AN EARLEY-TYPE PARSING ALGORITHM FOR TREE ADJOINING GRAMMARS *

Yves Schabes and Aravind K. Joshi

Department of Computer and Information Science
University of Pennsylvania
Philadelphia PA 19104-6389 USA
schabes@linc.cis.upenn.edu joshi@cis.upenn.edu

ABSTRACT

We will describe an Earley-type parser for Tree Adjoining Grammars (TAGs). Although a CKY-type parser for TAGs has been developed earlier (Vijay-Shanker and Joshi, 1985), this is the first practical parser for TAGs because as is well known for CFGs, the average behavior of Earley-type parsers is superior to that of CKY-type parsers. The core of the algorithm is described. Then we discuss modifications of the parsing algorithm that can parse extensions of TAGs such as constraints on adjunction, substitution, and feature structures for TAGs. We show how with the use of substitution in TAGs the system is able to parse directly CFGs and TAGs. The system parses unification formalisms that have a CFG skeleton and also those with a TAG skeleton. Thus it also allows us to embed the essential aspects of PATR-II.

1 Introduction

Although formal properties of Tree Adjoining Grammars (TAGs) have been investigated (Vijay-Shanker, 1987)—for example, there is an $O(n^6)$-time CKY-like algorithm for TAGs (Vijay-Shanker and Joshi, 1985)—so far there has been no attempt to develop an Earley-type parser for TAGs. This paper presents an Earley parser for TAGs and discusses modifications to the parsing algorithm that make it possible to handle extensions of TAGs such as constraints on adjunction, substitution, and feature structure representation for TAGs.

TAGs were first introduced by Joshi, Levy and Takahashi (1975) and Joshi (1983). We describe very briefly the Tree Adjoining Grammar formalism. For more details we refer the reader to Joshi (1983), Kroch and Joshi (1985) or Vijay-Shanker (1987).

Definition 1 (Tree Adjoining Grammar) : A TAG is a 5-tuple $G = (V_N, V_T, S, I, A)$ where V_N is a finite set of non-terminal symbols, V_T is a finite set of terminals, S is a distinguished non-terminal, I is a finite set of trees called **initial trees** and A is a finite set of trees called **auxiliary trees**. The trees in $I \cup A$ are called **elementary trees**.

Initial trees (see left tree in Figure 1) are characterized as follows: internal nodes are labeled by non-terminals; leaf nodes are labeled by either terminal symbols or the empty string.

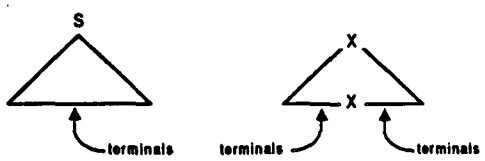

Figure 1: Schematic initial and auxiliary trees

Auxiliary trees (see right tree in Figure 1) are characterized as follows: internal nodes are labeled by non-terminals; leaf nodes are labeled by a terminal or by the empty string except for exactly one node (called the **foot node**) labeled by a non-terminal; furthermore the label of the foot node is the same as the label of the root node.

We now define a composition operation called **adjoining** or **adjunction** which builds a new tree from an auxiliary tree β and a tree α (α is any tree,

*This work is partially supported by ARO grant DAA29-84-9-007, DARPA grant N0014-85-K0018, NSF grants MCS-82-191169 and DCR-84-10413. The authors would like to express their gratitude to Vijay-Shanker for his helpful comments relating to the core of the algorithm, Richard Billington and Andrew Chalnick for their graphical TAG editor which we integrated in our system and for their programming advice. Thanks are also due to Anne Abeillé and Ellen Hays.

initial, auxiliary or tree derived by adjunction). The resulting tree is called a **derived tree**. Let α be a tree containing a node n labeled by X and let β be an auxiliary tree whose root node is also labeled by X. Then the adjunction of β to α at node n will be the tree γ shown in Figure 2. The resulting tree, γ, 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 n and its root node is identified with n.
- The sub-tree t is attached to the foot node of β and the root node n of t is identified with the foot node of β.

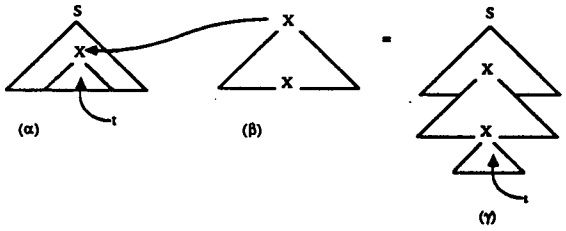

Figure 2: The mechanism of adjunction

Then define the **tree set** of a TAG G, $T(G)$ to be the set of all derived trees starting from initial trees in I. Furthermore, the **string language** generated by a TAG, $L(G)$, is defined to be the set of all terminal strings of the trees in $T(G)$.

TAGs factor recursion and dependencies by extending the domain of locality. They offer novel ways to encode the syntax of natural language grammars as discussed in Kroch and Joshi (1985) and Abeillé (1988).

In 1985, Vijay-Shanker and Joshi introduced a CKY-like algorithm for TAGs. They therefore established $O(n^6)$ 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. First the algorithm assumes that elementary trees are binary branching and that there are no empty categories on the frontiers of the elementary trees. Second, since it works on nodes that have been isolated from the tree they belong to, it isolates them from their domain of locality. However all important linguistic and computational properties of TAGs follow from this extended domain of locality. And 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.

Since the average time complexity of Earley's parser 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. Earley's algorithm for CFGs (Earley, 1970, Aho and Ullman, 1973) is a bottom-up parser which uses top-down information. It manipulates states of the form $A \rightarrow \alpha.\beta[i]$ while using three processors: the predictor, the completor and the scanner. The algorithm for CFGs runs in $O(|G|^2 n^3)$ time and in $O(|G|n^2)$ space in all cases, and parses unambiguous grammars in $O(n^2)$ time (n being the length of the input, $|G|$ the size of the grammar).

Given a context-free grammar in any form and an input string $a_1 \cdots a_n$, Earley's parser for CFGs maintains the following invariant:
The state $A \rightarrow \alpha.\beta[i]$ is in states set S_k iff
$S \stackrel{*}{\Rightarrow} \delta A \gamma$, $\delta \stackrel{*}{\Rightarrow} a_1 \cdots a_i$ and $\alpha \stackrel{*}{\Rightarrow} a_{i+1} \cdots a_k$
The correctness of the algorithm is a corollary of this invariant.

Finding a Earley-type parser for TAGs was a difficult task because it was not clear how to parse TAGs bottom up using top-down information while scanning the input string from left to right. In order to construct an Earley-type parser for TAGs, we will extend the notions of dotted rules and states to trees. Anticipating the proof of correctness and soundness of our algorithm, we will state an invariant similar to Earley's original invariant. Then we present the algorithm and its main extensions.

2 Dotted symbols, dotted trees, tree traversal

The full algorithm is explained in the next section. This section introduces preliminary concepts that will be used by the algorithm. We first show how dotted rules can be extended to trees. Then we introduce a tree traversal that the algorithm will mimic in order to scan the input from left to right.

We define a **dotted symbol** as a symbol associated with a dot *above* or *below* and either *to the left* or *to the right* of it. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below): ${}^{la}_{lb}A^{ra}_{rb}$.

Then we define a **dotted tree** as a tree with exactly one dotted symbol.

Given a dotted tree with the dot above and to the left of the root, we define a tree traversal of a dotted tree as follows (see Figure 3):

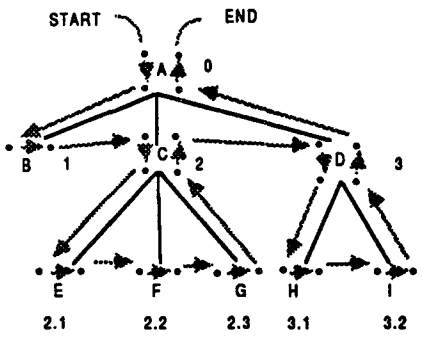

Figure 3: Example of a tree traversal

• if the dot is at position *la* of an internal node, 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 move the dot to the right sibling at position *la*.
- if the node does not have a right sibling, then 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.

3 The algorithm

We define an appropriate data structure for the algorithm. We explain how to interpret the structures that the parser produces. Then we describe the algorithm itself.

3.1 Data structures

The algorithm uses two basic data structures: state and states set.

A states set S is defined as a set of states. The states sets will be indexed by an integer: S_i with $i \in N$. The presence of any state in states set i will mean that the input string $a_1 ... a_i$ has been recognized.

Any tree α will be considered as 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 symbol at address x in the tree α.

Definition 2 A state s is defined as a 10-tuple, $[\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*, b_l^*]$ where:
• α: is the name of the dotted tree.
• *dot*: is the address of the dot in the tree α.
• *side*: is the side of the symbol the dot is on;
$side \in \{left, right\}$.
• *pos*: is the position of the dot;
$pos \in \{above, below\}$.
• *star*: is an address in α. The corresponding node in α is called the starred node.
• l (left), f_l (foot left), f_r (foot right), t_l^* (top left of starred node), b_l^* (bottom left of starred node) are indices of positions in the input string ranging over $[0, n]$, n being the length of the input string. They will be explained further below.

3.2 Invariant of the algorithm

The states s in a states set S_i have a common property. The following section describes this invariant in order to give an intuitive interpretation of what the algorithm does. This invariant is similar to Earley's invariant.

Before explaining the main characterization of the algorithm, we need to define the set of nodes on which an adjunction is allowed for a given state.

Definition 3 The set of nodes $\mathcal{P}(s)$ on which an adjunction is possible for a given state $s = [\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*, b_l^*]$, is defined as the union of the following sets of nodes in α:
• the set of nodes that have been traversed on the left and right sides, i.e., the four positions of the dot have been traversed;
• the set of nodes on the path from the root node to the starred node, root node and starred node included. Note that if there is no star this set is empty.

Definition 4 (Left part of a dotted tree) The left part of a dotted tree is the union of the set of nodes in the tree that have been traversed on the left and right sides and the set of nodes that have been traversed on the left side only.

We will first give an intuitive interpretation of the ten components of a state, and then give the necessary and sufficient conditions for membership of a state in a states set.

We interpret informally a state $s = [\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*, b_l^*]$ in the following way (see Figure 4):

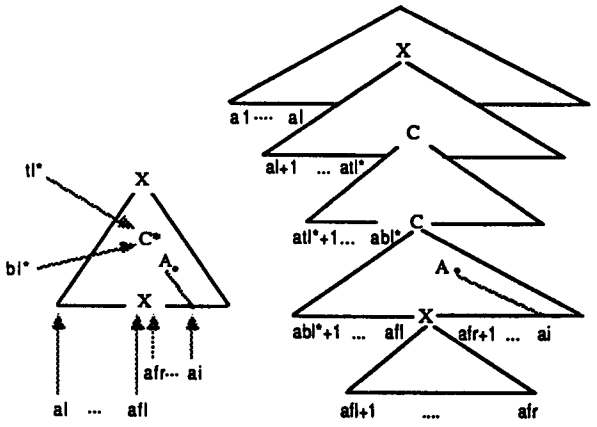

Figure 4: Meaning of $s \in S_i$

- l is an index in the input string indicating where the tree derived from α begins.
- f_l is an index in the input string corresponding to the point just before the foot node (if any) in the tree derived from α.
- f_r is an index in the input string corresponding to the point just after the foot node (if any) in the tree derived from α.The pair f_l and f_r will mean that the foot node subsumes the string $a_{f_l+1} \ldots a_{f_r}$.
- *star*: is the address in α of the deepest node that subsumes the dot on which an adjunction has been partially recognized. If there is no adjunction in the tree α along the path from the root to the dotted node, *star* is unbound.
- t_l^* is an index in the input string corresponding to the point in the tree where the adjunction on the starred node was made. If *star* is unbound, then t_l^* is also unbound.
- b_l^* is an index in the input string corresponding to the point in the tree just before the foot node of the tree adjoined at the starred node. The pair t_l^* and b_l^* will mean that the string as far as the foot node of the auxiliary tree adjoined at the starred node matches the substring $a_{t_l^*+1} \ldots a_{b_l^*}$ of the input string. If *star* is unbound, then b_l^* is also unbound.
- $s \in S_i$ means that the recognized part of the dotted tree α, which is the left part of it, is consistent with the input string from a_1 to a_l and from a_l to a_{f_l} and from a_{f_r} to a_i, or from a_1 to a_l and from a_l to a_i when the foot node is not in the recognized part of the tree.

We are now ready to characterize the membership of s in S_i:

Invariant 1
A state $s = [\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*, b_l^*]$ is in S_i if and only if there is a derived tree from an initial tree such that (see Figure 4):
1. The tree α is part of the derivation.
2. The tree derived from α in the derivation tree, $\overline{\alpha}$, has adjunctions only on nodes in $\mathcal{P}(s)$.
3. The part of the tree to the left of the dot in the tree derived spans the string $a_1 \ldots a_i$.
4. The tree derived from α, $\overline{\alpha}$, has a yield that starts just after a_l, ends at a_{f_l} before the foot node (if a_{f_l} is defined), and starts after the foot node just after a_{f_r} (if a_{f_r} is defined).
5. If there are adjunctions on the path from the dotted node to the root of α, then *star* is the address of the deepest adjunction on that path and the auxiliary tree adjoined at that node *star* has a yield that starts just after $a_{t_l^*}$ and stops at its foot node at $a_{b_l^*}$.

The proof of this invariant has as corollaries the soundness, completeness, and therefore the correctness of the algorithm.

3.3 The recognizer

The Earley-type recognizer for TAGs follows:

```
Let G be a TAG.
Let a_1...a_n be the input string.

program recognizer
begin
  S_0 = { [α,0,left,above,0,-,-,-,-,-]
          |α is an initial tree }
  For i := 0 to n do
  begin
    Process the states of S_i, performing one of
    the following seven operations on each state
    s = [α,dot,side,pos,l,f_l,f_r,star,t_l*,b_l*]
    until no more states can be added:
      1. Scanner
      2. Move dot down
      3. Move dot up
      4. Left Predictor
      5. Left Completor
      6. Right Predictor
      7. Right Completor
    If S_{i+1} is empty and i < n, return rejection.
  end
  If there is in S_n a state
  s = [α,0,right,above,0,-,-,-,-,-]
  such that α is an initial tree
  then return acceptance.
end.
```

The algorithm is a general recognizer for TAGs. Unlike the CKY algorithm, it requires no condition on the grammar: the trees can be binary or not, the elementary (initial or auxiliary) trees can have the empty string as frontier. It is an off-line algorithm: it needs to know the length n of the input string. However we will see later that it can very easily be modified to an on-line algorithm by the use of an end-marker in the input string.

We now describe one by one the seven processes. The current states set is presumed to be S_i and the state to be processed is

$$s = [\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*].$$

Only one of the seven processes can be applied to a given state. The side, the position, and the address of the dot determine the unique process that can be applied to the given state.

Definition 5 (*Adjunct*$(\alpha, address)$) Given a TAG G, define *Adjunct*$(\alpha, address)$ as the set of auxiliary trees that can be adjoined in the elementary tree α at the node n which has the given address. In a TAG without any constraints on adjunction, if n is a non-terminal node, this set consists of all auxiliary trees that are rooted by a node with same label as the label of n.

3.3.1 Scanner

The scanner scans the input string. Suppose that the dot is to the left of and above a terminal symbol (see Figure 5). Then if the terminal symbol matches the next input token, the program should record that a new token has been recognized and try to recognize the rest of the tree.

Therefore the scanner applies to
$s = [\alpha, dot, left, above, l, f_l, f_r, star, t_l^*, b_l^*]$
such that $\alpha(dot)$ **is a terminal symbol and** $\alpha(dot) = a_{i+1}$ **or** $\alpha(dot)$ **is the empty symbol** ϵ.

- **Case 1:** $\alpha(dot) = a_{i+1}$
  The scanner adds
  $[\alpha, dot, right, above, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_{i+1}.
- **Case 2:** $\alpha(dot) = \epsilon$
  The scanner adds
  $[\alpha, dot, right, above, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_i.

3.3.2 Move Dot Down

Move dot down (See Figure 6), moves the dot down, from position lb of the dotted node to position la of its leftmost child.

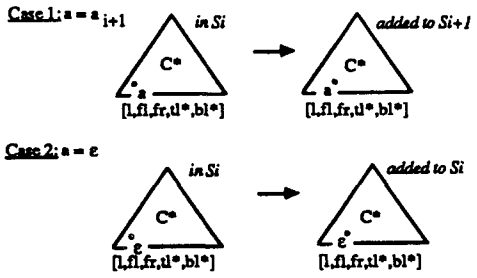

Figure 5: Scanner

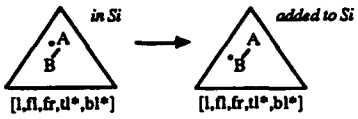

Figure 6: Move dot down

It therefore applies to
$s = [\alpha, dot, left, below, l, f_l, f_r, star, t_l^*, b_l^*]$
such that the node where the dot is has a leftmost child at address u.
It adds $[\alpha, u, left, above, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_i.

3.3.3 Move Dot Up

Move dot up (See Figure 7), moves the dot "up", from position ra of the dotted node to position la of its right sibling if it has a right sibling, otherwise to position rb of its parent.

It therefore applies to
$s = [\alpha, dot, right, above, l, f_l, f_r, star, t_l^*, b_l^*]$
such that the node on which the dot is has a parent node.

- **Case 1: the node where the dot is has a right sibling at address** r.
  It adds $[\alpha, r, left, above, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_i.
- **Case 2: the node where the dot is is the rightmost child of the parent node** p.
  It adds
  $[\alpha, p, right, below, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_i.

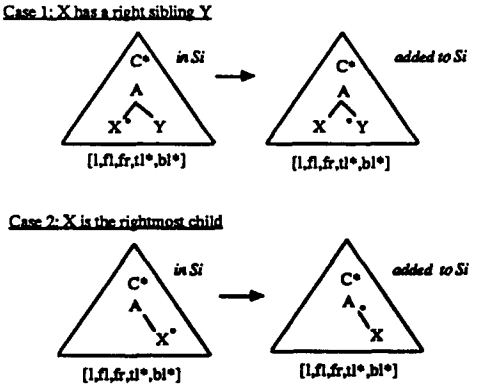

Figure 7: Move dot up

3.3.4 Left Predictor

Suppose that there is a dot to the left of and above a non-terminal symbol A (see Figure 8). Then the algorithm takes two paths in parallel: it makes a prediction of adjunction on the node labeled by A and tries to recognize the adjunction (step1) and it also considers the case where no adjunction has been done (step2). These operations are performed by the **Left Predictor**.

It applies to
$s = [\alpha, dot, left, above, l, f_l, f_r, star, t_l^*, b_l^*]$
such that $\alpha(dot)$ is a non-terminal.

- Step 1. It adds the states
  $\{[\beta, 0, left, above, i, -, -, -, -, -]$
  $|\beta \in Adjunct(\alpha, dot)\ \}$ to S_i.
- Step 2.
  - Case 1: the dot is not on the foot node.
    It adds the state
    $[\alpha, dot, left, below, l, f_l, f_r, star, t_l^*, b_l^*]$
    to S_i.
  - Case 2: the dot is on the foot node. Necessarily, since the foot node has not been already traversed, f_l and f_r are unspecified.
    It adds the state
    $[\alpha, dot, left, below, l, i, -, star, t_l^*, b_l^*]$ to S_i.

3.3.5 Left Completor

Suppose that the auxiliary that we left-predicted has been recognized as far as its foot (see Figure 9). Then the algorithm should try to recognize

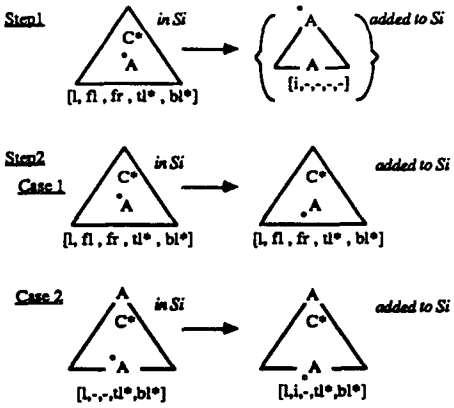

Figure 8: Left Predictor

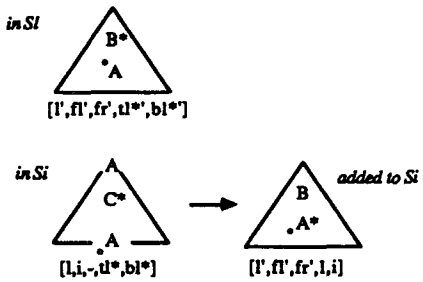

Figure 9: Left completor

what was pushed under the foot node. (A star in the original tree will signal that an adjunction has been made and half recognized.) This operation is performed by the **Left Completor**.

It applies to
$s = [\alpha, dot, left, below, l, i, -, star, t_l^*, b_l^*]$
such that the dot is on the foot node.
For all
$s = [\beta, dot', left, above, l', f_l', f_r', star', t_l^{*'}, b_l^{*'}]$ in S_l such that $\alpha \in Adjunct(\beta, dot')$

- Case 1: dot' is on the foot node of β. Then necessary, f_l' and f_r' are unbound.
  It adds the state
  $[\beta, dot', left, below, l', i, -, dot', l, i]$ to S_i.
- Case 2: dot' is not on the foot node of β.
  It adds the state
  $[\beta, dot', left, below, l', f_l', f_r', dot', l, i]$ to S_i.

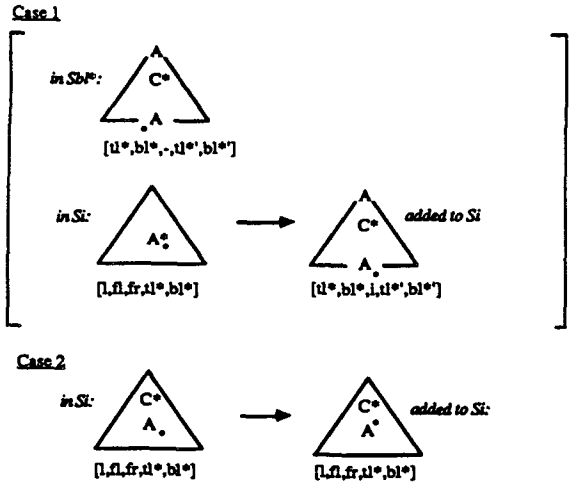

Figure 10: Right Predictor

3.3.6 Right Predictor

Suppose that there is a dot to the right of and below a node A (see Figure 10). If there has been an adjunction made on A (case 1), the program should try to recognize the right part of the auxiliary tree adjoined at A. However if there was no adjunction on A (case 2), then the dot should be moved up. Note that the star will tell us if an adjunction has been made or not. These operations are performed by the **Right predictor**.

The right predictor applies to
$s = [\alpha, dot, right, below, l, f_l, f_r, star, t_l^*, b_l^*]$

- **Case 1:** $dot = star$
  For all states
  $s = [\beta, dot', left, below, t_l^*, b_l^*, -, star', t_l^{*'}, b_l^{*'}]$
  in $S_{b_l^*}$ **such that** $\beta \in Adjunct(\alpha, dot)$,
  it adds the state
  $[\beta, dot', right, below, t_l^*, b_l^*, i, star', t_l^{*'}, b_l^{*'}]$ **to** S_i.
- **Case 2:** $dot \neq star$
  It adds the state
  $[\alpha, dot, right, above, l, f_l, f_r, star, t_l^*, b_l^*]$ **to** S_i.

3.3.7 Right Completor

Suppose that the dot is to the right of and above the root of an auxiliary tree (see Figure 11). Then the adjunction has been totally recognized and the program should try to recognize the rest of the tree in which the auxiliary tree has been adjoined. This operation is performed by the **Right Completor**.

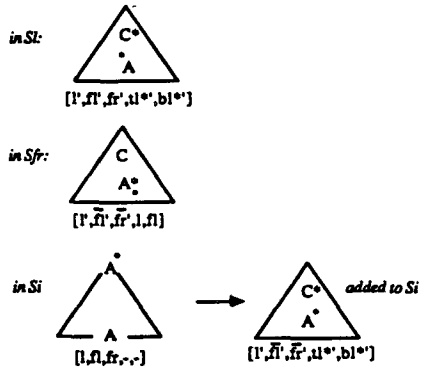

Figure 11: Right Completor

It applies to
$s = [\alpha, 0, right, above, l, f_l, f_r, -, -, -]$
For all states
$s_l = [\beta, dot', left, above, l', f_l', f_r', star', t_l^{*'}, b_l^{*'}]$
in S_l
and for all states
$[\beta, dot', right, below, l', \overline{f_l'}, \overline{f_r'}, dot', l, f_l]$ **in** S_{f_r}
such that $\alpha \in Adjunct(\beta, dot')$
It adds
$[\beta, dot', right, above, l', \overline{f_l'}, \overline{f_r'}, star', t_l^{*'}, b_l^{*'}]$ **to** S_i.
Where $\overline{f} = f$, **if** f **is bound in state** s_l, **and** $\overline{f}$ **can have any value, if** f **is unbound in state** s_l.

3.4 Handling constraints on adjunction

In a TAG, one can, for each node of an elementary tree, specify one of the following three constraints on adjunction (Joshi, 1987):

- **Null adjunction** (NA): disallow any adjunction on the given node.
- **Obligatory adjunction** (OA): an auxiliary tree must be adjoined on the given node.
- **Selective adjunction** ($SA(T)$): a set T of auxiliary trees that can be adjoined on the given node is specified.

The algorithm can be very easily modified to handle those constraints. First, the function $Adjunct(\alpha, address)$ must be modified as follows:

- $Adjunct(\alpha, address) = \emptyset$, if there is NA on the node.
- $Adjunct(\alpha, address)$ as previously defined, if there is OA on the node.
- $Adjunct(\alpha, address) = T$, if there is $SA(T)$ on the node.

Second, step 2 of the left predictor must be done

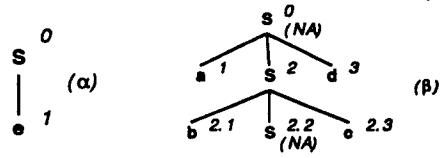

Figure 12: $L = \{a^n b^n e c^n d^n | n \geq 0\}$

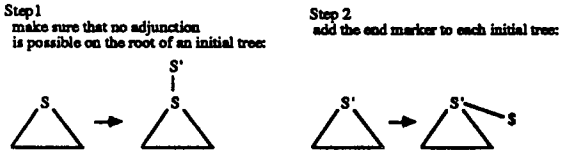

Figure 13: Use of end marker in TAG

only if there is no obligatory adjunction on the node at address *dot* in the tree α.

3.5 An example

We give one example that illustrates how the recognizer works. The grammar used for the example generates the language $L = \{a^n b^n e c^n d^n | n \geq 0\}$. The input string given to the recognizer is: *aabbeccdd*. The grammar is shown in Figure 12. The states sets are shown in Figure 14. Next to each state we have printed in parentheses the name of the processor that was applied to the state. The input is recognized since $[\alpha, 0, right, above, 0, -, -, -, -, -]$ is in states set S_9.

3.6 Remarks

Use of move dot up and move dot down
Move dot down and move dot up can be eliminated in the algorithm by merging the original dot and the position it is moved to. However for explanatory purposes we chose to use these two processors in this paper.

Off-line vs on-line
The algorithm given is an off-line recognizer. It can be very easily modified to work on line by adding an end marker to all initial trees in the grammar (see Figure 13).

Extracting a parse
The algorithm that we describe in section 3.3 is a recognizer. However, if we include pointers from a state to the other states which caused it to be placed in the states set, the recognizer can be modified to produce all parses of the input string.

3.7 Correctness

The correctness of the parser has been proven and is fully reported in Schabes and Joshi (1988). It consists of the proof of the invariant given in section 3.2. Our proof is similar in its concept to the proof of the correctness of Earley's parser given in Aho and Ullman 1973. The "only if" part of the invariant is proved by induction on the number of states that have been added so far to all states sets. The "if" part is proved by induction on a defined rank of a state. The soundness (the algorithm recognizes only valid strings) and the completeness (if a string is valid, then the algorithm will recognize it) are corollaries of this invariant.

3.8 Implementation

The parser has been implemented on Symbolics Lisp machines in Flavors. More details of the actual implementation can be found in Schabes and Joshi (1988). The current implementation has an $O(|G|^2 n^9)$ worst case time complexity and $O(|G|n^6)$ worst case space complexity. We have not as yet been able to reduce the worst case time complexity to $O(|G|^2 n^6)$. We are currently attempting to reduce this bound. However, the main purpose of constructing an Earley-type parser is to improve the average complexity, which is crucial in practice.

4 Extensions

We describe how substitution is defined in a TAG. We discuss the consequences of introducing substitution in TAGs. Then we show how substitution can be parsed. We extend the parser to deal with feature structures for TAGs. Finally the relationship with PATR-II is discussed.

4.1 Introducing substitution in TAGs

TAGs use adjunction as their basic composition operation. It is well known that Tree Adjoining Languages (TALs) are mildly context-sensitive. TALs properly contain context-free languages. 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

S_0	$[\alpha, 0, left, above, 0, -, -, -, -, -]$ (left predictor) $[\alpha, 0, left, below, 0, -, -, -, -, -]$ (move dot down) $[\beta, 1, left, above, 0, -, -, -, -, -]$ (scanner)	$[\beta, 0, left, above, 0, -, -, -, -, -]$ (left predictor) $[\beta, 0, left, below, 0, -, -, -, -, -]$ (move dot down) $[\alpha, 1, left, above, 0, -, -, -, -, -]$ (scanner)
S_1	$[\beta, 1, right, above, 0, -, -, -, -, -]$ (move dot up) $[\beta, 2, left, below, 0, -, -, -, -, -]$ (move dot down) $[\beta, 2.1, left, above, 0, -, -, -, -, -]$ (scanner) $[\beta, 1, left, above, 1, -, -, -, -, -]$ (scanner)	$[\beta, 2, left, above, 0, -, -, -, -, -]$ (left predictor) $[\beta, 0, left, above, 1, -, -, -, -, -]$ (left predictor) $[\beta, 0, left, below, 1, -, -, -, -, -]$ (move dot down)
S_2	$[\beta, 0, left, above, 2, -, -, -, -, -]$ (left predictor) $[\beta, 2, left, below, 1, -, -, -, -, -]$ (move dot down) $[\beta, 0, left, below, 2, -, -, -, -, -]$ (move dot down) $[\beta, 1, right, above, 1, -, -, -, -, -]$ (move dot up)	$[\beta, 2.1, left, above, 1, -, -, -, -, -]$ (scanner) $[\beta, 1, left, above, 2, -, -, -, -, -]$ (scanner) $[\beta, 2, left, above, 1, -, -, -, -, -]$ (left predictor)
S_3	$[\beta, 2.2, left, below, 1, 3, -, -, -, -]$ (left completor) $[\beta, 2.1, right, above, 1, -, -, -, -, -]$ (move dot up) $[\beta, 2.1, left, above, 0, -, -, 2, 1, 3]$ (scanner)	$[\beta, 2, left, below, 0, -, -, 2, 1, 3]$ (move dot down) $[\beta, 2.2, left, above, 1, -, -, -, -, -]$ (left predictor)
S_4	$[\alpha, 1, left, above, 0, -, -, 0, 0, 4]$ (scanner) $[\beta, 2.2, left, above, 0, -, -, 2, 1, 3]$ (left predictor) $[\beta, 2.2, left, below, 0, 4, -, 2, 1, 3]$ (left completor)	$[\alpha, 0, left, below, 0, -, -, 0, 0, 4]$ (move dot down) $[\beta, 2.1, right, above, 0, -, -, 2, 1, 3]$ (move dot up)
S_5	$[\beta, 2.3, left, above, 0, 4, 5, 2, 1, 3]$ (scanner) $[\beta, 2.2, right, above, 0, 4, 5, 2, 1, 3]$ (move dot up) $[\alpha, 1, right, above, 0, -, -, 0, 0, 4]$ (move dot up)	$[\beta, 2.2, right, below, 0, 4, 5, 2, 1, 3]$ (right predictor, case 2) $[\alpha, 0, right, below, 0, -, -, 0, 0, 4]$ (right predictor, case 1)
S_6	$[\beta, 2.2, right, above, 1, 3, 6, -, -, -]$ (move dot up) $[\beta, 2.3, left, above, 1, 3, 6, -, -, -]$ (scanner) $[\beta, 2.2, right, below, 1, 3, 6, -, -, -]$ (right predictor, case 2)	$[\beta, 2.3, right, above, 0, 4, 5, 2, 1, 3]$ (move dot up) $[\beta, 2, right, below, 0, 4, 5, 2, 1, 3]$ (right predictor, case 1)
S_7	$[\beta, 2, right, below, 1, 3, 6, -, -, -]$ (right predictor, case 2) $[\beta, 3, left, above, 1, 3, 6, -, -, -]$ (scanner)	$[\beta, 2, right, above, 1, 3, 6, -, -, -]$ (move dot up) $[\beta, 2.3, right, above, 1, 3, 6, -, -, -]$ (move dot up)
S_8	$[\beta, 0, right, below, 1, 3, 6, -, -, -]$ (right predictor, case 2) $[\beta, 3, left, above, 0, 4, 5, -, -, -]$ (scanner) $[\beta, 2, right, above, 0, 4, 5, -, -, -]$ (move dot up)	$[\beta, 0, right, above, 1, 3, 6, -, -, -]$ (right completor) $[\beta, 3, right, above, 1, 3, 6, -, -, -]$ (move dot up)
S_9	$[\beta, 0, right, below, 0, 4, 5, -, -, -]$ (right predictor, case 2) $[\beta, 0, right, above, 0, 4, 5, -, -, -]$ (right completor)	$[\alpha, 0, right, above, 0, -, -, -, -, -]$ (end test) $[\beta, 3, right, above, 0, 4, 5, -, -, -]$ (move dot up)

Figure 14: States sets for the input *aabbeccdd*

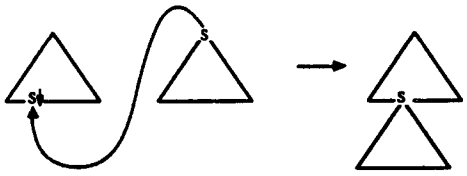

Figure 15: Mechanism of substitution

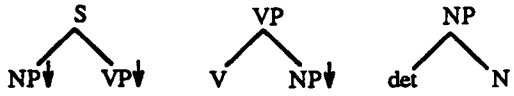

Figure 16: Writing a CFG in TAG

context free grammar since this encoding uses adjunction.

Substitution is the basic operation used in CFG. A CFG can be viewed as a tree rewriting system. It uses substitution as basic operation and it consists of a set of one-level trees. Substitution is a less powerful operation than adjunction.

However, recent linguistic work in TAG grammar development (Abeillé, 1988) showed the need for substitution in TAGs as an additional operation for obtaining appropriate structural descriptions in certain cases such as verbs taking two sentential arguments (e.g. "John equates solving this problem with doing the impossible") or compound categories. It has also been shown to be useful for lexical insertion (Schabes, Abeillé and Joshi, 1988). It should be emphasized that the introduction of substitution in TAGs does not increase their generative capacity. Neither is it a step back from the original idea of TAGs.

Definition 6 (Substitution in TAG) We define substitution in TAGs 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. Substitution is illustrated in Figure 15.

We conventionally mark substitution nodes by a down arrow ($\downarrow$).

As a consequence, we can now encode directly a CFG in a TAG with substitution. The resulting TAG has only one-level initial trees and uses only substitution. An example is shown in Figure 16.

4.2 Parsing substitution

The parser can be extended very easily to handle substitution. We use Earley's original predictor and completor to handle substitution.

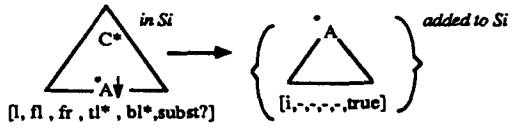

Figure 17: Substitution Predictor

The left predictor is restricted to apply to nodes to which adjunction can be applied.

A flag *subst*? is added to the states. When set, it indicates that the tree (initial) has been predicted for substitution. We use the index l (as in Earley's original parser) to know where it has been predicted for substitution. When the initial tree that has been predicted for substitution has been totally recognized, we complete the state as Earley's original parser does.

A state s is now an 11–tuple
$[\alpha, dot, side, pos, l, f_l, f_r, star, t_l^*, b_l^*, subst?]$
where *subst*? is a boolean that indicates whether the tree has been predicted for substitution. The other components have not been changed.

We add two more processors to the parser.

Substitution Predictor

Suppose that there is a dot to the left of and above a non-terminal symbol on the frontier A that is marked for substitution (see Figure 17). Then the algorithm predicts for substitution all initial trees rooted by A and tries to recognize the initial tree. This operation is performed by the **substitution predictor.**

It applies to
$s = [\alpha, dot, left, above, l, f_l, f_r, star, t_l^*, b_l^*, subst?]$
such that $\alpha(dot)$ **is a non-terminal on the frontier of** α **which is marked for substitution:**
It adds the states
$\{[\beta, 0, left, above, i, -, -, -, -, -, true]$
$|\beta$ **is an initial tree s.t.** $\beta(0) = \alpha(dot)\}$
to S_i.

Substitution Completor

Suppose that the initial tree that we predicted for substitution has been recognized (see Figure 18). Then the algorithm should try to recognize the rest of the tree in which we predicted a substitution. This operation is performed by the **substitution completor.**

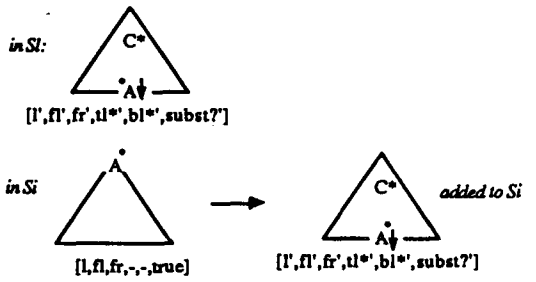

Figure 18: Substitution completor

It applies to
$s = [\alpha, 0, right, above, l, -, -, -, -, -, true]$
For all states $s =$
$[\beta, dot', left, above, l', f_l', f_r', star', t_l^{*'}, b_l^{*'}, subst?']$
in S_l **s.t.** $\beta(dot')$ **is marked for substitution and** $\beta(dot) = \alpha(0)$.
It adds the following state to S_i**:**
$[\beta, dot', right, above, l', f_l', f_r', star', t_l^{*'}, b_l^{*'}, subst?']$.

Complexity

The introduction of the substitution predictor and the substitution completor does not increase the complexity of the overall TAG parser.

If we encode a CFG with substitution in TAG, the parser behaves in $O(|G|^2 n^3)$ worst case time and $O(|G|n^2)$ worst case space like Earley's original parser. This comes from the fact that when there are no auxiliary trees and when only substitution is used, the indices f_l, f_r, t_l^*, b_l^* of a state will never be set. The algorithm will use only the substitution predictor and the substitution completor. Thus, it behaves exactly like Earley's original parser on CFGs.

4.3 Parsing feature structures for TAGs

The definition of feature structures for TAGs and their semantics was proposed by Vijay-Shanker (1987) and Vijay-Shanker and Joshi (1988). We first explain briefly how they work in TAGs and show how we have implemented them. We introduce in a TAG framework a language similar to PATR-II which was investigated by Shieber (Shieber, 1984 and 1986). We then show how one can embed the essential aspects of PATR-II in this system.

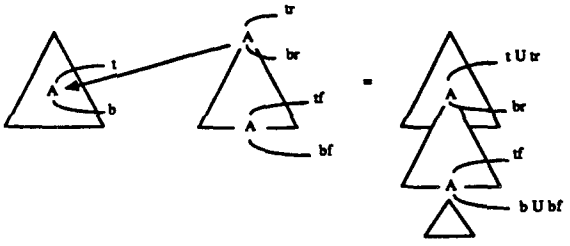

Figure 19: Updating of features

Feature structures in TAGs

As defined by Vijay-Shanker (1987) and Vijay-Shanker and Joshi(1988), to each adjunction node in an elementary tree two feature structures are attached: a top and a bottom feature structure. The top feature corresponds to a top view in the tree from the node. The bottom feature corresponds to the bottom view. When the derivation is completed, the top and bottom features of all nodes are unified. If the top and bottom features of a node do not unify, then a tree must be adjoined at that node.

This definition can be trivially extended to substitution nodes. To each substitution node we attach two identical feature structures (top and bottom).

The updating of features in case of adjunction is shown in Figure 19.

Unification equations

As in PATR-II, we express with unification equations dependencies between DAGs in an elementary tree. The system therefore consists of a TAG and a set of unification equations on the DAGs associated with nodes in elementary trees.

An example of the use of unification equations in TAGs is given in Figure 20. Note that the top and bottom features of node S in α can not be unified. This forces an adjunction to be performed on S. Thus, the following sentence is not accepted:

`*to go to the movies.`

The auxiliary tree $\beta 1$ can be adjoined at S in α:

`John wants to go to the movies.`

But since the bottom feature of S has tensed value $-$ in α and since the bottom feature of S has tensed value $+$ in β_2, β_1 can not be adjoined at node S in α:

`*Bob thinks to go to the movies.`

But β_2 can be adjoined in β_1, which itself can be adjoined in α:

`Bob thinks John wants to go to the movies.`

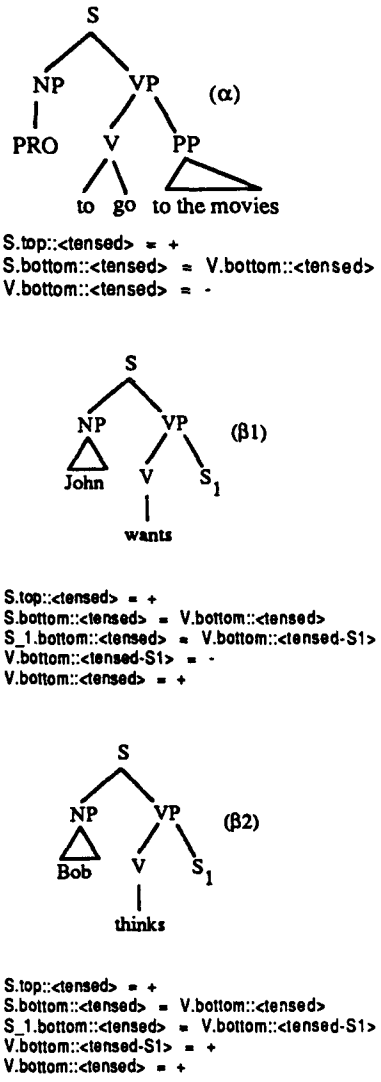

Figure 20: Example of unification equations

We refer the reader to Abeillé (1988) and to Schabes, Abeillé and Joshi (1988) for further explanation of the use of unification equations and substitution in TAGs.

Parsing and the relationship with PATR-II

By adding to each state the set of DAGs corresponding to the top and bottom features of each node, and by making sure that the unification equations are satisfied, we have extended the parser to parse TAGs with feature structures.

Since we introduced substitution and since we are able to encode a CFG directly, the system has the main functionalities of PATR-II. The system parses unification formalisms that have a CFG skeleton and a TAG skeleton.

5 Conclusion

We described an Earley-type parser for TAGs. We extended it to deal with substitution and feature structures for TAGs. By doing this, we have built a system that parses unification formalisms that have a CFG skeleton and also those that have a TAG skeleton. The system is being used for Tree Adjoining Grammar development (Abeillé, 1988). This work has led us to a new general parsing strategy (Schabes, Abeillé and Joshi, 1988) which allows us to construct a two-stage parser. In the first stage a subset of the elementary trees is extracted and in the second stage the sentence is parsed with respect to this subset. This strategy significantly improves performance, especially as the grammar size increases.

References

Abeillé, Anne, 1988. A Computational Grammar for French in TAG. In *Proceeding of the* 12^{th} *International Conference on Computational Linguistics.*

Aho, A. V. and Ullman, J. D., 1973. *Theory of Parsing, Translation and Compiling. Vol 1: Parsing.* Prentice-Hall, Englewood Cliffs, NJ.

Earley, J., 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13(2):94–102.

Joshi, Aravind K., 1985. How Much Context-Sensitivity is Necessary for Characterizing Structural Descriptions — Tree Adjoining Grammars. In Dowty, D.; Karttunen, L.; and Zwicky, A. (editors), *Natural Language Processing — Theoretical, Computational and Psychological Perspectives.* Cambridge University Press, New York. Originally presented in 1983.

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

Joshi, A. K.; Levy, L. S.; and Takahashi, M., 1975. Tree Adjunct Grammars. *J. Comput. Syst. Sci.* 10(1).

Kroch, A. and Joshi, A. K., 1985. *Linguistic Relevance of Tree Adjoining Grammars.* Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania.

Schabes, Yves and Joshi, Aravind K., 1988. *An Earley-type Parser for Tree Adjoining Grammars.* Technical Report, Department of Computer and Information Science, University of Pennsylvania.

Schabes, Yves; Abeillé, Anne; and Joshi, Aravind K., 1988. New Parsing Strategies for Tree Adjoining Grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics.*

Shieber, Stuart M., 1984. The Design of a Computer Language for Linguistic Information. In 22^{rd} *Meeting of the Association for Computational Linguistics*, pages 362–366.

Shieber, Stuart M., 1986. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, Stanford, CA.

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

Vijay-Shanker, K. and Joshi, A. K., 1985. Some Computational Properties of Tree Adjoining Grammars. In 23^{rd} *Meeting of the Association for Computational Linguistics*, pages 82–93.

Vijay-Shanker, K. and Joshi, A.K., 1988. Feature Structure Based Tree Adjoining Grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics.*