that consideration of a node involves two phases: (i) the $\top$ part where we consider adjunction at a node, and (ii) the $\bot$ part where we consider the subtree rooted at this node. Note that the states r, s are only needed when η is a node on the spine of an auxiliary tree. When this is not the case we let $r = s = -$.

Since we have characterized the frontier of β (i.e., the subtree rooted at the $root_\beta$, the root of β) by the four states i, j, p, q, we can use the nonterminal $(\top, root_\beta, i, j, p, q)$ and can capture the derivation involving adjunction of β at η by a production of the form

$$(\top, \eta, i, j, r, s) \rightarrow (\top, root_\beta, i, j, p, q)\,(\bot, \eta, p, q, r, s)$$

Without further discussion, we will give the productions of G_w. For each elementary node η do the following.

Case 1: When η is a node that is labeled by a terminal a, add the production

$$(\top, \eta, p, q, -, -) \rightarrow a$$

if and only if $\delta(p, a) = q$.

Case 2a: Let η_1 and η_2 be the children of η and the left-child η_1 dominates the foot node then add the production

$$(\bot, \eta, i, j, p, q) \rightarrow (\top, \eta_1, i, k, p, q)\,(\top, \eta_2, k, j, -, -)$$

if neither children dominate the foot node then add the production

$$(\bot, \eta, i, j, -, -) \rightarrow (\top, \eta_1, i, k, -, -)\,(\top, \eta_2, k, j, -, -)$$

Case 2b: Let η_1 and η_2 be the children of η and the right-child η_2 dominates the foot node then add the production

$$(\bot, \eta, i, j, p, q) \rightarrow (\top, \eta_1, i, k, -, -)\,(\top, \eta_2, k, j, p, q)$$

Case 3: When η is a nonterminal node that does not have an OA constraint, then to capture the fact that it is not necessary to adjoin at this node, we add

$$(\top, \eta, i, j, p, q) \rightarrow (\bot, \eta, i, j, p, q)$$

Case 4a: When η is a node where β can be adjoined and $root_\beta$ is the root node of β add the production

$$(\top, \eta, i, j, r, s) \rightarrow (\top, root_\beta, i, j, p, q)\,(\bot, \eta, p, q, r, s)$$

Case 4b: When η is the foot node of the auxiliary tree β add the production

$$(\bot, \eta, p, q, p, q) \rightarrow \epsilon$$

[1] Rather than repeatedly saying a node with an elementary node address η, henceforth we simply refer to it as the node η.

If η is the root of an initial tree then add the production

$$S \rightarrow (\top, \eta, 0, n, -, -)$$

where S is the start symbol of G_w.

Note that (cases 2a and 2b) we are assuming binary branching merely to simplify the presentation. We can use a sequence of binary cfg productions to encode situations where η has more than two children. That is, even if the object-level grammar was not binary branching, the shared forest grammar can still be.

Note that since the state set of M_w is $\{0, \ldots, n\}$, the number of nonterminals in G_o is $O(n^4)$. Since there are at most three nonterminals in a production, there are at most six states involved in a production. Therefore, the number of productions is $O(n^6)$ and construction of this grammar takes $O(n^6)$ time. Although the derivations of G_w encode derivations of the string w by G_o the specific set of terminal strings that is generated by G_w is not important. We do however have $\mathrm{L}(G_w) \neq \phi$ if and only if $w \in L(G_o)$. As before, we can determine whether $L(G_w) \neq \phi$ by checking whether the start nonterminal S is useful. Furthermore this can be detected in time and space linear to the size of the grammar. Since $w \in L(G_o)$ if and only if $L(G_w) \neq \phi$, recognition can be done in $O(n^6)$ time and space.

Once we have found all the useful symbols in the grammar we can prune the grammar by retaining only those productions that have only useful symbols. Since G_w is a cfg and since we can now guarantee that every nonterminal can derive a terminal string and therefore using any production will yield a terminal string eventually, the derivations of w in G_o can be read off by simply reading off derivations in G_w.

7 Using LIG for Shared Forests

We now present an alternate scheme to represent the derivations of a string w from a given object tag grammar G_o. In later sections show how it can be used for solving the recognition problem and how a single parse can be extracted.

The scheme presented in Section 6 that produced a cfg shared forest grammar captured the context-freeness of tag derivations. The approach that we now consider captures an alternative view of tag derivations in which a derivation is viewed as sensitive to the derivation history. In particular, the control of derivation can be captured with the use of additional stack machinery. This underlies the use of lig to represent the shared forests.

In order to understand how a lig can be used to encode a tag derivation, consider a top-down derivation in the object grammar as follows. A tag derivation can be seen as a traversal over the elementary trees beginning at the root of one of the initial trees. Suppose we have reached some elementary node η. We must first consider adjunction at η and after that we must visit each of η's subtrees from left to right. When we first reach η we say that we are in the top phase of η. The derivation lig encodes this with the nonterminal $\top$ associated with a stack whose top element is η. After having considered adjunction at η we are in the bottom phase of η. The derivation lig encodes this with the nonterminal $\bot$ associated with a stack whose top element is η.

When considering adjunction at η we may have a choice of either not adjoining at all or selecting some auxiliary tree to adjoin. If the former case we move directly to the bottom phase of η. In the latter case we move to (visit) the root of the auxiliary tree β that we have chosen to adjoin. Once we have finished visiting the nodes of β (i.e., we have reached the foot of β) we must return to (the bottom phase of) η. Therefore, it is necessary, while visiting the nodes in β to store the adjunction node η. This can be done by pushing η onto the stack at the point that we move to the root of β. Note that the stack may grow to unbounded length since we might adjoin at a node within β, and so on. When we reach the bottom phase of foot node of β the stack is popped and we find the node at which β was adjoined at the top of the stack.

From the above discussion it is clear that the lig needs just two nonterminals, $\top$ and $\bot$. At each step of a derivation in the lig shared forest grammar the top of the stack will specify the node being currently being visited. Also, if the node η being visited belongs to an auxiliary tree and is on its spine we can expect the symbol below the top of the stack to give us the node where β is adjoined. If η is not on the spine of an auxiliary tree then it is the only symbol on the stack.

We now show how the lig shared forest grammar can be constructed for a given string $w = a_1 \ldots a_n$. Suppose we have a tag

$$G_o = (V_N, V_T, S, I, A)$$

and the dfa

$$M_w = (V_N, Q, q_0, \delta, F)$$

as defined in Section 4. We construct the lig

$$G_w = (V'_N, V_T, V_I, S', P)$$

that generates the intersection of $L(G)$ and $L(M_w)$. P includes the following set of productions for the start symbol S'

$$\{S'[] \rightarrow (\top, q_0, q_f)[\eta] \mid \begin{array}{l} q_f \in F \text{ and} \\ \eta \text{ is root of initial tree} \end{array}$$

In addition, for each elementary node η do the following.

Case 1: When η is a node that is labeled by a terminal a P includes the production

$$(\top, p, q)[\eta] \rightarrow a$$

for each $p, q \in Q$ such that $q \in \delta(p, a)$.

Case 2a: When η_1 and η_2 are the children of a node η such that the left sibling η_1 is on the spine or neither child is on the spine, P includes the production

$$(\bot, p, q)[\circ\circ\, \eta] \rightarrow (\top, p, r)[\circ\circ\, \eta_1]\,(\top, r, q)[\eta_2]$$

for each $p, q, r \in Q$. Note that the stack of adjunction points must be passed to the ancestor of the foot node all the way to the root.

Case 2b: When η_1 and η_2 are the children of a node η such that the right sibling η_2 is on the spine P includes the production

$$(\bot, p, q)[\circ\circ\, \eta] \rightarrow (\top, p, r)[\eta_1]\,(\top, r, q)[\circ\circ\, \eta_2]$$

for each $p, q, r \in Q$.

Case 3: When η is a nonterminal node that does not have an OA constraint P includes the production

$$(\top, p, q)[\circ\circ\, \eta] \rightarrow (\bot, p, q)[\circ\circ\, \eta]$$

for each $p, q \in Q$. This production is used when no adjunction takes place and we move directly between the top and bottom phases of η.

Case 4a: When η is a node where β can be adjoined and η' is the root node of β P includes the production

$$(\top, p, q)[\circ\circ\, \eta] \rightarrow (\top, p, q)[\circ\circ\, \eta\eta']$$

for each $p, q \in Q$. Note that the adjunction node η has been pushed below the new node η' on the stack.

Case 4b: When η is a node where η can be adjoined and η' is the foot node of β P includes the production

$$(\bot, p, q)[\circ\circ\, \eta\eta'] \rightarrow (\bot, p, q)[\circ\circ\, \eta]$$

for each $p, q \in Q$. Note that the stack symbol that appeared below η will be the node at which β was adjoined.

Since the state set of M_w is $\{0, \ldots, n\}$ there are $O(n^2)$ nonterminals in the grammar. Since at most three states are used in the productions, M_w has $O(n^3)$ productions. The time taken to construct this grammar is also $O(n^3)$. As in the cfg shared forest grammar constructed in Section 6 we have assumed that the tag is binary branching for sake of simplifying the presentation. The construction can be adapted to allow for any degree of branching through the use of additional (binary) lig productions. Furthermore, this would not increase the space complexity of the grammar. Finally, note that unlike the cfg shared forest grammar, in the lig shared forest grammar G_w, w is derived in G_o if and only if w is derived in G_w. Of course in both cases $L(G_w) = \{w\} \cap L(G_o)$ and hence the recognition problem can be solved by determining whether the shared forest grammar generates the empty set or not.

8 Removing Useless Symbols

As in the case of the cfg shared forest grammar, to solve the original recognition problem we have to determine if $L(G_w) \neq \phi$. In particular, we have to determine whether $S'[\,]$ derives a terminal string. We solve this question by construcing an nfa, M_{G_w}, from G_w where the states of M_{G_w} correspond to the nonterminal and terminal symbols of G_w. This transforms the question of determining whether a symbol is useful into a reachibility question on the graph of M_{G_w}. In particular, for any string of stack symbols γ, the object $A[\gamma]$ derives a string of terminals if and only if it is possible, in the nfa M_{G_w}, to reach a final state from the state corresponding to A on the input γ. Thus, $w \in L(G_o)$ if and only if $S'[\,] \stackrel{*}{\underset{G_w}{\Longrightarrow}} w$ if and only if in M_{G_w} a final state is reachable from the state corresponding to S' on the empty string.

Given a lig $G_w = (V_N, V_T, V_I, S', P)$ we construct the nfa $M_{G_w} = (Q, \Sigma, \delta, q_0, F)$ as follows. Let the state set of M be the nonterminal and terminal alphabet of G_w: i.e., $Q = V_N \cup V_T$. The initial state of M_{G_w} is the start symbol of G_w, i.e., $q_0 = S'$. The input alphabet of M_{G_w} is the stack alphabet of G_w: i.e., $\Sigma = V_I$. Note that since G_w is the lig shared forest the set V_I is the set of the elementary node addresses of the object tag grammar G_o. The set of final states, F, of M_{G_w} is the set V_T. The transition function δ of M_{G_w} is defined as follows.

Case 1: If P contains the production

$$A[\eta] \rightarrow a$$

then add a to $\delta(A, \eta)$.

Case 2a: If P contains the production

$$A[\circ\circ\, \eta] \rightarrow B[\circ\circ\, \eta_1]C[\eta_2]$$

then if $\delta(C, \eta_2) \cap F \neq \phi$ and $D \in \delta(B, \eta_1)$ add D to $\delta(A, \eta)$.

Case 2b: The case where P contains the production

$$A[\circ\circ\, \eta] \rightarrow C[\eta_2]B[\circ\circ\, \eta_1]$$

is similar to Case 2a.

Case 3: If P contains the production

$$A[\circ\circ\, \eta] \rightarrow B[\circ\circ\, \eta]$$

then if $C \in \delta(B, \eta)$ add $C \in \delta(A, \eta)$.

Case 4a: If P contains the production

$$A[\circ\circ\, \eta] \rightarrow B[\circ\circ\, \eta\eta']$$

then for each C such that $C \in \delta(B, \eta')$ and each D such that $D \in \delta(C, \eta)$ add D to $\delta(A, \eta)$.

Case 4b: If P contains the production

$$A[\circ\circ\, \eta\eta'] \rightarrow B[\eta]$$

then add B to $\delta(A, \eta')$.

Case 5: If P contains the production

$$S'[\,] \rightarrow A[\eta]$$

then if $B \in \delta(A, \eta)$ add B to $\delta(S', \epsilon)$.

Given that $w = a_1 \ldots a_n$ and that the nonterminals (and corresponding states in M_{G_w}) of G_w are of the form $(\top, i, j)$ or $(\bot, i, j)$ where $0 \leq i < j \leq n$, there are $O(n^2)$ nonterminals (states in M_w) in the lig G_w. The size of M_{G_w} is $O(n^4)$ since there are $O(n^2)$ out-transitions from each state.

We can use standard dynamic programming techniques to ensure that each production is considered only once. Given such an algorithm it is easy to check that the construction of M_{G_w} will take $O(n^6)$ time. The worst case corresponds to case 4a which will take $O(n^4)$ for each production. However, there are only $O(n^2)$ such productions (for which case 4a applies). Once the nfa has been constructed the recognition problem (i.e., whether $w \in L(G_o)$) takes $O(n^2)$ time. We have to check if there is an ϵ-transition from the initial state to a final state and hence we will have to consider $O(n^2)$ transitions.

A straightforward algorithm can be used to remove the states for nonterminals that do not appear in any sentential form derived from S'. In other words, only keep states such that for some γ there is a derivation

$$S[\,] \underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_1 A[\gamma] \Upsilon_2$$

for some $\Upsilon_1 \Upsilon_2 \in (V_C(G_w) \cup V_T)^*$.

Note that the states to be removed are not those states that are not reachable from the initial state of M_{G_w}. The set of states reachable from the initial state includes only the set of nonterminals in objects that are the distiguished descendent of the root node in some derivation.

From the construction of M_{G_w} it is that case that for each $A \in V_N$ the set

$$\{\, \gamma \mid a \in \delta(A, \gamma) \text{ for some } a \in F \,\}$$

is equal to the set

$$\left\{ \gamma \;\middle|\; A[\gamma] \underset{G_w}{\overset{*}{\Longrightarrow}} x \text{ for some } x \in V_T^* \right\}$$

Thus, if a final state is accessible from a state A then for some γ (that witnesses the accessibility of a final state from A)

$$A[\gamma] \underset{G_w}{\overset{*}{\Longrightarrow}} x$$

for some $x \in V_T^*$.

Once the construction of M_{G_w} is complete we only retain those productions in G_w that involve nonterminals that remain in the state set of of M_{G_w}. However, unlike the case of the cfg shared forest grammar, the extraction of individual parses for the input w does not simply involve reading off a derivation of G_w. This is due to the fact that although retaining the state A does mean that there is a derivation $S[\,] \underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_1 A[\gamma] \Upsilon_2$ for some γ and $\Upsilon_1 \Upsilon_2$, we can not guarantee that $A[\gamma]$ will derive a string of terminals. The next section describes how to deal with this problem.

9 Recovery of a Parse

Let the lig G_w with useless productions removed be

$$G_w = (V_N, V_T, V_I, S', P)$$

and let the nfa M_{G_w} constructed in Section 8 with unnecessary states removed be

$$M_{G_w} = (V_N \cup V_T, V_I, \delta, S', V_T)$$

Recovering a parse of the string w by the object grammar G_o has now been converted into the problem of extracting one of the derivations of G_w. However, this is not entirely straightforward.

The presence of a state A in $V_N \cup V_T$ indicates that for some γ in V_I^* and Υ_1, Υ_2 in $(V_C(G_w) \cup V_T)^*$ we have

$$S'[\,] \underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_1 A[\gamma] \Upsilon_2$$

However, it is not necessarily the case that $\delta(A, \gamma) \cap V_T \neq \phi$, i.e., it might not be possible to reach a final state of M_{G_w} from A with input γ. All we know is that there is some $\gamma' \in V_I^*$ (that could be distinct from γ) such that $A[\gamma']$ derives a terminal string, i.e., at least one final state is accessible from A on the string γ'.

This means that in recovering a derivation of G_w by considering the top-down application of productions we must be careful about which production we choose at each stage. We cannot assume that any choice of production for an object, $A[\gamma]$ will eventually lead to a complete derivation. Even if the top of the stack γ is compatible with the use of a production, this does not guarantee that $A[\gamma]$ derives a terminal string.

We give an procedure recover that can be used to recover a derivation of G_w by using the nfa M_{G_w}. This procedure guarantees that when we reach a state A by traversing a path γ from the initial state then on the same string γ a final state can be reached from the state A.

If recover$(T_1 \ldots T_n a)$ is invoked the following hold.

- $n \geq 1$
- $a \in V_T$
- $T_i = (A_i, \eta_i)$ where $A_i \in V_N$ and $\eta_i \in V_I$ for each $1 \leq i \leq n$
- recover$(T_1 \ldots T_n a)$ returns a derivation from $A_1[\eta_n \ldots \eta_1]$

- $S'[\,] \underset{G_w}{\overset{*}{\Longrightarrow}} xA_1[\eta_n \ldots \eta_1]y$ for some $x, y \in V_T^*$
- $$\begin{aligned} A_1[\eta_n \ldots \eta_1] &\underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_{1,l} A_2[\eta_n \ldots \eta_2]\Upsilon_{1,r} \\ &\vdots \\ &\underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_{n-1,l} A_n[\eta_n]\Upsilon_{n-1,r} \\ &\underset{G_w}{\overset{*}{\Longrightarrow}} \Upsilon_{n,l} a \Upsilon_{n,r} \end{aligned}$$
- $\delta(A_i, \eta_n \ldots \eta_i) = a$ for each $1 \leq i \leq n$,

To recover a parse we call $\mathsf{recover}(((\top, 1, n), \eta)a)$ where $a \in V_T$ such that $\delta((\top, 1, n), \eta) = a$ and $\eta \in V_I$ is the root of some initial tree. The definition of recover is as follows.

Procedure $\mathsf{recover}((A_1, \eta_1)T_2 \ldots T_n a)$

Case 1: If $n = 1$ and

$$p = A_1[\eta_1] \rightarrow a \in P$$

then output p. Note there must be such a production

Case 2a: If there is some production

$$p = A_1[\circ\circ\, \eta_1] \rightarrow B[\circ\circ\, l']\, C[l''] \in P$$

such that $\delta(C, l'') = b$ for some $b \in V_T$, and either $n > 1$ and $A_2 \in \delta(B, l')$ (where $T_2 = (A_2, \eta_2)$) or $n = 1$ and $a \in \delta(B, l')$ then output

$$p \cdot \mathsf{recover}((B, l')T_2 \ldots T_n a) \cdot \mathsf{recover}((C, l'')b)$$

Case 2b: If there is some production

$$p = A_1[\circ\circ\, \eta_1] \rightarrow C[l'']\, B[\circ\circ\, l'] \in P$$

such that $\delta(C, l'') = b$ for some $b \in V_T$ and either $n > 1$ and $A_2 \in \delta(B, l')$ (where $T_2 = (A_2, \eta_2)$) or $n = 1$ and $a \in \delta(B, l')$ then output

$$p \cdot \mathsf{recover}((B, l')T_2 \ldots T_n a) \cdot \mathsf{recover}((C, l'')b)$$

Case 3: If there is some production

$$p = A_1[\circ\circ\, \eta_1] \rightarrow B[\circ\circ\, l'] \in P$$

such that either $n > 1$ and $A_2 \in \delta(B, l')$ (where $T_2 = (A_2, \eta_2)$) or $n = 1$ and $a \in \delta(B, l')$ then output

$$p \cdot \mathsf{recover}((B, l')T_2 \ldots T_n a)$$

Case 4a: If there is some production

$$p = A_1[\circ\circ\, \eta_1] \rightarrow B[\circ\circ\, \eta_2 l'] \in P$$

such that $C \in \delta(B, l')$ for some $C \in V_N$ and $A_2 \in \delta(C, \eta_1)$ and either $n > 1$ and $T_2 = (A_2, \eta_2)$ or $n = 1$ and $a \in \delta(C, \eta_1)$ then output

$$p \cdot \mathsf{recover}((B, l')(C, \eta_1)T_2 \ldots T_n a)$$

Case 4b: If there is a production

$$p = A_1[\circ\circ\, \eta_2\eta_1] \rightarrow A_2[\circ\circ\, \eta_2] \in P$$

such that $n > 1$ and $T_2 = (A_2, \eta_2)$ then output

$$p \cdot \mathsf{recover}(T_2 \ldots T_n)$$

Given the form of the nonterminals and productions of G_w we can see that the complexity of extracting a parse as above is dominated by the complexity by Case 4a which takes $O(n^4)$ time. If in G_o every elementary tree has at least one terminal symbol in its frontier (as in a lexicalized tag) then to derive a string of length n there can be at most n adjunctions. In that case, when we wish to recover a parse the derivation height (which gives recursion depth of the the invocation of the above procedure) is $O(n)$ and hence recovery of a parse will take $O(n^5)$ time.

10 Conclusions

We have shown that there are two distinct ways of representing the parses of a tag using lig and cfg.

- The cfg representation captures the fact that the choice of which trees to adjoin at each step of a derivation is context-free. In this approach the number of nonterminals is $O(n^4)$, the number of productions is $O(n^6)$ and, hence, the recognition problem can be resolved in $O(n^6)$ time with $O(n^4)$ space. Note that now the problem of whether the input string can be derived in the tag grammar is equivalent to deciding whether the shared forest cfg obtained generates the empty language or not. Each derivation of the shared forest cfg represents a parse of the given input string by the tag.
- In the scheme that uses lig the number of nonterminals is $O(n^2)$ and the number of productions is $O(n^3)$. While the space complexity of the shared forest is more compact in the case of lig, recovering a parse is less straightforward. In order to facilitate recovery of a parse as well as to solve the recognition problem (i.e., determine if the language generated by the shared forest grammar is nonempty) we use an augmented data structure (the nfa, M_{G_w}). With this structure the recognition problem can again be resolved in $O(n^6)$ with $O(n^4)$ space and the extraction of a parse has $O(n^5)$ time complexity.

The work described here is intended to provide a general framework that can be used to study and compare existing tag parsing algorithms (for example [Vijay-Shanker and Joshi, 1985; Vijay-Shanker and Weir, in pressb; Schabes and Joshi, 1988]). If we factor out the particular dynamic programming algorithm used to determine the sequence in which these rules are considered then the productions of our cfg and lig shared forest grammars encapsulate the steps of all of these algorithms. In particular, the algorithm presented in [Vijay-Shanker and Joshi, 1985] can be seen to corresponds to the approach involving the use of cfg to encode derivations, whereas, the algorithm of [Vijay-Shanker and Weir, in pressb]

uses lig in this role. Although the space complexity of the cited parsing algorithms is $O(n^4)$, the data structures used by them do not explicitly give the shared forest representation provided by our shared forest grammars. The data structures would have to be extended to record how each entry in the table gets added. With this kind of additional information the space requirements of these algorithms would become $O(n^6)$.

It is perhaps not surprising that the lig shared forest and cfg shared forest described here turn out to be closely related. In the nfa M_{G_w} (after useless symbols have been removed) we have $(B, p, q) \in \delta((A, i, j), \eta)$ if and only if in the cfg shared forest (A, η, i, j, p, q) is not a useless symbol. In addition, there is a close correspondence between productions in the two shared forest grammars. This shows that the two schemes result in essentially the same algorithms that store essentially the same information in the tables that they build.

We end by noting that Lang [1992] also considers tag parsing with shared forest grammars, however, he uses the tag formalism itself to encode the shared forest. This does not utilize the distinction between derivation and derived trees in a tag. The algorithms presented here specialize the derivation tree grammar to get shared forest whereas Lang [1992] specializes object grammar itself. As a result, in order to get $O(n^6)$ time complexity Lang must assume the object grammar tree in a very restricted normal form.

References

[Aho, 1968] A. V. Aho. Indexed grammars — An extension to context free grammars. *J. ACM*, 15:647–671, 1968.

[Billot and Lang, 1989] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In 27^{th} *meeting Assoc. Comput. Ling.*, 1989.

[Gazdar, 1988] G. Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*. D. Reidel, Dordrecht, Holland, 1988.

[Joshi *et al.*, 1975] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1), 1975.

[Lang, 1992] B. Lang. Recognition can be harder than parsing. Presented at the Second TAG Workshop, 1992.

[Schabes and Joshi, 1988] Y. Schabes and A. K. Joshi. An Earley-type parsing algorithm for tree adjoining grammars. In 26^{th} *meeting Assoc. Comput. Ling.*, 1988.

[Vijay-Shanker and Joshi, 1985] K. Vijay-Shanker and A. K. Joshi. Some computational properties of tree adjoining grammars. In 23^{rd} *meeting Assoc. Comput. Ling.*, pages 82–93, 1985.

[Vijay-Shanker and Weir, in pressa] K. Vijay-Shanker and D. J. Weir. The equivalence of four extensions of context-free grammars. *Math. Syst. Theory*, in press.

[Vijay-Shanker and Weir, in pressb] K. Vijay-Shanker and D. J. Weir. Parsing constrained grammar formalisms. *Comput. Ling.*, in press.

[Vijay-Shanker, 1987] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 1987.