Tree-Adjoining Grammars

Aravind K. Joshi[1] and Yves Schabes[2]

[1] Department of Computer and Information Science, and
The Institute for Research in Cognitive Science,
University of Pennsylvania, Philadelphia, PA 19104, USA
email: joshi@linc.cis.upenn.edu
[2] Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, USA
email: schabes@merl.com

1. Introduction

In this paper, we will describe a tree generating system called tree-adjoining grammar (TAG) and state some of the recent results about TAGs. The work on TAGs is motivated by linguistic considerations. However, a number of formal results have been established for TAGs, which we believe, would be of interest to researchers in formal languages and automata, including those interested in tree grammars and tree automata.

After giving a short introduction to TAG, we briefly state these results concerning both the properties of the string sets and tree sets (Section 2.). We will also describe the notion of lexicalization of grammars (Section 3.) and investigate the relationship of lexicalization to context-free grammars (CFGs) and TAGs and then summarize the issues on lexicalization (Sections 4., 5. and 6.). We then describe an automaton model that exactly corresponds to TAGs (Section 7.). As we have said earlier TAGs were motivated by some important linguistic considerations. The formal aspects of these considerations are mathematically important also. Hence we have presented a brief discussion of these issues together with some simple examples (Section 8.). We also present in Section 9. some variants of TAGs that are currently under investigation. We then present a bottom up predictive parser for TAGs, which is both theoretically and practically important (Section 10.) and then offer some concluding remarks (Section 11.).

The motivations for the study of tree-adjoining grammars (TAG) are of linguistic and formal nature. The elementary objects manipulated by a TAG are trees, i.e., structured objects and not strings. Using structured objects as the elementary objects of a formalism, it is possible to construct formalisms whose properties relate directly to the strong generative capacity (structural description) which is more relevant to linguistic descriptions than the weak generative capacity (set of strings).

TAG is a tree-generating system rather than a string generating system. The set of trees derived in a TAG constitute the object language. Hence, in order to describe the derivation of a tree in the object language, it is necessary to talk about derivation 'trees' for the object language trees. These derivation trees are important both syntactically and semantically. It has also turned out that some other formalisms which are weakly equivalent to TAGs are similar to each other in terms of the properties of the derivation 'trees' of these formalisms [Weir1988, Joshi et al.1991].

Another important linguistic motivation for TAGs is that TAGs allow factoring recursion from the statement of linguistic constraints (dependencies), thus making these constraints strictly local, and thereby simplifying linguistic description [Kroch and Joshi1985].

Lexicalization of grammar formalism is also one of the key motivations, both linguistic and formal. Most current linguistic theories give lexical accounts of several phenomena that used to be considered purely syntactic. The information put in the lexicon is thereby increased in both amount and complexity[1].

On the formal side, lexicalization allows us to associate each elementary structure in a grammar with a lexical item (terminal symbol in the context of formal grammars). The well-known Greibach Normal Form (CNF) for CFG is a kind of lexicalization, however it is a *weak* lexicalization in a certain sense as it does not preserve structures of the original grammar. Our tree based approach to lexicalization allows us to achieve lexicalization while preserving structures, which is linguistically very significant.

2. Tree-Adjoining Grammars

TAGs were introduced by Joshi, Levy and Takahashi (1975) and Joshi (1985). For more details on the original definition of TAGs, we refer the reader to [Joshi1987, Kroch and Joshi1985]. It is known that tree-adjoining languages (TALs) generate some strictly context-sensitive languages and fall in the class of the so-called 'mildly context-sensitive languages' [Joshi et al.1991]. TALs properly contain context-free languages and are properly contained by indexed languages.

Although the original definition of TAGs did not include substitution as a combining operation, it can be easily shown that the addition of substitution does not affect the formal properties of TAGs.

We first give an overview of TAG and then we study the lexicalization process.

Definition 2.1 (tree-adjoining grammar).
A tree-adjoining grammar (TAG) consists of a quintuple (Σ, NT, I, A, S), where

[1] Some of the linguistic formalisms illustrating the increased use of lexical information are, lexical rules in LFG [Kaplan and Bresnan1983], GPSG [Gazdar et al.1985], HPSG [Pollard and Sag1987], Combinatory Categorial Grammars [Steedman1987], Karttunen's version of Categorial Grammar [Karttunen1986], some versions of GB theory [Chomsky1981], and Lexicon-Grammars [Gross1984].

(i) Σ is a finite set of terminal symbols;
(ii) NT is a finite set of non-terminal symbols[2]: $\Sigma \cap NT = \emptyset$;
(iii) S is a distinguished non-terminal symbol: $S \in NT$;
(iv) I is a finite set of finite trees, called *initial trees*, characterized as follows (see tree on the left in Fig. 2.1):

- interior nodes are labeled by non-terminal symbols;
- the nodes on the frontier of initial trees are labeled by terminals or non-terminals; non-terminal symbols on the frontier of the trees in I are marked for substitution; by convention, we annotate nodes to be substituted with a down arrow ($\downarrow$);

(v) A is a finite set of finite trees, called *auxiliary trees*, characterized as follows (see tree on the right in Fig. 2.1):

- interior nodes are labeled by non-terminal symbols;
- the nodes on the frontier of auxiliary trees are labeled by terminal symbols or non-terminal symbols. Non-terminal symbol on the frontier of the trees in A are marked for substitution except for one node, called the *foot node*; by convention, we annotate the foot node with an asterisk ($*$); the label of the foot node must be identical to the label of the root node.

In *lexicalized TAG*, at least one terminal symbol (the anchor) must appear at the frontier of all initial or auxiliary trees.

The trees in $I \cup A$ are called *elementary trees*. We call an elementary tree an X-type elementary tree if its root is labeled by the non-terminal X.

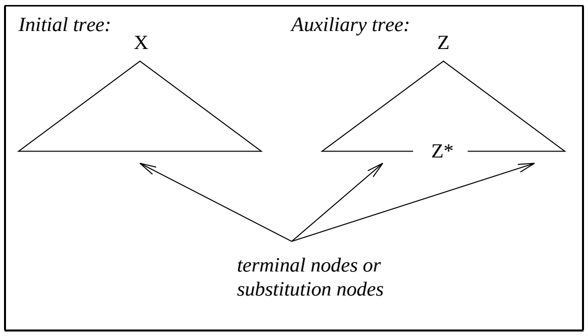


Fig. 2.1. Schematic initial and auxiliary trees.

A tree built by composition of two other trees is called a *derived tree*.

We now define the two composition operations that TAG uses: *adjoining* and *substitution*.

[2] We use lower-case letters for terminal symbols and upper-case letters for non-terminal symbols.

Adjoining builds a new tree from an auxiliary tree β and a tree α (α is any tree, initial, auxiliary or derived). Let α be a tree containing a non-substitution node n labeled by X and let β be an auxiliary tree whose root node is also labeled by X. The resulting tree, γ, obtained by adjoining β to α at node n (see top two illustrations in Fig. 2.2) is built as follows:

- the sub-tree of α dominated by n, call it t, is excised, leaving a copy of n behind.
- the auxiliary tree β is attached at the copy of n and its root node is identified with the copy of n.
- the sub-tree t is attached to the foot node of β and the root node of t (i.e. n) is identified with the foot node of β.

The top two illustrations in Fig. 2.2 illustrate how adjoining works. The auxiliary tree β_1 is adjoined on the VP node in the tree α_2. α_1 is the resulting tree.

Substitution takes only place on non-terminal nodes of the frontier of a tree (see bottom two illustrations in Fig. 2.2). An example of substitution is given in the fourth illustration (from the top) in Fig. 2.2. By convention, the nodes on which substitution is allowed are marked by a down arrow ($\downarrow$). When substitution occurs on a node n, the node is replaced by the tree to be substituted. When a node is marked for substitution, only trees derived from initial trees can be substituted for it.

By definition, any adjunction on a node marked for substitution is disallowed. For example, no adjunction can be performed on any NP node in the tree α_2. Of course, adjunction is possible on the root node of the tree substituted for the substitution node.

2.1 Adjoining Constraints

In the system that we have described so far, an auxiliary tree β can be adjoined on a node n if the label of n is identical to the label of the root node of the auxiliary tree β and if n is labeled by a non-terminal symbol not annotated for substitution. It is convenient for linguistic description to have more precision for specifying which auxiliary trees can be adjoined at a given node. This is exactly what is achieved by *constraints on adjunction* [Joshi1987]. In a TAG $G = (\Sigma, NT, I, A, S)$, one can, for each node of an elementary tree (on which adjoining is allowed), specify one of the following three constraints on adjunction:

- *Selective Adjunction* ($SA(T)$, for short): only members of a set $T \subseteq A$ of auxiliary trees can be adjoined on the given node. The adjunction of an auxiliary is not mandatory on the given node.
- *Null Adjunction* (NA for short): it disallows any adjunction on the given node.[3]

[3] Null adjunction constraint corresponds to a selective adjunction constraint for which the set of auxiliary trees T is empty: $NA = SA(\emptyset)$

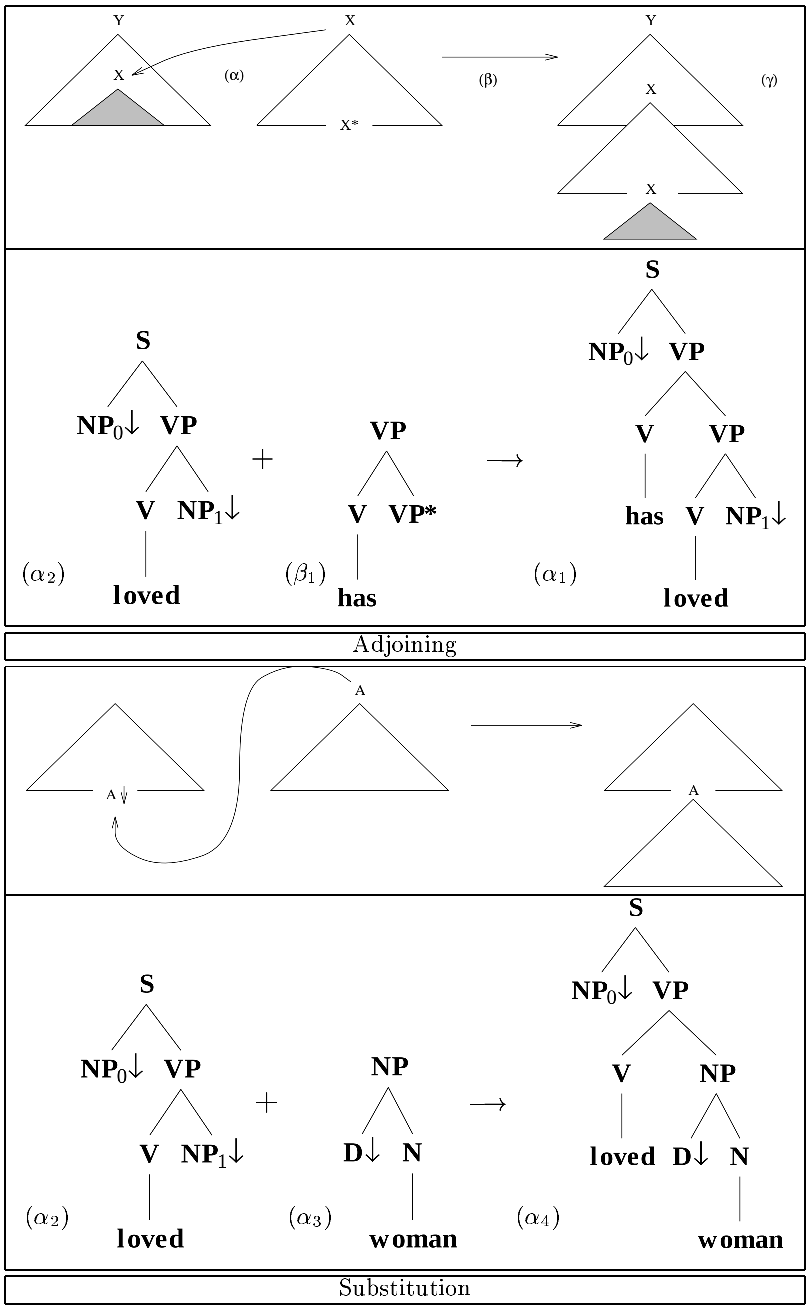


Fig. 2.2. Combining operations: adjoining and substitution

- *Obligatory Adjunction* ($OA(T)$, for short): an auxiliary tree member of the set $T \subseteq A$ must be adjoined on the given node. In this case, the adjunction of an auxiliary tree is mandatory. OA is used as a notational shorthand for $OA(A)$.

If there are no substitution nodes in the elementary trees and if there are no constraints on adjoining, then we have the 'pure' (old) Tree Adjoining Grammar (TAG) as described in [Joshi et al.1975].

The operation of substitution and the constraints on adjoining are both needed for linguistic reasons. Constraints on adjoining are also needed for formal reasons in order to obtain some closure properties.

2.2 Derivation in TAG

We now define by an example the notion of derivation in a TAG. Unlike CFGs, the tree obtained by derivation (the *derived tree*) does not give enough information to determine how it was constructed. The *derivation tree* is an object that specifies uniquely how a derived tree was constructed. Both operations, adjunction and substitution, are considered in a TAG derivation. Take for example the derived tree α_5 in Fig. 2.3; α_5 yields the sentence *yesterday a man saw Mary.* It has been built with the elementary trees shown in Fig. 2.4.

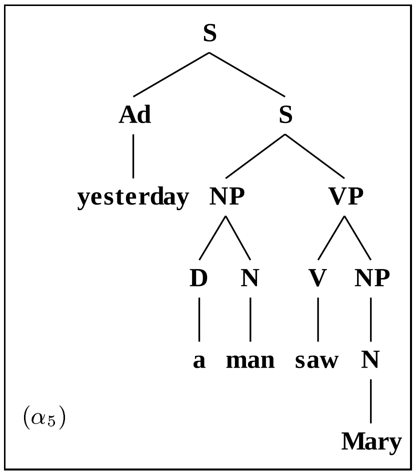


Fig. 2.3. Derived tree for: *yesterday a man saw Mary.*

The root of a derivation tree for TAGs is labeled by an S-type initial tree. All other nodes in the derivation tree are labeled by auxiliary trees in the case of adjunction or initial trees in the case of substitution. A tree address is associated with each node (except the root node) in the derivation tree. This tree address is the address of the node in the parent tree to which the adjunction or substitution has been performed. We use the following

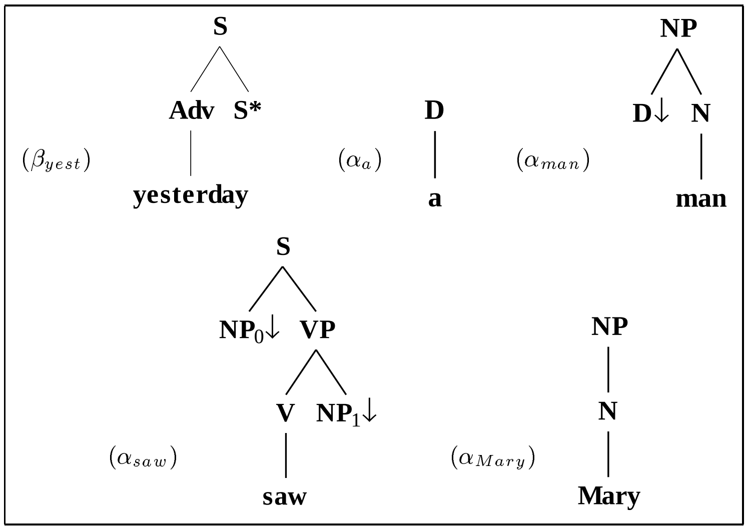


Fig. 2.4. Some elementary trees.

convention: trees that are adjoined to their parent tree are linked by an unbroken line to their parent, and trees that are substituted are linked by a dashed line.[4] Since by definition, adjunction can only occur at a particular node one time, all the children of a node in the derivation tree will have distinct addresses associated with them.

The derivation tree in Fig. 2.5 specifies how the derived tree α_5 pictured in Fig. 2.3 was obtained.

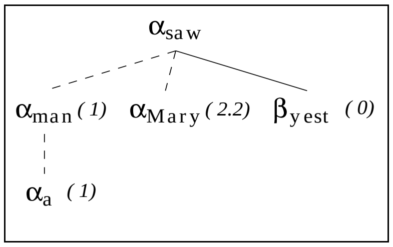


Fig. 2.5. Derivation tree for *Yesterday a man saw Mary.*

This derivation tree (Fig. 2.5) should be interpreted as follows: α_a is substituted in the tree α_{man} at the node of address 1 (D), α_{man} is substituted in the tree α_{saw} at address 1 (NP_0) in α_{man}, α_{Mary} is substituted in the tree

[4] We will use Gorn addresses as tree addresses: 0 is the address of the root node, k is the address of the k^{th} child of the root node, and $p \cdot q$ is the address of the q^{th} child of the node at address p.

α_{saw} at node $2 \cdot 2$ (NP_1) and the tree β_{yest} is adjoined in the tree α_{saw} at node 0 (S).

The order in which the derivation tree is interpreted has no impact on the resulting derived tree.

2.3 Some properties of the string languages and tree sets

We summarize some of the well known properties of tree-adjoining grammar's string languages and of the tree sets.

The *tree set* of a TAG is defined as the set of completed[5] initial trees derived from some S-rooted initial tree:

$$T_G = \{t \mid t \textit{ is `derived' from some S-rooted initial tree}\}$$

The string language of a TAG, $L(G)$, is then defined as the set of yields of all the trees in the tree set:

$$L_G = \{w \mid w \textit{ is the yield of some } t \textit{ in } T_G\}$$

Adjunction is more powerful than substitution and it generates some context-sensitive languages (see Joshi [1985] for more details). [6]

Some well known properties of the string languages follow:

- context-free languages are strictly included in tree-adjoining languages, which themselves are strictly included in indexed languages;
  $$CFL \subset TAL \subset \textit{Indexed Languages} \subset CSL$$
- TALs are semilinear;
- All closure properties of context-free languages also hold for tree-adjoining languages. In fact, TALs are a full abstract family of languages (full AFLs).
- a variant of the push-down automaton called embedded push-down automaton (EPDA) [Vijay-Shanker1987] characterizes exactly the set of tree-adjoining languages, just as push-down automaton characterizes CFLs.
- there is a pumping lemma for tree-adjoining languages.
- tree-adjoining languages can be parsed in polynomial time, in the worst case in $O(n^6)$ time.

Some well know properties of the tree sets of tree-adjoining grammars follow:

- the tree sets of recognizable sets (regular tree sets)[Thatcher1971] are strictly included in the tree sets of tree-adjoining grammars, $\mathcal{T}(TAG)$;
  $$\textit{recognizable sets} \subset \mathcal{T}(TAG)$$

[5] We say that an initial tree is *completed* if there is no substitution nodes on the frontier of it.

[6] Adjunction can simulate substitution with respect to the weak generative capacity. It is also possible to encode a context-free grammar with auxiliary trees using adjunction only. However, although the languages correspond, the possible encoding does not directly reflect the tree set of original context-free grammar since this encoding uses adjunction.

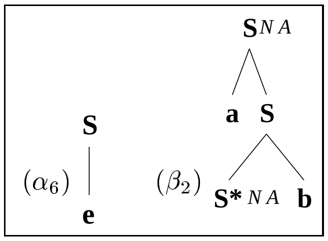
- the set of paths of all the trees in the tree set of a given TAG, $\mathcal{P}(T(G))$, is a context-free language;

$$\mathcal{P}(T(G)) \textit{ is a CFL}$$

- the tree sets of TAG are equivalent to the tree sets of linear indexed languages. Hence, linear versions of Schimpf-Gallier tree automaton [Schimpf and Gallier1985] are equivalent to $\mathcal{T}(TAG)$;
- for every TAG, G, the tree set of G, $T(G)$, is recognizable in polynomial time, in the worst case in $O(n^3)$-time, where n is the number of nodes in a tree $t \in T(G)$.

We now give two examples to illustrate some properties of tree-adjoining grammars.

Example 2.1. Consider the following TAG $G_1 = (\{a, e, b\}, \{S\}, \{\alpha_6\}, \{\beta_2\}, S)$



G_1 generates the language $L_1 = \{a^n e b^n | n \geq 1\}$. For example, in Fig. 2.6, α_7 has been obtained by adjoining β_2 on the root node of α_6 and α_8 has been obtained by adjoining β_2 on the node at address 2 in in the tree α_7.

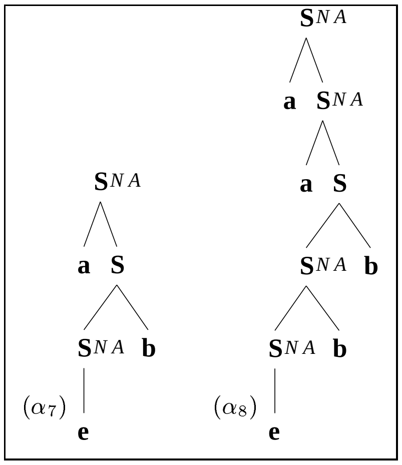
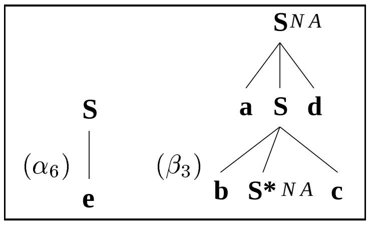


Fig. 2.6. Some derived trees of G_1

Although L_1 is a context-free language, G_1 generates cross serial dependencies. For example, α_7 generates, a_1eb_1, and α_8 generates, $a_1a_2eb_2b_1$. It can be shown that $T(G_1)$ is not recognizable.

Example 2.2. Consider the TAG $G_2 = (\{a,b,c,d,e\}, \{S\}, \{\alpha_6\}, \{\beta_3\}, S)$ below



G_2 generates the context-sensitive language $L_2 = \{a^nb^nec^nd^n | n \geq 1\}$. For example, in Fig. 2.7, α_9 has been obtained by adjoining β_3 on the root node of α_6 and α_{10} has been obtained by adjoining β_3 on the node at address 2 in in the tree α_9.

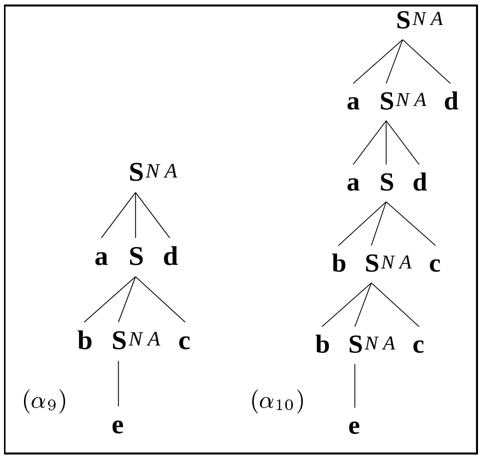


Fig. 2.7. Some derived trees of G_2

One can show that $\{a^nb^nc^nd^ne^n | n \geq 1\}$ is not a tree-adjoining language.

We have seen in Section 2.2 that the derivation in TAG is also a tree. For a given TAG, G, it can be shown that the set of derivations trees of G, $D(G)$, is recognizable (in fact a local set). Many different grammar formalisms have been shown to be equivalent to TAG, their derivation trees are also local sets.

3. Lexicalized Grammars

We define a notion of "lexicalized grammars" that is of both linguistic and formal significance. We then show how TAG arises in the processes of lexicalizing context-free grammars.

In this "lexicalized grammar" approach [Schabes et al.1988, Schabes1990], each elementary structure is systematically associated with a lexical item called the *anchor*. By 'lexicalized' we mean that in each structure there is a lexical item that is realized. The 'grammar' consists of a lexicon where each lexical item is associated with a finite number of structures for which that item is the anchor. There are operations which tell us how these structures are composed. A grammar of this form will be said to be 'lexicalized'.

Definition 3.1 (Lexicalized Grammar). A grammar is 'lexicalized' if it consists of:

- a finite set of structures each associated with a lexical item; each lexical item will be called the *anchor* of the corresponding structure;
- an operation or operations for composing the structures.

We require that the anchor must be an overt (i.e. not the empty string) lexical item.

The *lexicon* consists of a finite set of structures each associated with an anchor. The structures defined by the lexicon are called *elementary structures*. Structures built by combination of others are called *derived structures*.

As part of our definition of lexicalized grammars, we require that the structures be of finite size. We also require that the combining operations combine a finite set of structures into a finite number of structures. We will consider operations that combine two structures to form one derived structure.

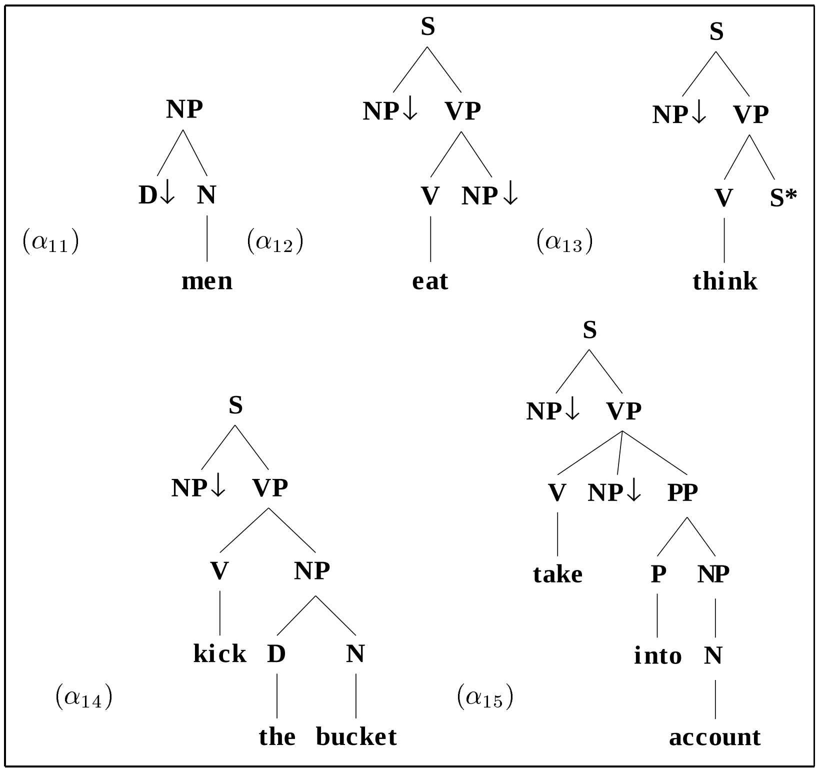
Other constraints can be put on the operations. For examples, the operations could be restricted not to copy, erase or restructure unbounded components of their arguments. We could also impose that the operations yield languages of constant growth (Joshi [1985]). The operations that we will use have these properties.

Categorial Grammars [Lambek1958, Steedman1987] are lexicalized according to our definition since each basic category has a lexical item associated with it.

As in Categorial Grammars, we say that the *category* of a word is the entire structure it selects. If a structure is associated with an anchor, we say that the entire structure is the *category* structure of the anchor.

We also use the term 'lexicalized' when speaking about structures. We say that a structure is *lexicalized* if there is at least one overt lexical item that appears in it. If more than one lexical item appears, either one lexical item is designated as the anchor or a subset of the lexical items local to the structure are designated as *multi-component anchor*. A grammar consisting of only lexicalized structures is of course lexicalized.

For example, the following structures are lexicalized according to our definition:[7]



Some simple properties follow immediately from the definition of lexicalized grammars.

Proposition 3.1. *Lexicalized grammars are finitely ambiguous.*

A grammar is said to be finitely ambiguous if there is no sentence of finite length that can be analyzed in an infinite number of ways.

The fact that lexicalized grammars are finitely ambiguous can be seen by considering an arbitrary sentence of finite length. The set of structures anchored by the words in the input sentence consists of a set of structures necessary to analyze the sentence; while any other structure introduces lexical items not present in the input string. Since the set of selected structures is finite, these structures can be combined in finitely many ways (since each tree is associated with at least one lexical item and since structures can combine to produce finitely many structures). Therefore lexicalized grammars are finitely ambiguous.

[7] The interpretation of the annotations on these structures is not relevant now and it will be given later.

Since a sentence of finite length can only be finitely ambiguous,the search space used for analysis is finite. Therefore, the recognition problem for lexicalized grammars is decidable

Proposition 3.2. *It is decidable whether or not a string is accepted by a lexicalized grammar.*[8]

Having stated the basic definition of lexicalized grammars and also some simple properties, we now turn our attention to one of the major issues: can context-free grammars be lexicalized?

Not every grammar is in a lexicalized form. Given a grammar G stated in a formalism, we will try find another grammar G_{lex} (not necessarily stated in the same formalism) that generates the same language and also the same tree set as G and for which the lexicalized property holds. We refer to this process as lexicalization of a grammar.

Definition 3.2 (Lexicalization). We say that a formalism F can be lexicalized by another formalism F', if for any finitely ambiguous grammar G in F there is a grammar G' in F' such that G' is a lexicalized grammar and such that G and G' generate the same tree set (and a fortiori the same language).

The next section discusses what it means to lexicalize a grammar. We will investigate the conditions under which such a 'lexicalization' is possible for CFGs and tree-adjoining grammars (TAGs). We present a method to lexicalize grammars such as CFGs, while keeping the rules in their full generality. We then show how a lexicalized grammar naturally follows from the extended domain of locality of TAGs.

4. 'Lexicalization' of CFGs

Our definition of lexicalized grammars implies their being finitely ambiguous. Therefore a necessary condition of lexicalization of a CFG is that it is finitely ambiguous. As a consequence, recursive chain rules obtained by derivation (such as $X \overset{*}{\Rightarrow} X$) or elementary (such as $X \rightarrow X$) are disallowed since they generate infinitely ambiguous branches without introducing lexical items.

In general, a CFG will not be in lexicalized form. For example a rule of the form, $S \rightarrow NP\ VP$ or $S \rightarrow S\ S$, is not lexicalized since no lexical item appears on the right hand side of the rule.

A lexicalized CFG would be one for which each production rule has a terminal symbol on its right hand side. These constitute the structures associated with the lexical anchors. The combining operation is the standard substitution operation.[9]

[8] Assuming that one can compute the result of the combination operations.

[9] Variables are independently substituted by substitution. This standard (or first order) substitution contrasts with more powerful versions of substitution which allow to substitute multiple occurrences of the same variable by the same term.

Lexicalization of CFG that is achieved by transforming it into an equivalent Greibach Normal Form CFG, can be regarded as a *weak* lexicalization, because it does not give us the same set of trees as the original CFG. Our notion of lexicalization can be regarded as *strong* lexicalization.[10]

In the next sections, we propose to extend the domain of locality of context-free grammar in order to make lexical item appear local to the production rules. The domain of locality of a CFG is extended by using a tree rewriting system that uses only substitution. We will see that in general, CFGs cannot be lexicalized using substitution alone, even if the domain of locality is extended to trees. Furthermore, in the cases where a CFG could be lexicalized by extending the domain of locality and using substitution alone, we will show that, in general, there is not enough freedom to choose the anchor of each structure. This is important because we want the choice of the anchor for a given structure to be determined on purely linguistic grounds. We will then show how the operation of adjunction enables us to freely 'lexicalize' CFGs.

4.1 Substitution and Lexicalization of CFGs

We already know that we need to assume that the given CFG is finitely ambiguous in order to be able to lexicalized it. We propose to extend the domain of locality of CFGs to make lexical items appear as part of the elementary structures by using a grammar on trees that uses substitution as combining operation. This tree-substitution grammar consists of a set of trees that are not restricted to be of depth one (rules of context-free grammars can be thought as trees of depth one) combined with substitution.[11]

A finite set of elementary trees that can be combined with substitution define a tree-based system that we will call a *tree substitution grammar*.

Definition 4.1 (Tree-Substitution Grammar).
A Tree-Substitution Grammar (TSG) consists of a quadruple (Σ, NT, I, S), where

(i) Σ is a finite set of terminal symbols;
(ii) NT is a finite set of non-terminal symbols[12]: $\Sigma \cap NT = \emptyset$;
(iii) S is a distinguished non-terminal symbol: $S \in NT$;
(iv) I is a finite set of finite trees whose interior nodes are labeled by non-terminal symbols and whose frontier nodes are labeled by terminal or non- terminal symbols. All non-terminal symbols on

[10] For some recent results in strong lexicalization and CFGs, see [Schabes and Waters1995] and the discussion in Section 10.
[11] We assume here first order substitution meaning that all substitutions are independent.
[12] We use lower-case letters for terminal symbols and upper-case letters for non-terminal symbols.

the frontier of the trees in I are marked for substitution. The trees in I are called *initial* trees.

We say that a tree is *derived* if it has been built from some initial tree in which initial or derived trees were substituted. A tree will be said of *type* X if its root is labeled by X. A tree is considered *completed* if its frontier is to be made only of nodes labeled by terminal symbols.

Whenever the string of labels of the nodes on the frontier of an X-type initial tree t_X is $\alpha \in (\Sigma \cup NT)^\star$, we will write: $Fr(t_X) = \alpha$.

As for TAG, the derivation in TSG is stated in the form of a tree called the *derivation tree* (see Section 2.2).

It is easy to see that the set of languages generated by this tree rewriting system is exactly the same set as context-free languages.

We now come back to the problem of lexicalizing context-free grammars. One can try to lexicalize finitely ambiguous CFGs by using tree-substitution grammars. However we will exhibit a counter example that shows that, in the general case, finitely ambiguous CFGs cannot be lexicalized with a tree system that uses substitution as the only combining operation.

Proposition 4.1. *Finitely ambiguous context-free grammars cannot be lexicalized with a tree-substitution grammar.*

Proof[13] of Proposition 4.1
We show this proposition by a contradiction. Suppose that finitely ambiguous CFGs can be lexicalized with TSG. Then the following CFG can be lexicalized:[14]

Example 4.1 (counter example).

$$\begin{array}{l} S \rightarrow S\ S \\ S \rightarrow a \end{array}$$

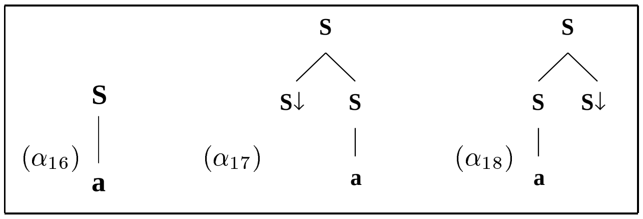
Suppose there were a lexicalized TSG G generating the same tree set as the one generated by the above grammar. Any derivation in G must start from some initial tree. Take an arbitrary initial tree t in G. Since G is a lexicalized version of the above context-free grammar, there is a node n on the frontier of t labeled by a. Since substitution can only take place on the frontier of a tree, the distance between n and the root node of t is constant in any derived tree from t. And this is the case for any initial tree t (of which there are only finitely many). This implies that in any derived tree from G there is at least one branch of bounded length from the root node to a node labeled by a (that branch cannot further expand). However in the derivation trees defined by the context-free grammar given above, a can occur arbitrarily

[13] The underlying idea behind this proof was suggested to us by Stuart Shieber.
[14] This example was pointed out to us by Fernando Pereira.

far away from the root node of the derivation. Contradiction.
□

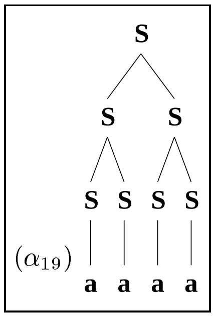
The CFG given in Example 4.1 cannot be lexicalized with a TSG. The difficulty is due to the fact that TSGs do not permit the distance between two nodes in the same initial tree to increase.

For example, one might think that the following TSG is a lexicalized version of the above grammar:



(α_{16}) (α_{17}) (α_{18})

However, this lexicalized TSG does not generate all the trees generated by the context-free grammar; for example the following tree (α_{19}) cannot be generated by the above TSG:



(α_{19})

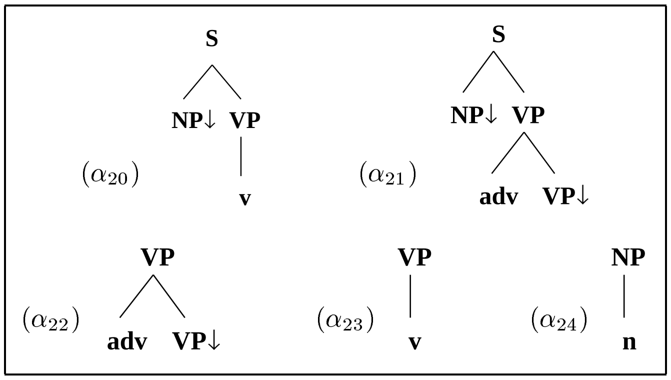
We now turn to a less formal observation. Even if some CFGs can be lexicalized by using TSG, the choice of the lexical items that emerge as the anchor may be too restrictive, for example, the choice may not be linguistically motivated.

Consider the following example:

Example 4.2.

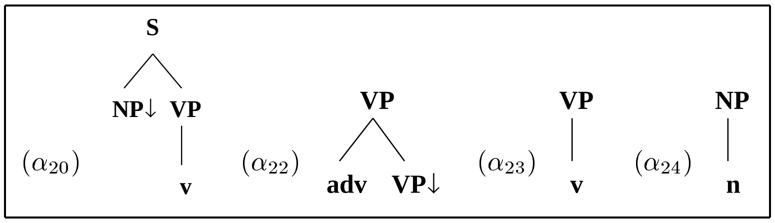
$$\begin{aligned} S &\rightarrow NP\ VP \\ VP &\rightarrow adv\ VP \\ VP &\rightarrow v \\ NP &\rightarrow n \end{aligned}$$

The grammar can be lexicalized as follows:

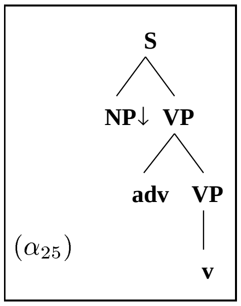


This tree-substitution grammar generates exactly the same set of trees as in Example 4.2, however, in this lexicalization one is forced to choose *adv* (or *n*) as the anchor of a structure rooted by S (α_{21}), and it cannot be avoided. This choice is not linguistically motivated. If one tried not to have an S-type initial tree anchored by n or by *adv*, recursion on the VP node would be inhibited.

For example, the grammar written below:



does not generate the tree α_{25}:



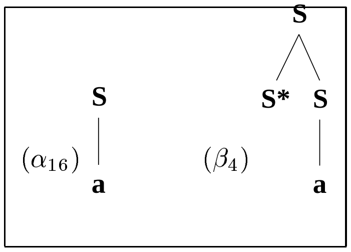
This example shows that even when it is possible to lexicalize a CFG, substitution (TSG) alone does not allow us to freely choose the lexical anchors. Substitution alone forces us to make choices of anchors that might not be linguistically (syntactically or semantically) justified. From the proof of proposition 4.1 we conclude that a tree based system that can lexicalize context-free grammars must permit the distance between two nodes in the same tree to be increased during a derivation. In the next section, we suggest the use of an additional operation when defining a tree-based system in which one tries to lexicalize CFGs.

4.2 Lexicalization of CFGs with TAGs

Another combining operation is needed to lexicalize finitely ambiguous CFGs. As the previous examples suggest us, we need an operation that is capable of inserting a tree inside another one. We suggest using adjunction as an additional combining operation. A tree-based system that uses substitution and adjunction coincides with a tree-adjoining grammar (TAG).

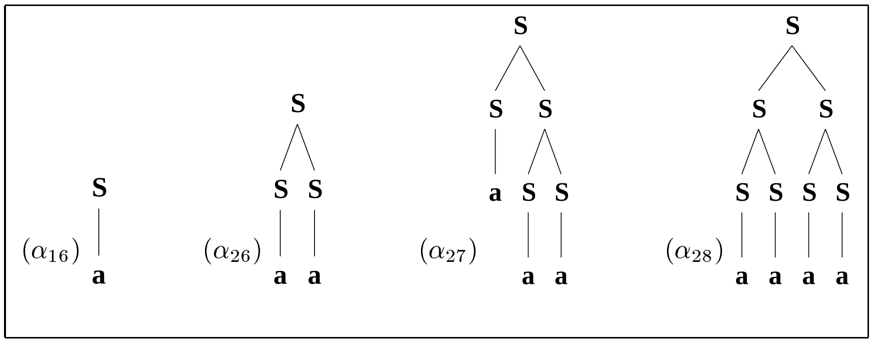
We first show that the CFGs in examples 4.1 and 4.2 for which TSG failed can be lexicalized within TAGs.

Example 4.3. Example 4.1 could not be lexicalized with TSG. It can be lexicalized by using adjunction as follows: [15]



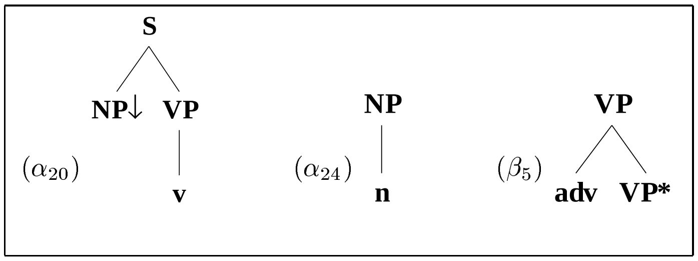
The auxiliary tree β_4 can now be inserted by adjunction inside the derived trees.

For example, the following derived trees can be derived by successive adjunction of β_4:

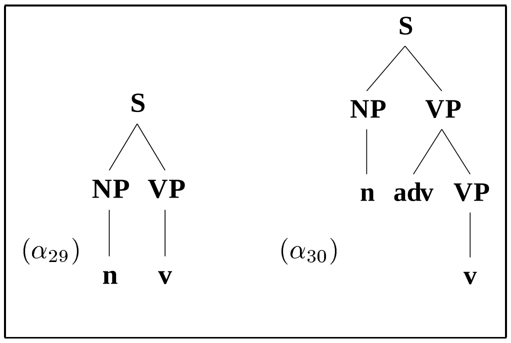


[15] a is taken as the lexical anchor of both the initial tree α_{16} and the auxiliary tree β_4.

Example 4.4. The CFG given in Example 4.2 can be lexicalized by using adjunction and one can choose the anchor freely:[16]



The auxiliary tree β_5 can be inserted in α_{20} at the VP node by adjunction. Using adjunction one is thus able to choose the appropriate lexical item as anchor. The following trees (α_{29} and α_{30}) can be derived by substitution of α_{24} into α_{20} for the NP node and by adjunction of β_5 on the VP node in α_{20}:



We are now ready to prove the main result: any finitely ambiguous context-free grammar can be lexicalized within tree-adjoining grammars; furthermore adjunction is the only operation needed. Substitution as an additional operation enables one to lexicalize CFGs in a more compact way.

Proposition 4.2. *If $G = (\Sigma, NT, P, S)$ is a finitely ambiguous CFG which does not generate the empty string, then there is a lexicalized tree-adjoining grammar $G_{lex} = (\Sigma, NT, I, A, S)$ generating the same language and tree set as G. Furthermore G_{lex} can be chosen to have no substitution nodes in any elementary trees.*

We give a constructive proof of this proposition. Given an arbitrary CFG, G, we construct a lexicalized TAG, G_{lex}, that generates the same language and tree set as G. The construction is not optimal with respect to time or the number of trees but it does satisfy the requirements.

[16] We chose v as the lexical anchor of α_{20} but, formally, we could have chosen n instead.

The idea is to separate the recursive part of the grammar G from the non-recursive part. The non-recursive part generates a finite number of trees, and we will take those trees as initial TAG trees. Whenever there is in G a recursion of the form $B \overset{*}{\Rightarrow} \alpha B \beta$, we will create an B-type auxiliary tree in which α and β are expanded in all possible ways by the non-recursive part of the grammar. Since the grammar is finitely ambiguous and since $\lambda \notin L(G)$, we are guaranteed that $\alpha\beta$ derives some lexical item within the non-recursive part of the grammar. The proof follows.

Proof. Proposition 4.2

Let $G = (\Sigma, NT, P, S)$ be a finitely ambiguous context-free grammar s.t. $\lambda \notin L(G)$. We say that $B \in NT$ is a *recursive symbol* if and only if $\exists \alpha, \beta \in (\Sigma \cup NT)^\star$ s.t. $B \overset{*}{\Rightarrow} \alpha B \beta$. We say that a production rule $B \rightarrow \delta$ is recursive whenever B is recursive.

The set of production rules of G can be partitioned into two sets: the set of recursive production rules, say $R \subseteq P$, and the set of non-recursive production rules, say $NR \subseteq P$; $R \cup NR = P$ and $R \cap NR = \emptyset$. In order to determine whether a production is recursive, given G, we construct a directed graph $\mathcal{G}$ whose nodes are labeled by non-terminal symbols and whose arcs are labeled by production rules. There is an arc labeled by $p \in P$ from a node labeled by B to a node labeled by C whenever p is of the form $B \rightarrow \alpha C \beta$, where $\alpha, \beta \in (\Sigma \cup NT)^\star$. Then, a symbol B is recursive if the node labeled by B in $\mathcal{G}$ belongs to a cycle. A production is recursive if there is an arc labeled by the production which belongs to a cycle.

Let $L(NR) = \{w | S \overset{*}{\Rightarrow} w$ using only production rules in $NR\}$. $L(NR)$ is a finite set. Since $\lambda \notin L(G)$, $\lambda \notin L(NR)$. Let I be the set of all derivation trees defined by $L(NR)$. I is a finite set of trees; the trees in I have at least one terminal symbol on the frontier since the empty string is not part of the language. I will be the set of initial trees of the lexicalized TAG G_{lex}.

We then form a base of minimal cycles of $\mathcal{G}$. Classical algorithms on graphs gives us methods to find a finite set of so-called 'base-cycles' such that any cycle is a combination of those cycles and such that they do not have any sub-cycle. Let $\{c_1 \cdots c_k\}$ be a base of cycles of $\mathcal{G}$ (each c_i is a cycle of $\mathcal{G}$).

We initialize the set of auxiliary trees of G_{lex} to the empty set, i.e. $A := \emptyset$. We repeat the following procedure for all cycles c_i in the base until no more trees can be added to A.

```
For all nodes $n_i$ in $c_i$, let $B_i$ be the label of $n_i$,
    According to $c_i$, $B_i \overset{*}{\Rightarrow} \alpha_i B_i \beta_i$,
    If $B_i$ is the label of a node in a tree in $I \cup A$ then
        for all derivations $\alpha_i \overset{*}{\Rightarrow} w_i \in \Sigma^\star$, $\beta_i \overset{*}{\Rightarrow} z_i \in \Sigma^\star$
        that use only non-recursive production rules
            add to $A$ the auxiliary tree corresponding to all derivations:
            $B_i \overset{*}{\Rightarrow} \alpha_i B_i \beta_i \overset{*}{\Rightarrow} w_i B_i z_i$ where the node labeled $B_i$ on the frontier is
            the foot node.
```

In this procedure, we are guaranteed that the auxiliary trees have at least one lexical item on the frontier, because $\alpha_i \beta_i$ must always derive some terminal symbol otherwise, the derivation $B_i \overset{*}{\Rightarrow} \alpha_i B_i \beta_i$ would derive a rule of the form $B_i \overset{*}{\Rightarrow} B_i$ and the grammar would be infinitely ambiguous.

It is clear that G_{lex} generates exactly the same tree set as G. Furthermore G_{lex} is lexicalized.

We just showed that adjunction is sufficient to lexicalize context-free grammars. However, the use of substitution as an additional operation to adjunction enables one to lexicalize a grammar with a more compact TAG.

5. Closure of TAGs under Lexicalization

In the previous section, we showed that context-free grammars can be lexicalized within tree-adjoining grammars. We now ask ourselves if TAGs are closed under lexicalization: given a finitely ambiguous TAG, G, ($\lambda \notin L(G)$), is there a lexicalized TAG, G_{lex}, which generates the same language and the same tree set as G? The answer is yes. We therefore establish that TAGs are closed under lexicalization. The following proposition holds:

Proposition 5.1 (TAGs are closed under lexicalization).
If G is a finitely ambiguous TAG that uses substitution and adjunction as combining operation, s.t. $\lambda \notin L(G)$, then there exists a lexicalized TAG G_{lex} which generates the same language and the same tree set as G.

The proof of this proposition is similar to the proof of proposition 4.2 and we only give a sketch of it. It consists of separating the recursive part of the grammar from the non-recursive part. The recursive part of the language is represented in G_{lex} by auxiliary trees. Since G is finitely ambiguous, those auxiliary trees will have at least one terminal symbol on the frontier. The non-recursive part of the grammar is encoded as initial trees. Since the empty string is not generated, those initial trees have at least one terminal symbol on the frontier. In order to determine whether an elementary tree is recursive, given G, we construct a directed graph $\mathcal{G}$ whose nodes are labeled by elementary trees and whose arcs are labeled by tree addresses. There is an arc labeled by ad from a node labeled by β to a node labeled by α whenever β can operate (by adjunction or substitution) at address ad in α. Then, an elementary tree δ is recursive if the node labeled by δ in $\mathcal{G}$ belongs to a cycle. The construction of the lexicalized TAG is then similar to the one proposed for proposition 4.2.

6. Summary of Lexicalization[17]

The elementary objects manipulated by a tree-adjoining grammar are trees, i.e., structured objects and not strings. The properties of TAGs relate directly to the strong generative capacity (structural description) which is more relevant to linguistic descriptions than the weak generative capacity (set of strings). The tree sets of TAGs are not recognizable sets but are equivalent to the tree sets of linear indexed languages. Hence, tree-adjoining grammars generate some context-sensitive languages. However, tree-adjoining languages are strictly contained in the class of indexed languages.

The lexicalization of grammar formalisms is of linguistic and formal interest. We have taken the point of view that rules should not be separated totally from their lexical realization. In this "lexicalized" approach, each elementary structure is systematically associated with a lexical anchor. These structures specify extended domains of locality (as compared to Context Free Grammars) over which constraints can be stated.

The process of lexicalization of context-free rules forces us to use operations for combining structures that make the formalism fall in the class of mildly context sensitive languages. Substitution and adjunction give us the freedom to lexicalize CFGs. Elementary structures of extended domain of locality, when they are combined with substitution and adjunction, yield Lexicalized TAGs. TAGs were so far introduced as an independent formal system. We have shown that they derive from the lexicalization process of context-free grammars. We also have shown that TAGs are closed under lexicalization.

Until recently it was an open problem whether or not there is a subclass of TAGs such that lexicalization can be achieved and yet this class was not more powerful than CFGs. This question has now been answered in the affirmative in [Schabes and Waters1995]. The tree insertion grammar they define

[17] There are several important papers about TAGs describing their linguistic, computational and formal properties. Some of these are: Joshi [Joshi1987], Joshi, Vijay-Shanker and Weir [Joshi et al.1991], Vijay-Shanker [Vijay-Shanker1987], Weir [Weir1988], Schabes [Schabes1990, Schabes1991], Schabes and Joshi [Schabes and Joshi1988, Schabes and Joshi1989], Kroch [Kroch1987], Kroch and Joshi [Kroch and Joshi1985], Abeillé, Bishop, Cote and Schabes [Abeillé et al.1990], Abeillé [Abeillé1988], Schabes and Waters [Schabes and Waters1995], Rambow, Vijay-Shanker and Weir [Rambow et al.1995], Joshi and Srinivas [Joshi and Srinivas1994], Rambow [Rambow1994], Vijay-Shanker [Vijay-Shanker1992], Shieber and Schabes [Shieber and Schabes1990]. A reader interested in TAGs will find these papers very useful. Additional useful references will be found in the Special Issue of *Computational Intelligence* (November 1994) [CI1994] devoted to Tree-Adjoining Grammars. A wide coverage lexicalized TAG grammar for English (about 300,000 inflected items and about 570 trees in 38 families) and a parser (XTAG System) has been described in [XTAG-Group1995], which includes evaluation of XTAG on corpora such as the Wall Street Journal, IBM Computer Manuals and ATIS Corpus.

is such a class. This class however cannot capture context-sensitive phenomena needed for the description of natural languages (for example, crossed dependencies), although it appears to be adequate for English for practical purposes.

7. Embedded Push-Down Automaton (EPDA)

We will now describe an automaton related to TAGs and briefly describe a processing application to crossed dependencies.

An EPDA, M', is very similar to a PDA, except that the push-down store is not necessarily just one stack but a sequence of stacks. The overall stack discipline is similar to a PDA, i.e., the stack head will be always at the top symbol of the top stack, and if the stack head ever reaches the bottom of a stack, then the stack head automatically moves to the top of the stack below (or to the left of) the current stack, if there is one (Vijay-Shanker, 1987; Joshi, 1987; Joshi, Vijay-Shanker, and Weir, 1988).

Initially, M' starts with only one stack, but unlike a PDA, an EPDA may create new stacks above and below (right and left of) the current stack. The behavior of M is specified by a transition function, δ', which for a given input symbol, the state of the finite control, and the stack symbol, specifies the new state, and whether the current stack is pushed or popped; it also specifies new stacks to be created above and below the current stack. The number of stacks to be created above and below the current stack are specified by the move. Also, in each one of the newly created stacks, some specified finite strings of symbols can be written (pushed). Thus:

$$\delta' \text{ (input symbol, current state, stack symbol)} =$$

$$(\text{new state}, sb_1, sb_2, \ldots, sb_m, \text{push/pop on current stack}, st_1, st_2, \ldots, st_n)$$

where $sb_1, sb_2, \ldots, sb_m$ are the stacks introduced below the current stack, and $st_1, st_2, \ldots, st_n$ are the stacks introduced above the current stack[18]. In each one of the newly created stacks, specified information may be pushed. For simplicity, we have not shown this information explicitly in the above definition. As in the case of a PDA, an EPDA can be nondeterministic also.

A string of symbols on the input tape is recognized (parsed, accepted) by M', if starting in the initial state, and with the input head on the leftmost symbol of the string on the input tape, there is a sequence of moves as specified by δ' such that the input head moves past the rightmost symbol on the input tape and the current stack is empty, and there are no more stacks below the current stack. Figures 7.1 and 7.2 illustrate moves of an EPDA, M'.

[18] The transition function must also specify whether the input head moves one symbol to the right or stays where it is.

Given the initial configuration as shown in (1), let us assume that for the given input symbol, the current state of the finite control, and the stack symbol, δ' specifies the move shown in (2):

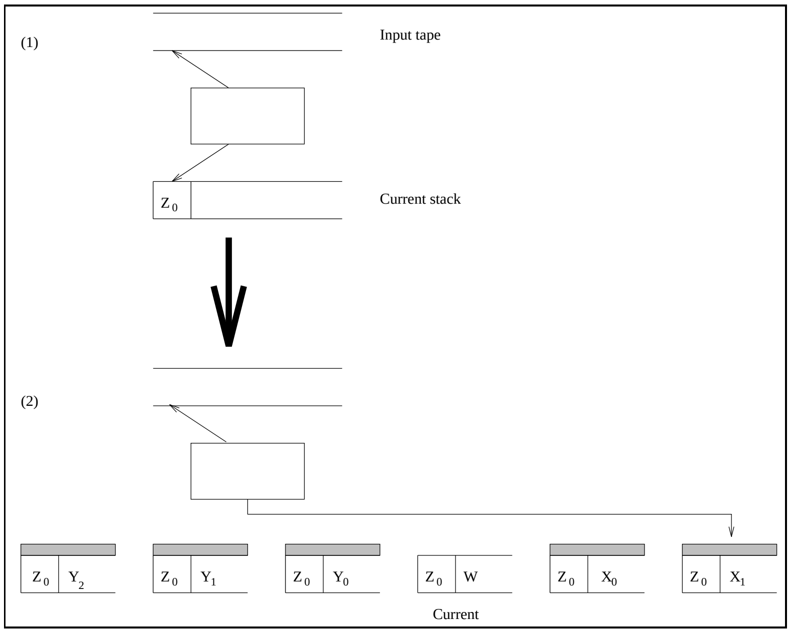


Fig. 7.1. Moves of an EPDA

In this move, two stacks have been created above (to the right of) the current stack (which is shown by dotted lines), and three stacks have been created below (to the left of) the current stack (i.e., the current stack in (1), the old current stack). W has been pushed on the current stack, X_0 and X_1, respectively, have been pushed on the two stacks introduced above the current stack, and Y_0, Y_1, and Y_2, respectively, have been pushed on the stacks created below the (old) current stack. The stack head has moved to the top of top stack, so now the topmost stack is the new current stack and the stack head is on the topmost symbol in the new current stack. We will use Z_0 to denote the bottom of each stack.

Let us assume that in the next move the configuration is as shown in (3) in Fig. 7.2. In this move, 1 stack has been created below the current stack (which is shown by dotted lines) with V_0 pushed on it, 2 stacks have been created above the (old) current stack with T_0 and T_1 pushed on them, respectively. V is pushed on the (old) current stack. The stack head has again moved to the topmost symbol of top stack, which is now the new current stack.

Thus in an EPDA in a given configuration there is a sequence of stacks; however, the stack head is always at the top of the top stack at the end of a

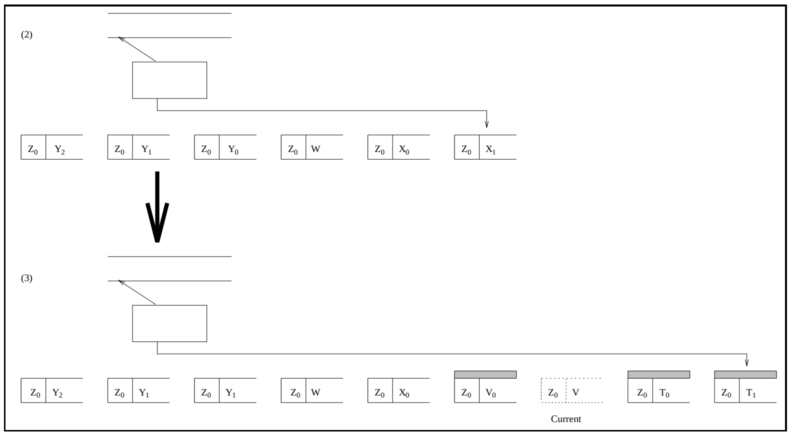


Fig. 7.2. Moves of an EPDA

move. Thus although, unlike a PDA, there is a sequence of stacks in a given configuration, the overall stack discipline is the same as in a PDA. PDAs are special cases of EPDAs, where in each move no new stacks are created, only a push/pop is carried out on the current stack. Note that in an EPDA, during each move, push/pop is carried out on the current stack and pushes on the newly created stacks. Since, in a given move, the information popped from the current stack may be identical to the information pushed on a newly created stack, we will have the effect of moving information from one stack to another. In this case the information, although popped from the current stack, is still in the EPDA. We will use the term POP (capitalized) to denote the case when information is popped from the current stack and it is not 'moved' to a newly created stack, i.e., the information is discharged from the EPDA and it is lost from the EPDA.

7.1 Crossed Dependencies

We will now illustrate how EPDAs can process crossed dependencies as they arise in certain languages, for example, Dutch. The analysis presented below not only recognizes the strings with crossed dependencies but also shows how the interpretation is done incrementally. The reader will also note that the machine shown in Fig. 7.3 can also recognize strings of the form $\{a^n b^n c^n \mid n \geq 1\}$ and $\{a^n b^n c^n d^n \mid n \geq 1\}$, which correspond to a mixture of crossed and nested dependencies.

Rather than defining the EPDA, M_d, formally, (i.e. specifying the transition function completely), we will describe simply the moves M_d goes through during the processing of the input string (see Fig. 7.3. The symbols in the input string are indexed so as to bring out the dependencies explicitly and thus the indexing is only for convenience. Also NPs are treated as single symbols. In the initial configuration, the input head is on NP_1 and the stack head is

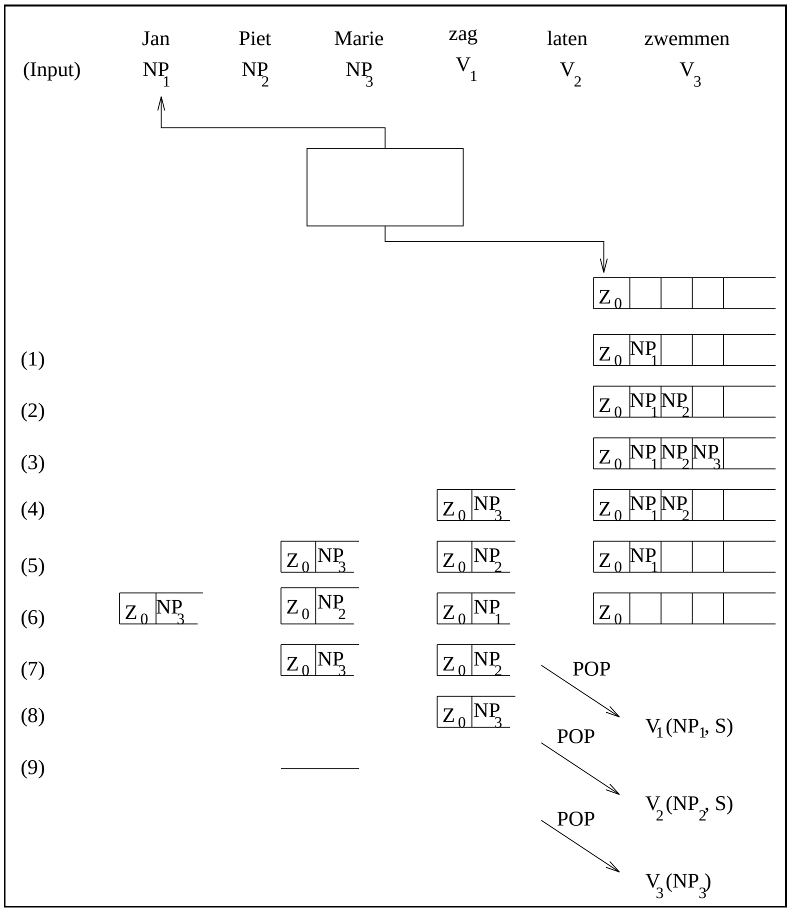


Fig. 7.3. Crossed Dependencies (Dutch)

on top of the current stack. The first three moves of M_d, i.e., moves 1, 2, and 3, push NP_1, NP_2, NP_3 on the stack. At the end of the third move, the current stack has NP_1, NP_2, and NP_3 on it and the input head is on V_1. No new stacks have been created in these moves. In move 4, NP_3 is popped from the current stack and a new stack has been created below the current stack and NP_3 is pushed on this stack, thus NP_3 is still within the EPDA and not POPPED out of the EPDA. At the end of move 4, the stack head is on top of the topmost stack, i.e., on NP_2 and the input head stays at V_1. Moves 5 and 6 are similar to move 4. In move 5, NP_2 is popped from the current stack and a new stack with NP_2 on it is created below the current stack. Thus the stack containing NP_2 appears between the stack containing NP_3 and the current stack. The input head stays at V_1. Similarly, in move 6, NP_1 is popped from the current stack and a new stack is created below the current stack, and NP_1 is pushed on it. The input head stays at V_1. The current stack is now empty and since there are stacks below the current stack, the stack head moves to the top of topmost stack below the empty current stack, i.e., it is on NP_1. In move 7, NP_1 is POPPED. In effect, we have matched V_1 from the input to NP_1 and the structure $V_1(NP_1, S)$ is now POPPED and NP_1 is no longer held by M_d[19]. $V_1(NP_1, S)$ denotes a structure encoding V_1 and its argument structure. Note that this structure has its predicate and one argument filled in, and it has a slot for an S type argument, which will be filled in by the next package that is POPPED by M_d. Thus we are following the principle of partial interpretation (PPI), as described in Section 1. Similarly in move 8, V_2 and NP_2 are matched and NP_2 is POPPED, i.e., the structure $V_2(NP_2, S)$ is POPPED. This structure now fills in the S argument of the structure POPPED earlier, and it itself is ready to receive a structure to fill its S argument. In move 9, V_3 and NP_3 are matched and NP_3 is POPPED, i.e., the structure $V_3(NP_3)$ is POPPED, which fills in the S argument of the structure previously POPPED. During the moves 7, 8, and 9, the input head moves one symbol to the right. Hence, at the end of move 9, the input head is past the rightmost symbol on the input tape; also, the current stack is empty and there are no stacks below the current stack. Hence, the input string has been successfully recognized (parsed).

8. Linguistic Relevance

In this section we will discuss very briefly the linguistic relevance of TAGs. The two formal properties discussed below are linguistically very crucial and they are mathematically also very interesting. Tree-adjoining grammars

[19] Although we are encoding a structure, only a bounded amount of information is stored in the EPDA stacks. The symbols *NP*, S, etc. are all atomic symbols. In an EPDA behaving as a parser, these symbols can be regarded as pointers to relevant structures, already constructed, and outside the EPDA.

(TAG) constitute a tree-generating formalism with some attractive properties for characterizing the strong generative capacity of grammars, that is, their capacity to characterize the structural descriptions associated with sentences. Among these properties the two most basic ones are as follows.

1. Extended domain of locality (EDL): TAGs have a larger domain of locality than context-free grammars (CFG) and CFG based grammars such as head-driven phrase structure grammars (HPSG) and lexical-functional grammars (LFG). In the CFG in Fig. 8.1 the dependency between the verb *likes* and its two arguments, subject (NP) and object (NP), is specified by means of the first two rules of the grammar. It is not possible to specify this dependency in a single rule without giving up the VP (verb phrase) node in the structure. That is, if we introduce the rule, S $\rightarrow$ NP V NP, then we express this dependency in one rule, but then we cannot have VP in the grammar. Hence, if we regard each rule of a CFG as specifying the domain of locality, then the domain of locality for a CFG cannot locally (i.e., in one rule) encode the dependency between a verb and its arguments, and still keep the VP node in the grammar. TAGs will permit the localization of these dependencies, including the so-called long-distance dependencies.

Syntactic Rules	*Lexical Rules*
1. **S $\rightarrow$ NP VP**	4. **NP $\rightarrow$ Harry**
2. **VP $\rightarrow$ VP ADV**	5. **NP $\rightarrow$ peanuts**
3. **VP $\rightarrow$ V NP**	6. **V $\rightarrow$ likes**
	7. **ADV $\rightarrow$ passionately**

Fig. 8.1. A Context-Free Grammar (CFG)

2. Factoring recursion from the domain of dependencies (FRD): The elementary structures of TAGs are the domains over which dependencies such as agreement, subcategorization, and filler-gap, for example, are stated. The long-distance behavior of some of these dependencies follows from the operation of 'adjoining', thus factoring recursion from the domain over which dependencies are initially stated.

All other properties of TAGs, mathematical, computational, and linguistic follow from EDL and FRD. TAGs belong to the so-called 'mildly context-sensitive grammars' (MCSG). Roughly speaking, MCSG are only slightly more powerful than CFGs and preserve all the properties of CFGs. This extra power is a corollary of EDL and FRD and appears to be adequate for characterizing various phenomena which require more formal power than CFGs. Parsing algorithms for CFGs have been extended to TAGs. Three other formal systems, linear indexed grammars, head grammars and combinatory categorial grammars, have been shown to be weakly equivalent, that is in terms of the sets of strings these systems generate.

The only operation in TAGs for composing trees is adjoining. Although adjoining can simulate substitution, by adding the operation of substitution explicitly, we obtain lexicalized TAGs, called LTAGs. LTAGs are formally equivalent to TAGs. Hence, we will describe LTAGs only. Moreover, LTAGs have direct linguistic relevance. All recent work, linguistic and computational, has been done in the framework of LTAGs.

As we have already discussed in Section 3., in a 'lexicalized' grammar, each elementary structure is associated with a lexical item called its 'anchor'. By 'lexicalized' we mean that in each structure there is a lexical item that is realized. In a lexicalized grammar the grammar consists of a lexicon where each lexical item is associated with a finite set of elementary structures for which that item is the anchor (multiple anchors are possible). In addition to these anchored structures we have a finite set of operations which tell us how these structures are composed. There are no rules as such in the grammar. The composition operations are universal in the sense that they are the same for all grammars in the class of lexicalized grammars.

In a CFG, if we take each rule in the grammar as an elementary structure then it is easily seen that, in general, a CFG is not a lexicalized grammar. A CFG, G, which is not in a lexicalized form cannot be lexicalized by a lexicalized grammar, G', using substitution as the only operation for composing structures, such that both G and G' generate the same strings and the same structural descriptions. If in addition to the operation of substitution we add the operation of adjoining, then any CFG can be lexicalized and, more interestingly, the resulting system of grammar is exactly the lexicalized TAG (LTAG) system, which is described below. Thus LTAGs arise naturally in the process of lexicalizing CFGs.

In the LTAG, G_1, in Fig. 8.2, each word is associated with a structure (tree) (the word serves as an anchor for the tree) which encodes the dependencies between this word and its arguments (and therefore indirectly its dependency on other words which are anchors for structures that will fill up the slots of the arguments).

Thus, for *likes*, the associated tree encodes the arguments of *likes* (that is, the two NP nodes in the tree for *likes*) and also provides slots in the structure where they fit. The trees for *Harry* and *peanuts* can be substituted respectively in the subject and object slots of the tree for *likes*. The tree for *passionately* can be inserted (adjoined) into the tree for *likes* at the VP node. Adjoinable trees, which have a root node and a foot node (marked with *) with the same label, are called auxiliary trees. The other elementary trees are called initial trees. Derivation in a TAG is quite different from a derivation in a CFG. The tree in Fig. 8.3 is a derived tree in G_1. It is not the derivation tree. The derivation tree (not shown here) will be a record of the history of

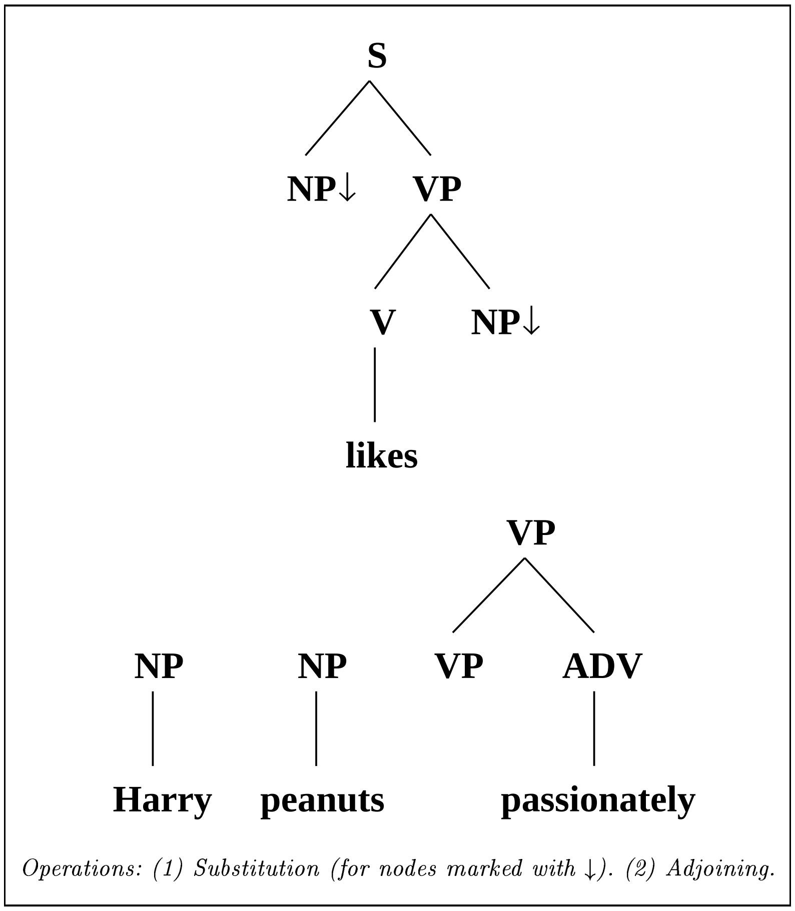

S
NP↓
VP
V
NP↓
likes
VP
NP
NP
VP
ADV
Harry
peanuts
passionately
Operations: (1) Substitution (for nodes marked with ↓). (2) Adjoining.

Fig. 8.2. Elementary Trees for a TAG, G_1.

the various adjoinings and substitutions carried out to produce the tree in Fig. 8.3.

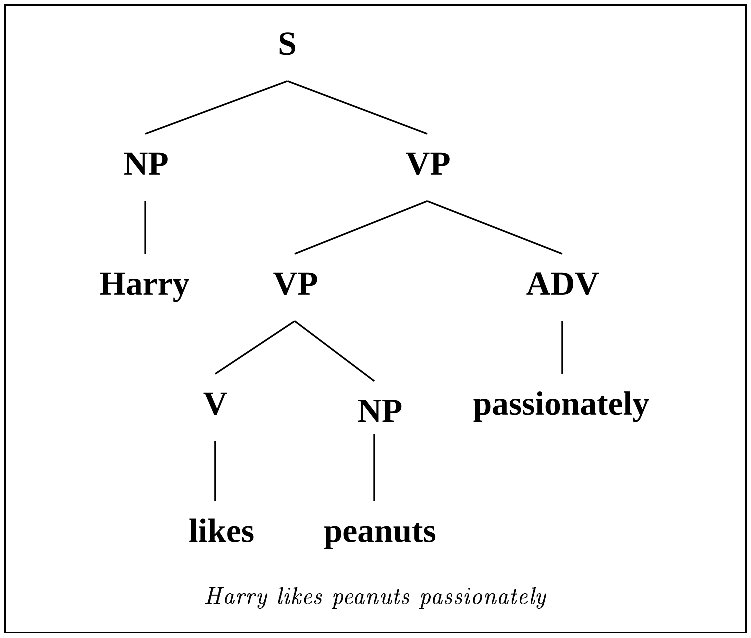


Fig. 8.3. A Derived Tree for the TGA, G_1.

Now consider the TAG, G_2, in Fig. 8.4. For simplicity the trees in G_2 are shown with the NP substitutions already carried out. By adjoining the tree for *tell* at the interior S' node of the tree for *likes* in G_2, we obtain the derived tree in Fig. 8.5 corresponding to *who*$_i$ *did John tell Sam that Bill likes* e_i Note that in the tree for *likes*, the dependent elements *who*$_i$ and the gap e_i both appear in the same elementary tree. In the derived tree in Fig. 8.5 the two dependent elements have moved apart and thus the dependencies have become long-distance.[20] This example illustrates the property FRD.

Using FRD, it can be shown that the principle of subjacency, a constraint on long-distance dependencies, which rules out sentences such as *Who*$_i$ *did you wonder why she wrote to* e_i, is a corollary of the fact that there cannot be an elementary tree (for English) which has two preposed WH-phrases. Thus, the long-distance constraint is replaced by a local constraint on an elementary tree, a constraint that is needed in any case to define the elementary trees that are allowed and those that are not. Further linguistic results using EDL

[20] The specific linguistic details in Figures 8.4 and 8.5 may be ignored by a reader not familiar with the Government and Binding Theory. The factoring of recursion from the domain of dependencies illustrated in these figures is independent of any particular grammatical theory.

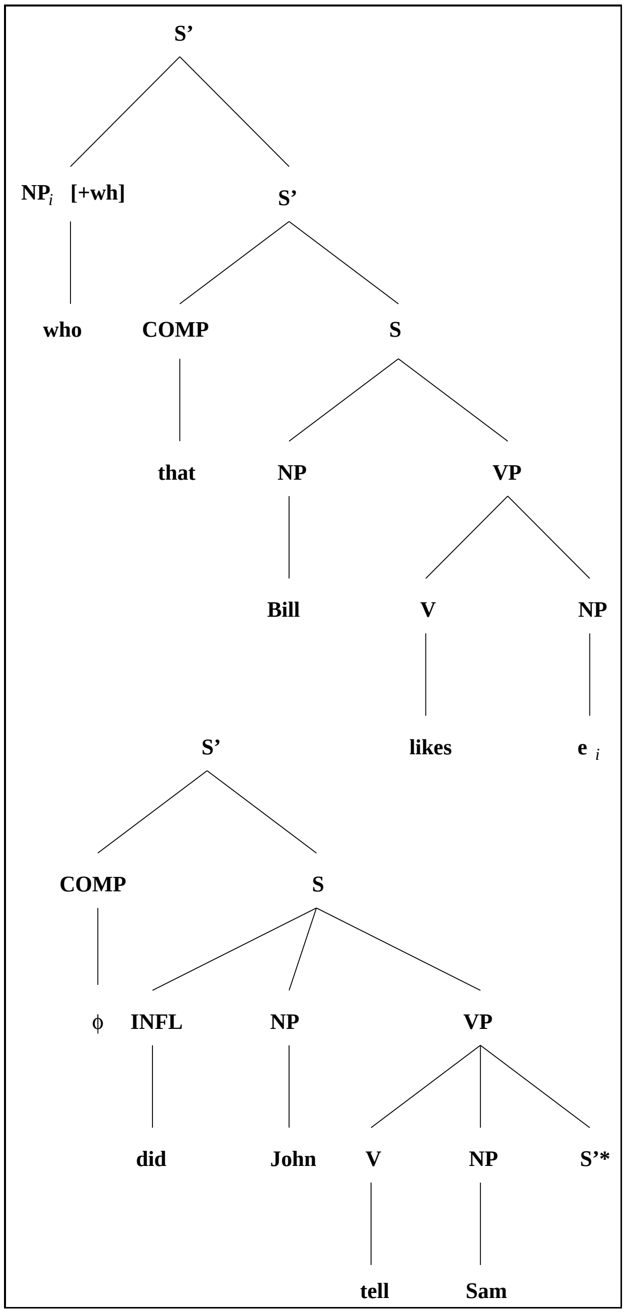


Fig. 8.4. Elementary Trees for a TAG, G_2.

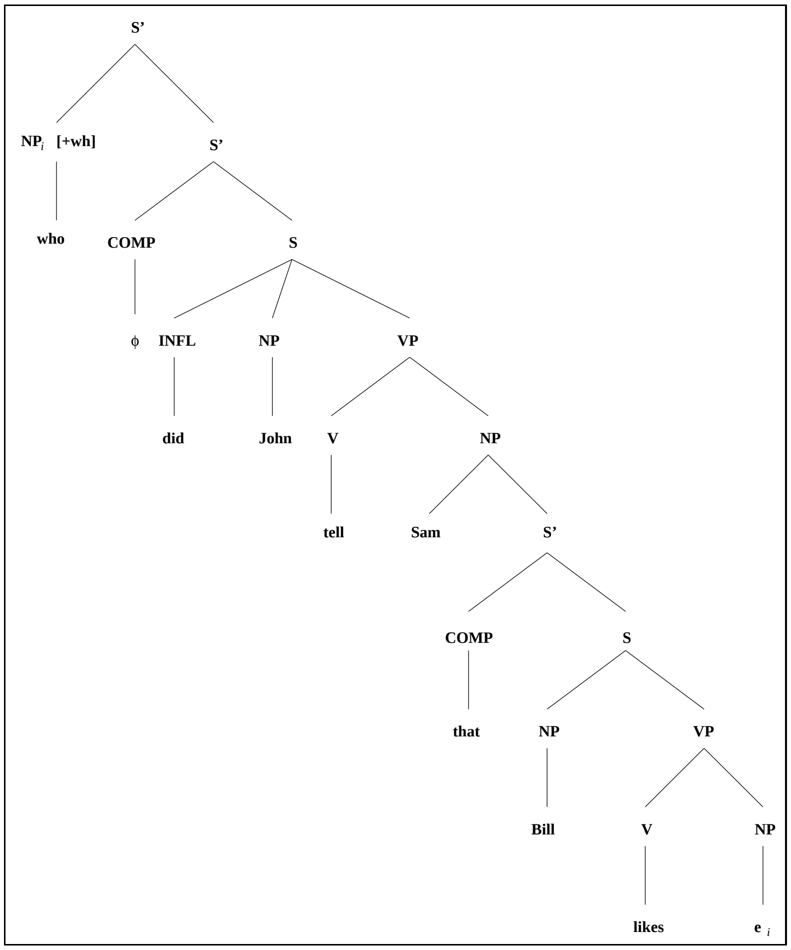


Fig. 8.5. A Derived Tree for the TAG, G_2.

and FRD have been obtained in the treatment of linguistic phenomena such as extraposition, asymmetries in long-distance extractions, the clause-final verb clusters in Dutch and German and idioms–their noncompositional and compositional aspects, among others.

9. Some variants of TAGs

9.1 Feature Structure Based TAGs

A feature structure consists of a set of attribute-value pairs, where a value may be atomic or another feature structure. The main operation for combining feature structures is unification. A variety of grammars such as GPSG, HPSG, and LFG are feature structure based grammars with CFG as their skeletons. A feature structure based TAG is a TAG in which feature structures are associated with the nodes of the elementary trees. The operations of substitution and adjoining are then defined in terms of unifications of appropriate feature structures, thus allowing the constraints on substitution and adjoining to be modeled by the success or failure of unifications. In contrast to feature structure based grammars with CFG skeletons, those with TAG skeletons need only finite-valued features due to EDL and FRD, thus reducing the role of unification as compared to CFG based systems [Joshi et al.1991].

9.2 Synchronous TAGs

Synchronous TAGs are a variant of TAGs, which characterize correspondences between languages [Shieber and Schabes1990]. Using EDL and FRD, synchronous TAGs allow the application of TAGs beyond syntax to the task of semantic interpretation, language generation and automatic translation. The task of interpretation consists of associating a syntactic analysis of a sentence with some other structure–a logical form representation or an analysis of a target language sentence. In a synchronous TAG both the original language and its associated structure are defined by grammars stated in the TAG formalism. The two TAGs are synchronized with respect to the operations of substitution and adjoining, which are applied simultaneously to related nodes in pairs of trees, one tree for each language. The left member of a pair is an elementary tree from the TAG for one language, say English, and the right member of the pair is an elementary tree from the TAG for another language, say the logical form language.

9.3 Probabilistic LTAGs

Probabilistic CFGs can be defined by associating a probability with each rule of the grammar. Then the probability of a derivation can be easily computed because each rewriting in a CFG derivation is independent of context and hence, the probabilities associated with the different rewriting rules can be multiplied. However, the rule expansions are, in general, not context-free. A probabilistic CFG can distinguish two words or phrases w and w' only if the probabilities $P(w/N)$ and $P(w'/N)$ as given by the grammar differ for some nonterminal. That is, all the distinctions made by a probabilistic CFG must be mediated by the nonterminals of the grammar. Representing distributional distinctions in nonterminals leads to an explosion in the number of parameters required to model the language. These problems can be avoided by adopting probabilistic TAGs, which provide a framework for integrating the lexical sensitivity of stochastic approaches and the hierarchical structure of grammatical systems [Resnick1994, Schabes1992]. Since every tree is associated with a lexical anchor, words and their associated structures are tightly linked. Thus the probabilities associated with the operations of substitution and adjoining are sensitive to lexical context. This attention to lexical context is not acquired at the expense of the independence assumption of probabilities because substitutions and adjoinings at different nodes are independent of each other. Further EDL and FRD allow one to capture locally the co-occurrences between, the verb *likes* and the head nouns of the subject and the object of *likes* even though they may be arbitrarily apart in a sentence.

9.4 Using Description Trees in TAG

A new perspective on TAGs is provided in [Vijay-Shanker1992] by providing a formalism for describing the trees in TAG. Roughly speaking an internal node of a tree in TAG is viewed as a pair of nodes (say *top* and *bottom*)one dominating the other. The root and the foot node of an auxiliary tree in a TAG can be identified with the *top* and *bottom* nodes respectively during the process of adjunction. Formal correspondence of this approach has been studied in [Vijay-Shanker1992] and the key ideas in this approach are also incorporated in D-Tree Grammars [Rambow et al.1995].

9.5 Muti-component TAGs (MCTAG)

In MCTAGs a variant of the adjoining operation is introduced under which, instead of a single auxiliary tree, a set of such trees is adjoined to a given elementary tree. Under this definition, MCTAGs are equivalent to TAGs both in terms of the strings and structural descriptions they generate [Joshi1987, Joshi et al.1991]. Such MCTAGs have been used in the analysis of extraposition as well as certain kinds of word order variations, for example scrambling [Rambow1994].

10. Parsing Lexicalized Tree-Adjoining Grammars (LTAG)

Although formal properties of tree-adjoining grammars have been investigated (Vijay-Shanker, 1987, Vijay-Shanker and Joshi, 1985)—for example, there is an $O(n^6)$-time CKY-like algorithm for TAGs—little attention has been put on the design of practical parser for TAG whose average complexity is superior to its worse case. In this Section we will present a predictive bottom-up parser for TAGs which is in the spirit of Earley's parser and we discuss modifications to the parsing algorithms that make it possible to handle extensions of TAGs such as constraints on adjunction, substitution, and feature structure representation for TAGs. We present this algorithm as it is both theoretically and practically important.

In 1985, Vijay-Shanker and Joshi introduced a CKY-like algorithm for TAGs which uses dynamic programming techniques. They established for the first time $O(n^6)$ time as an upper bound for parsing TAGs. The algorithm was implemented, but in our opinion the result was more theoretical than practical for several reasons. The algorithm assumes that elementary trees are binary branching. Most importantly, although it runs in $O(n^6)$ worst time, it also runs in $O(n^6)$ best time. As a consequence, the CKY algorithm is in practice very slow. The lack of any predictive information (i.e. top-down information) in this purely bottom-up parser disallows for a better behavior. We investigate the use of predictive information in the design of parsers for TAGs whose practical performance is superior to the worst case complexity. In order to use predictive information, any algorithm should have enough information to know which tokens are to be expected after a given left context. That type of information is in nature top-down and is therefore not available for pure bottom-up parsers as the CKY-type parser for TAGs. Our main objective is to define practical parsers for TAGs that are easy to modify to handle extensions of TAGs such as unification-based TAG.

Since the average time complexity of Earley's parser for CFGs depends on the grammar and in practice runs much better than its worst time complexity, we decided to try to adapt Earley's parser for CFGs to TAGs.

Finding an Earley-type parser for TAGs that use predictive information may seem a difficult task because it is not clear how to parse TAGs bottom up using top-down information while scanning the input string from left to right: since adjunction wraps a pair of strings around a string, parsing from left to right requires to remember more information is expected.

10.1 Left to Right Parsing of TAGs

Our main goal is to define a practical parser for tree-adjoining grammars. We are going to improve the performance of the parser by designing a bottom-up parser which uses top-down prediction. The predictive information will be used as the parser reads the input from left to right and will enable the parser to restrict the number of hypothesis it needs to assume. The most filtering that one might expect with some left to right predictive information is to rule out all hypotheses that are not consistent with the prefix of the input string that has been read so far. This notion is captured in the definition of the valid prefix property (see Section 10.8). However, the parser will not use all predictive information in order to lower its worst case complexity.

The algorithm relies on a tree traversal that enables one to scan the input string from left to right while recognizing adjunction.

This traversal is exemplified by the notion of *dotted tree* which consists of a node in a tree associated with a position relative to the node. The position is characterized as a *dot* at one of the following four possible positions: above or below and either to the left or to the right of the node. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below).

The tree traversal consists of moving the dot in the tree in a manner consistent with the left to right scanning of the yield while still being able to recognize adjunctions on interior nodes of the tree. The tree traversal starts when the dot is above and to the left of the root node and ends when the dot is above and to the right of the root node. At any time, there is only one dot in the dotted tree.

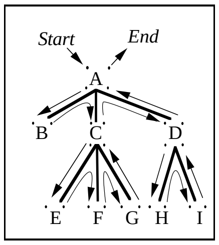


Fig. 10.1. Example of a tree traversal.

The tree traversal is illustrated in (see Fig. 10.1) and is precisely defined as follows:

- if the dot is at position la of an internal node (i.e. not a leaf), we move the dot down to position lb,
- if the dot is at position lb of an internal node, we move to position la of its leftmost child,
- if the dot is at position la of a leaf, we move the dot to the right to position ra of the leaf,
- if the dot is at position rb of a node, we move the dot up to position ra of the same node,
- if the dot is at position ra of a node, there are two cases:

- if the node has a right sibling, then we move the dot to the right sibling at position *la*.
- if the node does not have a right sibling, then we move the dot to its parent at position *rb*.

This traversal will enable us to scan the frontier of an elementary tree from left to right while trying to recognize possible adjunctions between the above and below positions of the dot.

In effect, a dotted tree separates a tree into two parts (see Fig. 10.2): a *left context* consisting of nodes that have been already traversed and a *right context* that still needs to be traversed.

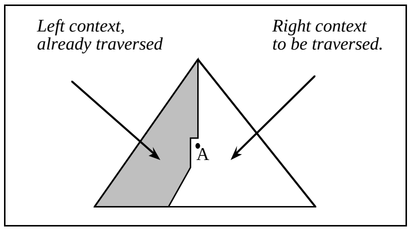

Left context, already traversed
Right context to be traversed.
A

Fig. 10.2. Left and right contexts of a dotted tree.

It is important to note that the nodes on the path from the root node to the dotted node have been traversed only to their left sides. This fact is significant since adjoining may add material on both sides of a node and is therefore more complex to recognize than concatenation. For those nodes, only the left parts of the auxiliary trees that were adjoining to them have been seen.

Suppose that we are trying to recognize the input $w_1w_2w_3w_4w_5$ (where each w_i are strings of tokens) which was obtained by adjoining an auxiliary tree β into the tree α (see Fig. 10.3).

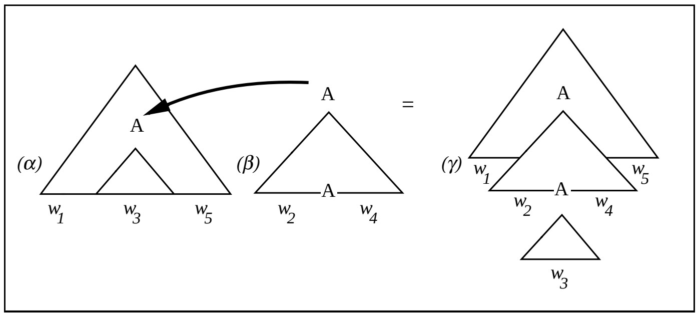


Fig. 10.3. An adjunction to be recognized.

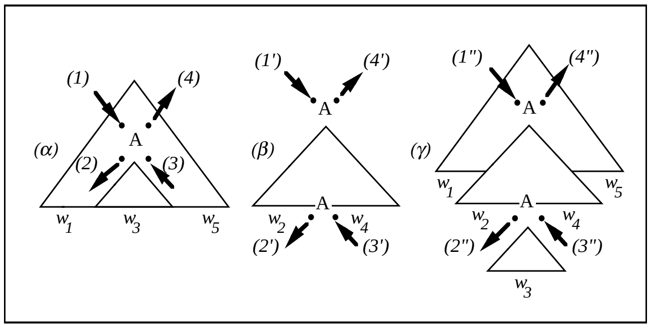


Fig. 10.4. Moving the dot while recognizing an adjunction.

Since the input is recognized from left to right, the algorithm must act as if it visits the resulting derived tree (rightmost tree in Fig. 10.3 and Fig. 10.4) from left to right. In particular, it should visit the nodes labeled by A in the following order (see Fig. 10.4)

$$1'' \cdots 2'' \cdots 3'' \cdots 4'' \cdots$$

This derived tree however is a composite tree, result of adjoining the second tree in Fig. 10.3 and Fig. 10.4 into the first tree in in Fig. 10.3 and Fig. 10.4. Since the algorithm never builds derived tree, it visits the A nodes in the following order (see Fig. 10.4)

$$1\ 1' \cdots 2'\ 2 \cdots 3\ 3' \cdots 4'\ 4 \cdots$$

In order to achieve this traversal across and within trees several concepts and data structures must be defined.

In the tree traversal defined above we consider equivalent (and therefore indistinguishable) two successive dot positions (according to the tree traversal) that do not cross a node in the tree (see Fig. 10.5). For example the following equivalences hold for the tree α pictured in Fig. 10.1:

$$< \alpha, 0, lb > \equiv < \alpha, 1, la >$$
$$< \alpha, 1, ra > \equiv < \alpha, 2, la >$$
$$< \alpha, 2, lb > \equiv < \alpha, 2 \cdot 1, la >$$

where $< \alpha, dot, pos >$ is the dotted tree in which the dot is at address dot and at position pos in the tree α.

We assume that the input string is $a_1 \cdots a_n$ and that the tree-adjoining grammar is $G = (\Sigma, NT, I, A, S)$: Σ is the finite set of terminal symbols, NT is the set of non-terminal symbols ($\Sigma \cap NT = \emptyset$), I is the set of initial trees, A is the set of auxiliary trees and S is the start non-terminal symbol.

The algorithm collects items into a chart.[21] An *item* s is defined as an 8-tuple, $s = [\alpha, dot, pos, i, j, k, l, sat?]$ where:

[21] We could have grouped the items into item sets as in [Earley1968] but we chose not to, allowing us to define an agenda driven parser.

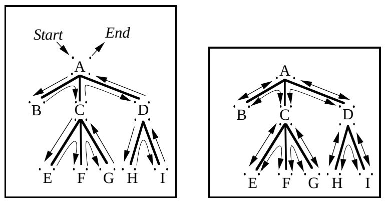


Fig. 10.5. *Left*, left to right tree traversal; *right*, equivalent dot positions.

- α is an elementary tree, initial or auxiliary tree: $\alpha \in I \cup A$.
- *dot* is an address in α. It is the address of the dot in the tree α.
- *pos* $\in$ {*la,lb, rb,ra*}. It is the position of the dot: to the left and above, or to the left and below, or to the right and below, or to the right and above.
- i, j, k, l are indices of positions in the input string ranging over $\{0, \cdots, n\} \cup \{-\}$, n being the length of the input string and $-$ indicating that the index is not bound. i and l are always bound, j and k can be simultaneously unbound. We will come back to these indices later.
- $sat?$ is a boolean; $sat? \in \{true, nil\}$. The boolean $sat?$ indicates whether an adjunction has been recognized on the node at address *dot* in the tree α. The algorithm may set $sat?$ to t only when $pos = rb$.

The components α, dot, pos of an item define a dotted tree. The additional indices i, j, k, l record portions of the input string. In the following, we will refer only to one of the two equivalent dot positions for a dotted tree. For example, if the dot at address *dot* and at position *pos* in the tree α is equivalent to the dot at address dot' at position pos' in α, then $s = [\alpha, dot, pos, i, j, k, l, sat?]$ and $s' = [\alpha, dot', pos', i, j, k, l, sat?]$ refer to the same item. We will use to our convenience s or s' to refer to this unique item.

It is useful to think of TAG elementary trees as a set of productions on pairs of trees and addresses (i.e. nodes). For example, the tree in Fig. 10.1, let's call it α, can be written as:

$$\begin{array}{l}(\alpha, 0) \rightarrow (\alpha, 1)\ (\alpha, 2)\ (\alpha, 3)\\ (\alpha, 2) \rightarrow (\alpha, 2 \cdot 1)\ (\alpha, 2 \cdot 2)\ (\alpha, 2 \cdot 3)\\ (\alpha, 3) \rightarrow (\alpha, 3 \cdot 1)\ (\alpha, 3 \cdot 2)\end{array}$$

Of course, the label of the node at address i in α is associated with each pair (α, i).[22] For example, consider the dotted tree $< \alpha, 2, ra >$ in which the dot is at address 2 and at position "right above" in the tree α (the tree in Fig. 10.1). Note that the dotted trees $< \alpha, 2, ra >$ and $< \alpha, 3, la >$ are equivalent. The dotted tree $< \alpha, 2, ra >$ is can be written in the following notation:

[22] TAGs could be defined in terms of such productions. However adjunction must be defined within this production system. This is not our goal, since we want to draw an analogy and not to define a formal system.

$$(\alpha,0) \rightarrow (\alpha,1)\ (\alpha,2)\ \bullet\ (\alpha,3)$$

One can therefore put into correspondence an item defined on a dotted tree with an item defined on a dotted rule. An item $s = [\alpha, dot, pos, i, j, k, l, sat?]$ can also be written as the corresponding dotted rule associated with the indices i, j, k, l and the flag $sat?$:

$\eta_0 \rightarrow \eta_1 \cdots \eta_y \bullet \eta_{y+1} \cdots \eta_z\ \ [i, j, k, l, sat?]$
$where\ \eta_0 = (\alpha, u)\ and\ \eta_p = (\alpha, u \cdot p),\ p \in [1, z]$

Here u is the address of the parent node (η_0) of the dotted node when $pos \in \{la, ra\}$, and where $u = dot$ when $pos \in \{lb, rb\}$.

The algorithm collects items into a set $\mathcal{C}$ called a *chart*. The algorithm maintains a property that is satisfied for all items in the chart $\mathcal{C}$. This property is pictured in Fig. 10.6 in terms of dotted trees. We informally describe it equivalently in terms of dotted rules. If an item of the form:

$\eta_0 \rightarrow \eta_1 \cdots \eta_y \bullet \eta_{y+1} \cdots \eta_z\ \ [i, j, k, l, sat?]\ with\ \eta_0 = (\alpha, u)\ and\ \eta_p = (\alpha, u \cdot p)$

is in the chart then the elementary tree α derives a tree such that:
$(i)\quad \eta_1 \cdots \eta_y \overset{*}{\Rightarrow} a_i \cdots a_l$
$(ii)\quad (\alpha, f) \overset{*}{\Rightarrow} a_{j+1} \cdots a_k$
where f is the address of the foot node of α. (ii) only applies when the foot node of α (if there is one) is subsumed by one of the nodes $\eta_1 \cdots \eta_y$. When no foot node is found below $\eta_1 \cdots \eta_y$, the indices j and k are not bound.

When $pos = rb$, the dot is at the end the dotted rule and if $sat? = t$ the boundaries $a_i \cdots a_l$ include the string introduced by an adjunction on the dotted tree.

The flag $sat?$ is needed since standard TAG derivations [Vijay-Shanker1987] disallow more than one adjunction on the same node. $sat? = t$ indicates that an adjunction was recognized on the dotted node (node at address dot in α). No more adjunction must be attempted on this node. $sat? = nil$ indicates that an auxiliary tree can still be adjoined on the dotted node.

Initially, the chart $\mathcal{C}$ consists of all items of the form $[\alpha, 0, la, 0, -, -, 0, nil]$, with α an initial tree whose root node is labeled with S. These initial items correspond to dotted initial trees with the dot above and to the left of the root node (at address 0).

Depending on the existing items in the chart $\mathcal{C}$, new items are added to the chart by four operations until no more items can be added to the chart. The operations are: Predict, Complete, Adjoin and Scan. If, in the final chart, there is an item corresponding to an initial tree completely recognized, i.e. with the dot to the right and above the root node which spans the input from position 0 to n (i.e. an item of the form $[\alpha, 0, ra, 0, -, -, n, nil]$), the input is recognized.

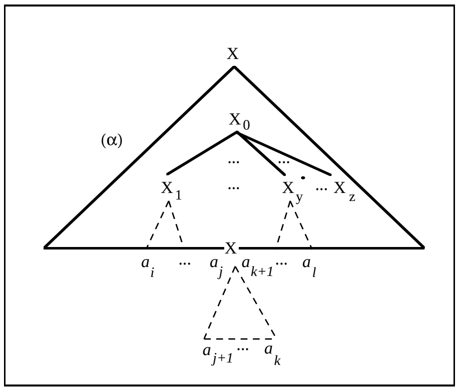


Fig. 10.6. Invariant.

The Scan (see Fig. 10.7) operation is a bottom-up operation that scans the input string. It applies when the dot is to the left and above a terminal symbol.

This operation consists of two cases. In the first case, the dotted node is to the left and above a node labeled by a non empty terminal which matches the next token to be expected. In this case, the dot is moved to the right and the span from i to l is increased by one (from i to $l+1$). The indices regarding the foot node (j and k) remain unchanged. In the second case, the dot is to the left and above an empty symbol. In this case, the dot is moved to the right and no token is consumed.

The Predict (see Fig. 10.7) operation is a top-down operation. It predicts new items accordingly to the left context that has already been seen.

It consists of three steps which may not always be applicable simultaneously. Step 1 applies when the dot is to the left and above a non-terminal symbol. All auxiliary trees adjoinable at the dotted node are predicted. Step 2 also applies when the dot is to the left and above a non-terminal symbol. If there is no obligatory adjoining constraint on the dotted node, the algorithm tries to recognize the tree without any adjunction by moving the dot below the dotted node. Step 3 applies when the dot is to the left and below the foot node of an auxiliary tree. The algorithm then considers all nodes on which the auxiliary tree could have been adjoined and tries to recognize the subtree below each one. It is in Step 3 of the Predict operation that the valid prefix property is violated since not all predicted nodes are compatible with the left context seen so far. The ones that are not compatible will be pruned in a later point in the algorithm (by the Complete operation). Ruling them out during this step requires more complex data-structures and increases the complexity of the algorithm [Schabes and Joshi1988].

The Complete (see Fig. 10.7) operation is a bottom-up operation that combines two items to form another item that spans a bigger portion of the input.

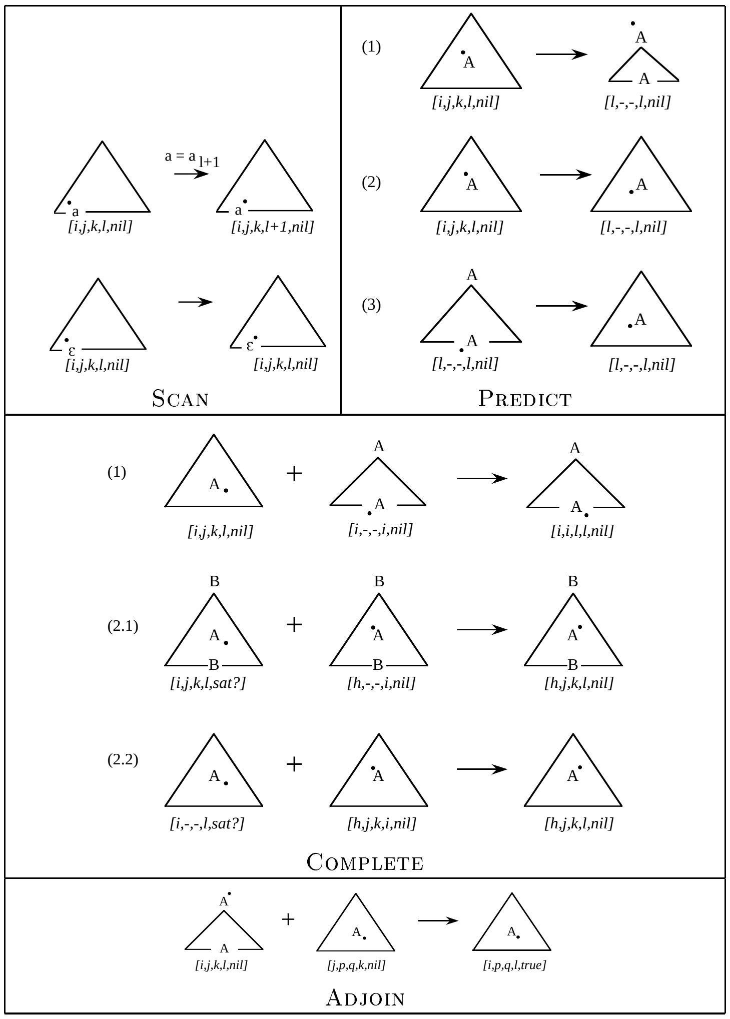


Fig. 10.7. The four operations of the parser.

It consists of two possibly non-exclusive steps that apply when the dot is at position *rb* (right below). Step 1 considers that the next token comes from the part to the right of the foot node of an auxiliary tree adjoined on the dotted node. Step 2 tries to further recognize the same tree and concatenate boundaries of two items within the same tree. Step 2 must be differentiated into two cases, one when the dotted node subsumes the foot node (in which case j and k are set when the dot is at position right below), the other when it does not subsume the foot node (in which case j and k are not set when the dot is at position right below).

The ADJOIN operation (see Fig. 10.7) is a bottom-up operation that combines two items by adjunction to form an item that spans a bigger portion of the input. It consists of a single step. The item produced has its flag (*sat*?) set to true since an adjunction was recognized on the dotted node. The setting of this flag prevents that the same operation applies a second time on the same item and therefore allows only one adjunction on a given node. This flag would be ignored for cases of alternative definition of tag derivations which allow for repeated adjunctions on a same node as suggested in [Schabes and Shieber1994].

10.2 The Algorithm

The algorithm is a bottom-up parser that uses top-down information. It is a general recognizer for TAGs with adjunction constraints. Unlike the CKY-type algorithm for TAGs, it requires no condition on the grammar: the elementary trees (initial or auxiliary) need not to be binary and they may have the empty string as frontier. The algorithm given below is off-line: it needs to know the length n of the input string before starting any computation. However it can very easily be modified to an on-line algorithm by the use of an end-marker in the input string.

The pseudo code for the recognizer is shown in Fig. 10.8. Each operation is stated as inference rules using the following notation:

$$\frac{\text{item1} \qquad\qquad \text{item2}}{\text{add item3}} \ \text{conditions}$$

It specifies that if the items above the horizontal line are present in the chart, then item below the horizontal line must be added to the chart only if the conditions of application are met.

In addition, we use the following notations. A *tree* α will be considered to be a function from tree addresses to symbols of the grammar (terminal and non-terminal symbols): if x is a valid address in α, then $\alpha(x)$ is the label of the node at address x in the tree α.

Addresses of nodes in a tree are encoded by Gorn-positions as defined by the following inductive definition: 0 is the address of the root node, k (k natural number) is the address of the k^{th} child of the root node,

$x \cdot y$ (x is an address, y a natural number) is the address of the y^{th} child of the node at address x.

Given a tree α and an address *add* in α, we define $Adj(\alpha, add)$ to be the set of auxiliary trees that can be adjoined at the node at address *add* in α; $OA(\alpha, dot)$ is defined as a boolean, true when the node at address *dot* in the tree α has an obligatory adjoining constraint, false otherwise; $Foot(\alpha)$ is defined as the address of the foot node of the tree α if there is one, otherwise $Foot(\alpha)$ is undefined. For TAGs with no constraints on adjunction, $Adj(\alpha, add)$ is the set of elementary auxiliary trees whose root node is labeled by $\alpha(add)$.

10.3 An Example

We give an example that illustrates how the recognizer works. The grammar used for the example (see Fig. 10.9) generates the language $L = \{a^n b^n e c^n d^n | n \geq 0\}$. It consists of an initial tree α and an auxiliary tree β. There is a null adjoining constraint on the root node and on the foot node of β.

The input string given to the recognizer is: *aabbeccdd*. The corresponding chart is shown in Fig. 10.10. The input is recognized since $[\alpha, 0, right, above, 0, -, -, 9, nil]$ is in the chart $\mathcal{C}$. For purpose of explanation, we have preceded each item with a number that uniquely identifies the item and the operation(s) that caused it to be placed into the chart are written to its right. We used the following abbreviations: *init* for the initialization step, *pred(k)* for the PREDICT operation applied to the item numbered by k, *sc(k)* for the SCAN operation applied to the item numbered by k, *compl(k+l)* for the COMPLETE operation applied to the items numbered by k and l and *adj(k+l)* for the ADJOIN operation applied to the items numbered by k and l. With this convention, one can trace step by step the construction of the chart. For example,

31. $[\beta, dot: 2, rb, 1, 4, 5, 8, t]$ *adj(30+24)*

stands for the item $[\beta, dot: 2, rb, 1, 4, 5, 8, t]$ uniquely identified by the number 31 which was placed into the chart by applying the ADJOIN operation on the items identified by the numbers 30 and 24.

10.4 Implementation

The algorithm described previously can be implemented to follow an arbitrary search space strategy by using a priority function that ranks the items to be processed. The ranking function can also be defined to obtain a left to right behavior as for context-free grammars [Earley1968].

In order to bound the worst case complexity as stated in the next section, arrays must be used to implement efficiently the different operations. Due to the lack of space we do not address these issues.

```
Let $G = (\Sigma, NT, I, A, S)$ be a TAG.
Let $a_1 \cdots a_n$ be the input string.

program recognizer
begin
 $\mathcal{C}$ = $\{[\alpha, 0, la, 0, -, -, 0, nil] | \alpha \in I, \alpha(0) = S\}$

  Apply the following operations on each item in the chart $\mathcal{C}$
  until no more items can be added to the chart $\mathcal{C}$:

    (1)  $\dfrac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, ra, i, j, k, l+1, nil]}$  $\alpha(dot) \in \Sigma, \alpha(dot) = a_{l+1}$
    (2)  $\dfrac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, ra, i, j, k, l, nil]}$  $\alpha(dot) \in \Sigma, \alpha(dot) = \epsilon$
    (3)  $\dfrac{[\alpha, dot, la, i, j, k, l, nil]}{[\beta, 0, la, l, -, -, l, nil]}$  $\alpha(dot) \in NT, \beta \in Adj(\alpha, dot)$
    (4)  $\dfrac{[\alpha, dot, la, i, j, k, l, nil]}{[\alpha, dot, lb, i, j, k, l, nil]}$  $\alpha(dot) \in NT, OA(\alpha, dot) = false$
    (5)  $\dfrac{[\alpha, dot, lb, l, -, -, l, nil]}{[\delta, dot', lb, l, -, -, l, nil]}$  $dot = Foot(\alpha), \alpha \in Adj(\delta, dot')$
    (6)  $\dfrac{[\alpha, dot, rb, i, j, k, l, nil] \quad [\beta, dot', lb, i, -, -, i, nil]}{[\beta, dot', rb, i, i, l, l, nil]}$  $dot' = Foot(\beta)$,
                                                                         $\beta \in Adj(\alpha, dot)$
    (7)  $\dfrac{[\alpha, dot, rb, i, j, k, l, true] \quad [\alpha, dot, la, h, j', k', i, sat?]}{[\alpha, dot, ra, h, j \cup j', k \cup k', l, sat?]}$  $\alpha(dot) \in NT$
    (8)  $\dfrac{[\beta, 0, ra, i, j, k, l, nil] \quad [\alpha, dot, rb, j, p, q, k, nil]}{[\alpha, dot, rb, i, p, q, l, true]}$  $\beta \in Adj(\alpha, dot)$

  If there is an item of the form $[\alpha, 0, ra, 0, -, -, n, nil]$ in $\mathcal{C}$ with $\alpha \in I$
  and $\alpha(0) = S$ then return acceptance, otherwise return rejection.
end.
```

Fig. 10.8. Pseudo-code for the recognizer

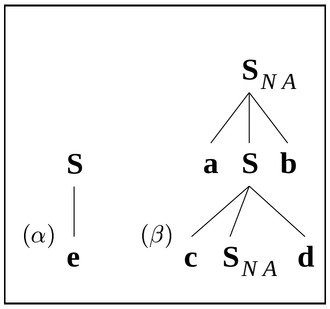


Fig. 10.9. TAG generating $L = \{a^n b^n e c^n d^n | n \geq 0\}$

Input read	Items in the chart
	1. $[\alpha, dot: 0, la, 0, -, -, 0, nil]$ *init*
	2. $[\beta, dot: 0, la, 0, -, -, 0, nil]$ *pred(1)*
	3. $[\alpha, dot: 1, la, 0, -, -, 0, nil]$ *pred(1)*
	4. $[\beta, dot: 1, la, 0, -, -, 0, nil]$ *pred(2)*
a	*5.* $[\beta, dot: 2, la, 0, -, -, 1, nil]$ *sc(4)*
a	*6.* $[\beta, dot: 0, la, 1, -, -, 1, nil]$ *pred(5)*
a	*7.* $[\beta, dot: 2.1, la, 1, -, -, 1, nil]$ *pred(5)*
a	*8.* $[\beta, dot: 1, la, 1, -, -, 1, nil]$ *pred(6)*
aa	*9.* $[\beta, dot: 2, la, 1, -, -, 2, nil]$ *sc(8)*
aa	*10.* $[\beta, dot: 0, la, 2, -, -, 2, nil]$ *pred(9)*
aa	*11.* $[\beta, dot: 2.1, la, 2, -, -, 2, nil]$ *pred(9)*
aa	*12.* $[\beta, dot: 1, la, 2, -, -, 2, nil]$ *pred(10)*
aab	*13.* $[\beta, dot: 2.2, la, 2, -, -, 3, nil]$ *sc(11)*
aab	*14.* $[\beta, dot: 2.2, lb, 3, -, -, 3, nil]$ *pred(13)*
aab	*15.* $[\beta, dot: 2.1, la, 3, -, -, 3, nil]$ *pred(14)*
aab	*16.* $[\alpha, dot: 1, la, 3, -, -, 3, nil]$ *pred(14)*
aabb	*17.* $[\beta, dot: 2.2, la, 3, -, -, 4, nil]$ *sc(15)*
aabb	*18.* $[\beta, dot: 2.2, lb, 4, -, -, 4, nil]$ *pred(17)*
aabb	*19.* $[\beta, dot: 2.1, la, 4, -, -, 4, nil]$ *pred(18)*
aabb	*20.* $[\alpha, dot: 1, la, 4, -, -, 4, nil]$ *pred(18)*
aabbe	*21.* $[\alpha, dot: 0, rb, 4, -, -, 5, nil]$ *sc(20)*
aabbe	*22.* $[\beta, dot: 2.2, rb, 4, 4, 5, 5, nil]$ *comp(21+18)*
aabbe	*23.* $[\beta, dot: 2.3, la, 3, 4, 5, 5, nil]$ *compl(22+15)*
aabbec	*24.* $[\beta, dot: 2, rb, 3, 4, 5, 6, nil]$ *sc(23)*
aabbec	*25.* $[\beta, dot: 2.2, rb, 3, 3, 6, 6, nil]$ *compl(24+14)*
aabbec	*26.* $[\beta, dot: 2.3, la, 2, 3, 6, 6, nil]$ *compl(25+13)*
aabbecc	*27.* $[\beta, dot: 2, rb, 2, 3, 6, 7, nil]$ *sc(26)*
aabbecc	*28.* $[\beta, dot: 3, la, 1, 3, 6, 7, nil]$ *compl(26+9)*
aabbeccd	*29.* $[\beta, dot: 0, rb, 1, 3, 6, 8, nil]$ *sc(28)*
aabbeccd	*30.* $[\beta, dot: 0, ra, 1, 3, 6, 8, nil]$ *compl(28+6)*
aabbeccd	*31.* $[\beta, dot: 2, rb, 1, 4, 5, 8, t]$ *adj(30+24)*
aabbeccd	*32.* $[\beta, dot: 3, la, 0, 4, 5, 8, nil]$ *compl(31+5)*
aabbeccdd	*33.* $[\beta, dot: 0, rb, 0, 4, 5, 9, nil]$ *sc(32)*
aabbeccdd	*34.* $[\beta, dot: 0, ra, 0, 4, 5, 9, nil]$ *compl(33+2)*
aabbeccdd	*35.* $[\alpha, dot: 0, rb, 0, -, -, 9, t]$ *adj(34+21)*
aabbeccdd	*36.* $[\alpha, dot: 0, ra, 0, -, -, 9, nil]$ *compl(35+1)*

Fig. 10.10. Items constituting the chart for the input: $_0\, a\, _1\, a\, _2\, b\, _3\, b\, _4\, e\, _5\, c\, _6\, c\, _7\, d\, _8\, d\, _9$

10.5 Complexity

The algorithm is a general parser for TAGs with constraints on adjunction that takes in worst case $O(|A||A \cup I|Nn^6)$ time and $O(|A \cup I|Nn^4)$ space, n being the length of the input string, $|A|$ the number of auxiliary trees, $|A \cup I|$ the number of elementary trees in the grammar and N the maximum number of nodes in an elementary tree. The worst case complexity is similar to the CKY-type parser for TAG [Vijay-Shanker1987, Vijay-Shanker and Joshi1985].[23]

The worst case complexity can be reached by the ADJOIN operation. An intuition of the validity of this result can be obtained by observing that the ADJOIN operation

$$\frac{[\beta, 0, ra, i, j, k, l, nil] \quad [\alpha, dot, rb, j, m, n, k, nil]}{[\alpha, dot, rb, i, m, n, l, true]} \; \beta \in Adj(\alpha, dot)$$

may be called at most $|A||A \cup I|Nn^6$ time since there are at most n^6 instances of the indices (i, j, k, l, m, n) and at most $|A| \times |A \cup I|N$ (α, β, dot) pairs of dotted trees to combine. When it is used with unambiguous tree-adjoining grammars, the algorithm takes at most $O(|A||A \cup I|Nn^4)$-time and linear time on a large class of grammars.

It is possible to achieve $O(|A \cup I|Nn^6)$-time worst complexity, however the implementation for such bounds requires complex data structures.

10.6 The Parser

The algorithm we described so far is a recognizer. However, if we include pointers from an item to a set of items of set of item pairs (pairs of items for the COMPLETE and the ADJOIN operation, or item for the SCAN and the PREDICT operations) which caused it to be placed in the chart (in a similar manner to that shown in Fig. 10.10), the recognizer can be modified to record all parse trees of the input string.

Instead of storing items of the form $[\alpha, dot, pos, i, j, k, l, sat?]$ in the chart, the parser stores items with a set of pairs or singletons of other items that caused them to exist. The fact that the same item may be added more than once reflects the fact that the item can be obtained in more than one way. This corresponds to local or global ambiguity. Therefore, when an item is added, if the item is already in the chart, the new items that caused the same item to exist are added to the set of causes.[24]

[23] Recently Satta [Satta1994] was able to transfer the complexity bound of TAG parsing to the one of matrix multiplication. As a consequence, it is shown that if one were to improve the bound of $O(n^6)$-time for the TAG parsing problem, one would have implicitly improved upon the bound of $O(n^3)$-time for matrix multiplication. Although this is possible, it can require very elaborate (and non practical) techniques.

[24] These operations can be done in constant time since if an item is added more than once, each of the pairs (or singletons) of items that caused it to be places on the chart are distinct.

The worst case time complexity for the parser is the same as for the recognizer ($O(|A||A \cup I|Nn^6)$-time) since keeping track of the source of each item does not introduce any overhead. However, the worst case space complexity increases to $O(|A||A \cup I|Nn^6)$-space since each cause of existence must be recorded. Due to the nature of each operation, the additional space required to record the derivation is not worse than $O(|A||A \cup I|Nn^6)$. For example, in the case of the Adjoin operation,

$$\frac{[\beta, 0, ra, i, j, k, l, nil] \quad [\alpha, dot, rb, j, m, n, k, nil]}{[\alpha, dot, rb, i, m, n, l, true]} \ \beta \in Adj(\alpha, dot)$$

for a given item $[\alpha, dot, rb, i, m, n, l, true]$ there can be at most $O(|A||A \cup I|)$ pairs of the form $([\beta, 0, ra, i, j, k, l, nil], [\alpha, dot, rb, j, m, n, k, nil])$ which need to be stored.

Once the recognizer has succeeded, it has encoded all possible parses in the form of a graph (encoded with those pairs of causes) which takes in the worst case $O(|G|^2Nn^6)$-space. All the derivations can then be enumerated by tracing back the causes of each item. Of course, the enumeration of all the derivations may take exponential time when the string is exponentially ambiguous or may not terminate when the input is infinitely ambiguous.

10.7 Parsing Substitution

TAGs use adjunction as their basic composition operation. As a consequence tree-adjoining languages (TALs) are mildly context-sensitive and properly contain context-free languages.[25]

Substitution of non-terminal symbols is the basic operation used in CFG. Substitution can be very easily extended to trees and has been found to be a useful additional operation for obtaining appropriate structural descriptions [Abeillé1988].

Substitution of trees is defined to take place on specified nodes on the frontiers of elementary trees. When a node is marked to be substituted, no adjunction can take place on that node. Furthermore, substitution is always mandatory. Only trees derived from initial trees rooted by a node of the same label can be substituted on a substitution node. The resulting tree is obtained by replacing the node by the tree derived from the initial tree.

The parser can be extended very easily to handle substitution. The algorithm must be modified as follows.

First, the ADJ function must disallow any adjunction to be taken place on nodes marked for substitution.

[25] It is also possible to encode a context-free grammar with auxiliary trees using adjunction only. However, although the languages correspond, the possible encoding does not reflect directly the original context free grammar since this encoding uses adjunction.

Then, more operations must be added to the parser: PREDICT SUBSTITUTION and COMPLETE SUBSTITUTION. These two operations are explained in details below.

Given a tree α and an address *add* in α, assuming that the node at address *add* in α is marked for substitution, we define $Substit(\alpha, add)$ to be the set of initial trees that can be substituted at the node at address *add* in α. For TAGs with no constraints on substitution, $Substit(\alpha, add)$ is the set of elementary initial trees whose root node is labeled by $\alpha(add)$.

PREDICT SUBSTITUTION operation predicts all possible initial trees that can be substituted at a node marked for substitution.

$$\frac{[\alpha, dot, la, i, j, k, l, sat?]}{[\alpha', 0, la, l, -, -, l, nil]} \quad \alpha' \in Substit(\alpha, dot) \tag{10.1}$$

COMPLETE SUBSTITUTION is a bottom-up operation that combines two items by substitution.

$$\frac{[\alpha, nil, ra, l, -, -, m, nil][\alpha', dot', la, i, j, k, l, sat?]}{[\alpha', dot', ra, i, j, k, m, sat?]} \quad \alpha' \in Substit(\alpha, dot) \tag{10.2}$$

The introduction of PREDICT SUBSTITUTION and of COMPLETE SUBSTITUTION does not increase the worst case complexity of the overall TAG parser.

10.8 The Valid Prefix Property and Parsing of Tree-Adjoining Grammar

The valid prefix property, the capability of a left to right parser to detect errors “as soon as possible”, is often unobserved in parsing CFGs. Earley's parser for CFGs [Earley1968] maintains the valid prefix property and obtains a worst case complexity ($O(n^3)$-time) as good as parsers that do not maintain it, such as the CKY parser [Younger1967, Kasami1965]. This follows from the path set complexity, as we will see.

A parser that satisfies the valid prefix property will only consider hypotheses consistent with the input seen so far. More precisely, parsers satisfying the *valid prefix property* guarantee that, as they read the input from left to right, the substrings read so far are valid prefixes of the language defined by the grammar: if the parser has read the tokens $a_1 \cdots a_k$ from the input $a_1 \cdots a_k a_{k+1} \cdots a_n$, then it is guaranteed that there is a string of tokens $b_1 \cdots b_m$ (b_i may not be part of the input) with which the string $a_1 \cdots a_k$ can be suffixed to form a string of the language; i.e. $a_1 \cdots a_k b_1 \cdots b_m$ is a valid string of the language.[26]

[26] The valid prefix property is independent from the *on-line* property. An on-line left to right parser is able to output for each new token read whether the string

The valid prefix property for an algorithm implies that it must have some top-down component that enables it to compute which tokens are to be expected as the input string is read. Pure bottom-up parsers as the CKY-type parsers[27] cannot have the valid prefix property since they do not use any top-down information.

Maintaining the VPP requires a parser to recognize the possible parse trees in a prefix order. The prefix traversal of the output tree consists of two components: a top-down component that expands a constituent to go to the next level down, and a bottom-up component that reduces a constituent to go to the next level up. When the VPP is maintained, these two components must be constrained together.

Context-free productions can be expanded independently of their context, in particular, independently of the productions that subsume them. The path set (language defined as the set of paths from root to frontier of all derived trees) of CFGs is therefore a regular set.[28] It follows that no additional complexity is required to correctly constrain the top-down and bottom-up behavior required by the prefix traversal of the parse tree: the expansion and the reduction of a constituent.

Contrary to CFGs, maintaining the valid prefix property for TAGs seems costly. Two observations corroborate this statement and an explanation can be found in the path set complexity of TAG.

Our first observation was that the worst case complexity of parsers for TAG that maintain the VPP is higher than the parsers that do not maintain VPP. Vijay-Shanker and Joshi [Vijay-Shanker and Joshi1985][29] proposed a CKY-type parser for TAG that achieves $O(n^6)$-time worst case complexity. As the original CKY parser for CFGs, this parser does not maintain the VPP. The Earley-type parser developed for TAGs [Schabes and Joshi1988] is bottom-up and uses top-down prediction. It maintains the VPP at a cost to its worst case complexity — $O(n^9)$-time in the worst case. Other parsers for TAGs have been proposed [Lang1988, Lavelli and Satta1991, Vijay-Shanker and Weir1990]. Although they achieve $O(n^6)$ worst case time complexity, none of these algorithms satisfies the VPP. To our knowledge, Schabes and Joshi's parser (1988) is the only known polynomial-time parser for TAG which satisfies the valid prefix property. It is still an open problem whether a better worst case complexity can be obtained for parsing TAGs while maintaining the valid prefix property.

seen so far is a valid string of the language. The valid prefix property is also sometimes referred as the error detecting property because it implies that errors can be detected as soon as possible. However, the lack of VPP does not imply that errors are undetected.

[27] We exclude any use of top-down information, even precompiled before run-time.

[28] This result follows from Thatcher [Thatcher1971], who defines frontier to root finite state tree automata.

[29] The parser is also reported in Vijay-Shanker [Vijay-Shanker1987].

The second observation is in the context of deterministic left to right parsing of TAGs [Schabes and Vijay-Shanker1990] where it was for the first time explicitly noticed that VPP is problematic to obtain. The authors were not able to define a bottom-up deterministic machine that satisfies the valid prefix property and which recognizes exactly tree-adjoining languages when used non-deterministically. Instead, they used a deterministic machine that does not satisfy the VPP, the bottom-up embedded push-down automaton. However, that machine recognizes exactly tree-adjoining languages when used non-deterministically.

The explanation for the difficulty of maintaining the VPP can be seen in in the complexity of the path set of TAGs. Tree-adjoining grammars generate some languages that are context-sensitive. The path set of a TAG is a context-free language [Weir1988] and is therefore more powerful than the path set of a CFG. Therefore in TAGs, the expansion of a branch may depend on the parent super-tree, i.e. what is above this branch. Going bottom-up, these dependencies can be captured by a stack mechanism since trees are embedded by adjunction. However, if one would like to maintain the valid prefix property, which requires traversing the output tree in a prefix fashion, the dependencies are more complex than a context-free language and the complexity of the parsing algorithm increases.

For example, consider the trees α, β and γ in Fig. 10.11. When γ is adjoined into β at the B node, and the result is adjoined into α at the A node, the resulting tree yields the string *ux'zx"vy"ty'w* (see Fig. 10.11).

If this TAG derived tree is recognized purely bottom-up from leaf to root (and therefore without maintaining the VPP), a stack based mechanism suffices for keeping track of the trees to which to algorithm needs to come back. This is illustrated by the fact that the tree domains are embedded (see bottom left tree in Fig. 10.11) when they are read from leaf to root in the derived tree.

However, if this derivation is recognized from left to right while maintaining the valid prefix property, the dependencies are more complex and can no longer be captured by a stack (see bottom right tree in Fig. 10.11).

The context-free complexity of the path set of TAGs makes the valid prefix property harder to maintain. We suspect that the same difficulty arises for context-sensitive formalism which use operations such as adjoining or wrapping [Joshi et al.1991].

In conclusion, Earley's parser for context-free grammars has been extended to tree-adjoining grammars. The notion of dotted rule was extended to tree and a left to right tree traversal was designed to recognize adjunction while reading the input from left to right. The parser for tree-adjoining grammars achieves $O(|A||A \cup I|Nn^6)$ time in the worst case. However, because of predictive information based on the prefixes of the input, the parser behaves in practice much faster than its worst case.

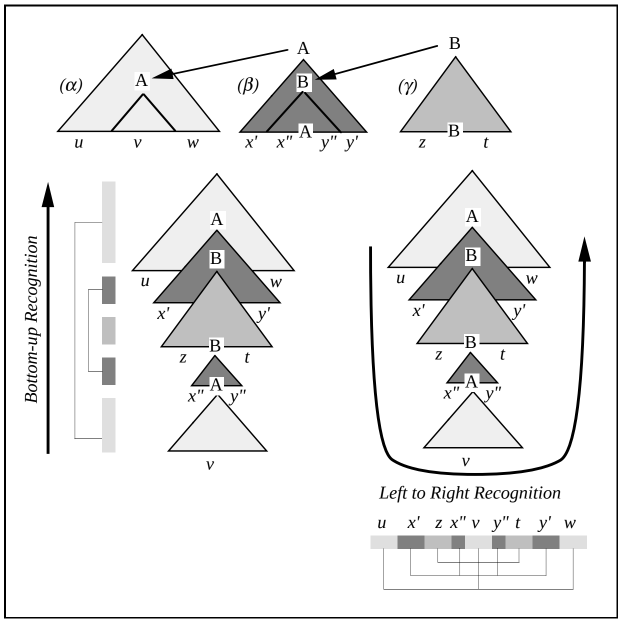


Fig. 10.11. *Above*, a sequence of adjunctions; *below left*, bottom-up recognition of the derived tree; *right*, left to right recognition of the derived tree.

This parser also handles various extensions of tree-adjoining grammars such as adjoining constraints and feature based tree-adjoining grammars [Vijay-Shanker and Joshi1988]. Its performance is further improved by a strategy which uses the lexicalized aspect of lexicalized tree-adjoining grammars. This parser is part of the XTAG System, which includes a wide coverage grammar of English together with a morphological analyzer [XTAG-Group1995]. See also Section 6.

11. Summary

We have presented a class of grammars, Tree-Adjoining Grammars (TAG). Although motivated originally by some important linguistic considerations TAGs have turned out to be mathematically and computationally very interesting and have led to important mathematical results, which in turn have important linguistic implications. Thus TAGs represent an important class of grammars which demostrate the fascinating interplay between formal linguistic properties and mathematical/computational properties investigated in formal language theory and automata theory, including tree languages and tree automata.

References

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

[Abeillé1988] Anne Abeillé. 1988. Parsing french with tree adjoining grammar: some linguistic accounts. In *Proceedings of the* 12th *International Conference on Computational Linguistics (COLING'88)*, Budapest, August.

[Chomsky1981] N. Chomsky. 1981. *Lectures on Government and Binding*. Foris, Dordrecht.

[CI1994] Special issue of *Computational Intelligence*, November 1994, 10(4). Devoted to Tree-Adjoining Grammars.

[Earley1968] Jay C. Earley. 1968. *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

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

[Gross1984] Maurice Gross. 1984. Lexicon-grammar and the syntactic analysis of french. In *Proceedings of the* 10th *International Conference on Computational Linguistics (COLING'84)*, Stanford, 2-6 July.

[Joshi and Schabes1992] Aravind K. Joshi and Yves Schabes. 1992. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages*. Elsevier Science.

[Joshi and Srinivas1994] Aravind K. Joshi and B. Srinivas. 1994. Disambiguation of super parts of speech (Supertags): almost parsing. In *Proceedings of the* 17th *International Conference on Computational Linguistics (COLING'94)*, Kyoto, Japan, August.

[Joshi et al.1975] Aravind K. Joshi, L. S. Levy, and M. Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1).

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

[Joshi1985] Aravind K. Joshi. 1985. 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. Originally presented in a Workshop on Natural Language Parsing at Ohio State University, Columbus, Ohio, May 1983.

[Joshi1987] Aravind K. Joshi. 1987. An Introduction to Tree Adjoining Grammars. In A. Manaster-Ramer, editor, *Mathematics of Language*. John Benjamins, Amsterdam.

[Kaplan and Bresnan1983] R. Kaplan and J. Bresnan. 1983. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge MA.

[Karttunen1986] Lauri Karttunen. 1986. Radical lexicalism. Technical Report CSLI-86-68, CSLI, Stanford University. Also in *Alternative Conceptions of Phrase Structure*, University of Chicago Press, Baltin, M. and Kroch A., Chicago, 1989.

[Kasami1965] T. Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

[Kroch and Joshi1985] Anthony Kroch and Aravind K. Joshi. 1985. Linguistic relevance of tree adjoining grammars. Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania, April.

[Kroch1987] Anthony Kroch. 1987. Unbounded dependencies and subjacency in a tree adjoining grammar. In A. Manaster-Ramer, editor, *Mathematics of Language*. John Benjamins, Amsterdam.

[Lambek1958] Joachim Lambek. 1958. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

[Lang1988] Bernard Lang. 1988. The systematic constructions of Earley parsers: Application to the production of $O(n^6)$ Earley parsers for Tree Adjoining Grammars. Unpublished manuscript, December 30.

[Lavelli and Satta1991] Alberto Lavelli and Giorgio Satta. 1991. Bidirectional parsing of lexicalized tree adjoining grammars. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics (EACL'91)*, Berlin.

[Pollard and Sag1987] Carl Pollard and Ivan A. Sag. 1987. *Information-Based Syntax and Semantics. Vol 1: Fundamentals.* CSLI.

[Rambow et al.1995] Owen Rambow, K. Vijay-Shanker, and David Weir. 1995. D-tree grammars. In *Proceedings of the* 33rd *Annual Meeting of the Association for Computational Linguistics (ACL)*, Cambridge, MA, June, pp 151–158.

[Rambow1994] Owen Rambow. 1994. *Formal and Computational Aspects of Natural Language Syntax.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

[Resnick1994] P. Resnick. 1992. Probabilistic tree-adjoining grammars as framework for statistical natural language processing. In *Proceedings of the* 15th *International Conference on Computational Linguistics (COLING'92)*, Nantes, France, August.

[Satta1994] Giorgio Satta. 1994. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2), 173–192.

[Schabes and Joshi1988] Yves Schabes and Aravind K. Joshi. 1988. An Earley-type parsing algorithm for Tree Adjoining Grammars. In 26th *Meeting of the Association for Computational Linguistics (ACL'88)*, Buffalo, June.

[Schabes and Joshi1989] Yves Schabes and Aravind K. Joshi. 1989. The relevance of lexicalization to parsing. In *Proceedings of the International Workshop on Parsing Technologies*, Pittsburgh, August. Also appeared under the title *Parsing with Lexicalized Tree adjoining Grammar* in *Current Issues in Parsing Technologies*, MIT Press.

[Schabes and Shieber1994] Yves Schabes and Stuart Shieber. 1994. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20(1), 91–124.

[Schabes and Vijay-Shanker1990] Yves Schabes and K. Vijay-Shanker. 1990. Deterministic left to right parsing of Tree Adjoining Languages. In 28th *Meeting of the Association for Computational Linguistics (ACL'90)*, Pittsburgh.

[Schabes and Waters1995] Yves Schabes and Richard C. Waters. 1995. Tree-Insertion Grammars: A cubic time, parsable formalism that lexicalizes context-free grammars without changing the trees produced. *Computational Linguistics*, 21(4), pp 479–514.

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

[Schabes1990] Yves Schabes. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars.* Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, August. Available as technical report (MS-CIS-90-48, LINC LAB179) from the Department of Computer Science.

[Schabes1991] Yves Schabes. 1991. Left to right parsing of tree-adjoining grammars. *Computational Intelligence*, 10(4), 506–524.

[Schabes1992] Yves Schabes. 1992. Stochastic tree-adjoining grammars. In *Proceedings of the* 15^{th} *International Conference on Computational Linguistics (COLING'92)*, Nantes, France, August.

[Schimpf and Gallier1985] K. M. Schimpf and J. H. Gallier. 1985. Tree pushdown automata. *Journal of Computer and System Sciences*, 30:25–39.

[Shieber and Schabes1990] Stuart Shieber and Yves Schabes. 1990. Synchronous Tree-Adjoining Grammars. In *Proceedings of the* 13^{th} *International Conference on Computational Linguistics (COLING'90)*, Helsinki, Finland, August.

[Steedman1987] Mark Steedman. 1987. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5:403–439.

[Thatcher1971] J. W. Thatcher. 1971. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396.

[Vijay-Shanker and Joshi1985] K. Vijay-Shanker and Aravind K. Joshi. 1985. Some computational properties of Tree Adjoining Grammars. In 23^{rd} *Meeting of the Association for Computational Linguistics*, pages 82–93, Chicago, Illinois, July.

[Vijay-Shanker and Joshi1988] K. Vijay-Shanker and Aravind K. Joshi. 1988. Feature structure based tree adjoining grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, August.

[Vijay-Shanker and Weir1990] K. Vijay-Shanker and David J. Weir. 1990. Parsing constrained grammar formalisms. *Computational Linguistics*, 19(4), 591–636.

[Vijay-Shanker1987] K. Vijay-Shanker. 1987. *A Study of Tree Adjoining Grammars.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

[Vijay-Shanker1992] K. Vijay-Shanker. 1992. Using description of trees in a tree-adjoining grammar. *Computational Linguistics*, 18(4), pp 481–517.

[Weir1988] David J. Weir. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

[XTAG-Group1995] XTAG-Group. 1995. A lexicalized tree-adjoining grammar of English. Technical Report, Institute for Research in Cognitive Science (IRCS), University of Pennsylvania, 95-03.

[Younger1967] D. H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.