The Valid Prefix Property and Left to Right Parsing of Tree-Adjoining Grammar*

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

Abstract

The valid prefix property (VPP), the capability of a left to right parser to detect errors as soon as possible, often goes unnoticed in parsing CFGs. Earley's parser for CFGs (Earley, 1968; Earley, 1970) maintains the valid prefix property and obtains an $O(n^3)$-time worst case complexity, as good as parsers that do not maintain such as the CKY parser (Younger, 1967; Kasami, 1965). Contrary to CFGs, maintaining the valid prefix property for TAGs is costly.

In 1988, Schabes and Joshi proposed an Earley-type parser for TAGs. It maintains the valid prefix property at the expense of its worst case complexity ($O(n^9)$-time). To our knowledge, it is the only known polynomial time parser for TAGs that maintains the valid prefix property.

In this paper, we explain why the valid prefix property is expensive to maintain for TAGs and we introduce a predictive left to right parser for TAGs that does not maintain the valid prefix property but that achieves an $O(n^6)$-time worst case behavior, $O(n^4)$-time for unambiguous grammars and linear time for a large class of grammars.

*This research was partially funded by ARO grant DAAL03-89-C0031PRI and DARPA grant N00014-90-J-1863. The difficulty of maintaining the valid prefix property for TAGS was first noticed in joint work with Vijay-Shanker in the context of deterministic parsing of tree-adjoining grammars (Schabes and Vijay-Shanker, 1990). I am indebted to Vijay-Shanker for numerous discussions on this topic. I am also indebted to Aravind Joshi for his suggestions and for his support of this research. The discussions I had with Mitchell Marcus greatly improved the presentation of the algorithm introduced in this paper. I would also like to thank Bob Frank, Bernard Lang, Fernando Pereira, Philip Resnik and Stuart Shieber for providing valuable comments.

Organization of the paper

This paper discusses of two subjects: the difficulty of parsing tree-adjoining grammars (TAGs) while maintaining the valid prefix property (Sections 1 and 2) and the design of a predictive left to right parser for TAGs (Section 3). Although the two topics are related, they can be read independently of each other.

1 Definition of the Valid Prefix Property

The valid prefix property is a property of left of to right parsing algorithms which guarantees that errors in the input are detected "as soon as possible".

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.[1]

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.

[1] 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 seen so far is a valid string of the language.

2 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 (Earley, 1968) 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 (Younger, 1967; Kasami, 1965). This follows from the path set complexity, as we will see.

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.[2] 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 is costly.[3] 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 (1985)[4] proposed a CKY-type parser for TAG that achieves $O(n^6)$-time worst case complexity.[5] As the original CKY parser for CFGs, this parser does not maintain the VPP. The Earley-type parser developed for TAGs (Schabes and Joshi, 1988) 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). However, the goal of our 1988 enterprise was to build a practical parser which behaves in practice better than its worst case complexity. Other parsers for TAGs have been proposed (Lang, 1988; Satta and Lavelli, 1990; Vijay-Shanker and Weir, 1990).[6] 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.

The second observation is in the context of deterministic left to right parsing of TAGs(Schabes and Vijay-Shanker, 1990) 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, which 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 (Weir, 1988) 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 Figure 1. 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 Figure 1).

[2] This result follows from Thatcher's work (1971), which defines frontier to root finite state tree automata.

[3] We assume familiarity with tree-adjoining grammars. See, for instance, the introduction by Joshi (Joshi, 1987).

[4] The parser is also reported in Vijay-Shanker (1987).

[5] However, this algorithm is not a practical parser for TAGs because, as is well known for CFGs, the average behavior of CKY-type parsers is the same as the worst case behavior.

[6] In a recent paper, Karen Harbusch (1990) claimed to have defined an $O(n^4 log(n))$ worst time general TAG parser based on the CKY parser for CFGs. However, since the paper does not include a proof of correctness and complexity, the relationship between the parser and the set of

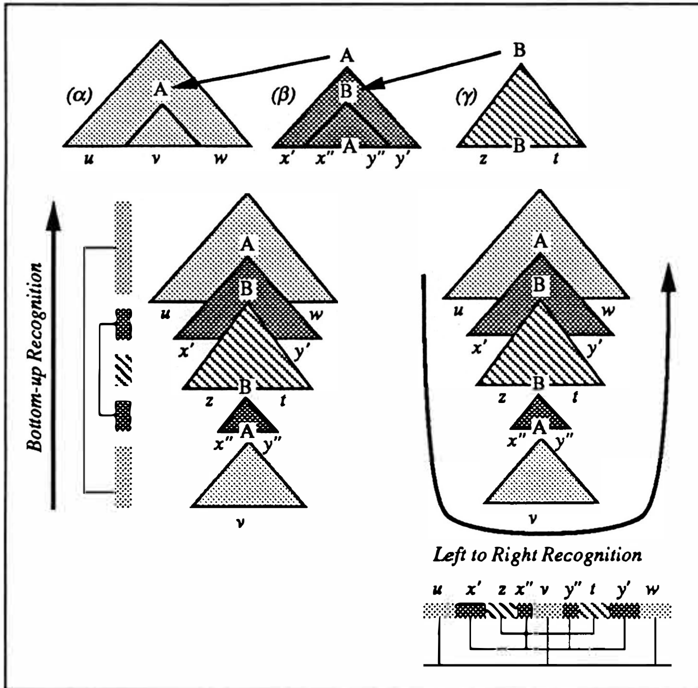

Figure 1: *Above*, a sequence of adjunctions; *below left*, bottom-up recognition of the derived tree; *right*, left to right recognition of the derived tree.

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 Figure 1) 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 Figure 1).

The context-free complexity of the path set of TAGs makes the valid prefix property costly to maintain. We suspect that the same difficulty arises for context-sensitive formalism which use operations such as adjoining or wrapping (Joshi et al., Forthcoming 1990).

languages it recognizes still needs to be determined.

3 A Predictive Left to Right Parser for TAGs

In this section, we define a new predictive left to right (Earley-style) parser for TAGs with adjoining constraints (Joshi, 1987). It is a bottom-up parser that uses some but not all the top-down information given by prediction. As a consequence, the parser does not satisfy the valid prefix property: it always detects errors but not as soon as possible. However, it achieves an $O(n^6)$-time worst case behavior, $O(n^4)$-time for unambiguous grammars and linear time for a large class of grammars (for example, the language $a^n b^n e c^n d^n$ is recognized in linear time). This parser as well as in the one introduced by Schabes and Joshi (1988) are practical parsers for TAGs since as is well known for CFGs, the average behavior of Earley-style parsers is superior to their worst case complexity. The algorithm has been modified to handle extensions of TAGs such as substitution, feature structures for TAGs and a version of multiple component TAG (these extensions are explained in Schabes [1990]).

3.1 Preliminary Concepts

Any *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 (Gorn, 1965) as defined by the following inductive definition: 0 is the address of the root node, k ($k \in \mathcal{N}$) is the address of the kth child of the root node, $x \cdot y$ (x is an address, $y \in \mathcal{N}$) is the address of the yth child of the node at address x.

Given a tree α and an address *address* in α, we define *Adjunct*(α, *address*) to be the set of auxiliary trees that can be adjoined at the node at address *address* in α. For TAGs with no constraints on adjunction, *Adjunct*(α, *address*) is the set of elementary auxiliary trees whose root node is labeled by α(*address*).

We define a *dotted tree* as a tree associated with a dot *above* or *below* and either *to the left* or *to the right* of a given node. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below): ${}^{la}_{lb}A^{ra}_{rb}$. We write $< \alpha, dot, pos >$ for a dotted tree in which the dot is at address *dot* and at position *pos* in the tree α.

The *tree traversal* we define for parsing TAGs consists of moving a dot in an elementary 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. An example of tree traversal is shown in Figure 2.

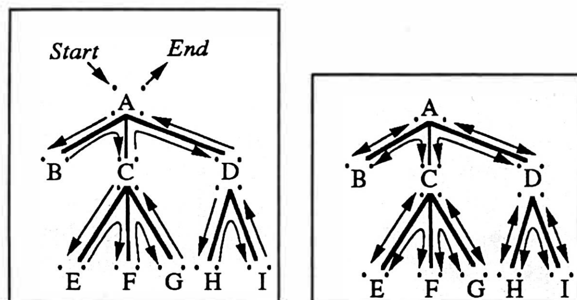

Figure 2: *Left*, left to right tree traversal; *right*, equivalent dot positions.

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.

We consider to equivalent two successive (according to the tree traversal) dot positions that do not cross a node in the tree (see Figure 2). For example the following equivalences hold for the tree α pictured in Figure 2: $< \alpha, 0, lb > \equiv < \alpha, 1, la >$, $< \alpha, 1, ra > \equiv < \alpha, 2, la >$, $< \alpha, 2, lb > \equiv < \alpha, 2 \cdot 1, la >$, $\cdots$.

3.2 Data Structures

We now define the data structures used by the parser. The input string is $a_1 \cdots a_n$ and the tree-adjoining grammar is $G = (\Sigma, NT, I, A)$: 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 and A is the set of auxiliary trees.

The algorithm uses one data structure: a state.[7]

A **state** s is defined as an 8-tuple,

$s = [\alpha, dot, pos, i, j, k, l, sat?]$ where:

- α is an elementary tree, initial or auxiliary tree: $\alpha \in I \cup A$.
- dot is the address of the dot in the tree α.
- pos 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; $pos \in \{la, lb, rb, ra\}$.
- 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.
- $sat?$ is a boolean; $sat? \in \{true, nil\}$.

The components α, dot, pos of a state define a dotted tree. Similarly to a dotted rule for context-free grammars defined by Earley (1968), a dotted tree splits a tree into two contexts: a left context that has been traversed and a right context that needs to be recognized.

The additional indices i, j, k, l record the portions of the input string.

The boolean $sat?$ indicates whether an adjunction has been recognized on the node at address dot in the tree α.

In the following, we will refer to one of the two equivalent dot positions for the 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 state. We will use to our convenience s or s' to refer to this unique state.

3.3 Analogy between Dotted Trees and Dotted Rules

There is a useful analogy between dotted TAG trees and dotted rules. It is by no mean a formal correspondence between TAG and a production system but it will give an intuitive understanding of the parser we define. It will also be used as a notation for referring to a dotted tree.

One can interpret a TAG elementary tree as a set of productions on pairs of trees and addresses (i.e. nodes). For example, the tree in Figure 2, let's call it α, can be written as:

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

Of course, the label of the node at address i in α is associated with each pair (α, i).[8] One can then relate a dotted tree to a dotted rule. 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 (tree in Figure 2). Note that the dotted trees $< \alpha, 2, ra >$ and $< \alpha, 3, la >$ are equivalent.

[7] We could have chosen to group the states into state sets as in (Earley, 1968) but we chose not to, allowing us to define an agenda driven parser.

[8] TAGs could be defined in term 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.

The dotted tree $< \alpha, 2, ra >$ is analogous to the dotted rule:

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

One can therefore put into correspondence a state defined on a dotted tree with a state defined on a dotted rule. A state $s = [\alpha, dot, pos, i, j, k, l, sat?]$ 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 of the dotted node when the dot is above to the left or to the right, and where $u = dot$ when the dot is below to the left or to the right.

3.4 Invariant of the Algorithm

The algorithm collects states into a set $\mathcal{C}$ called a *chart*. The algorithm maintains an invariant that is satisfied for all states in the chart $\mathcal{C}$. The correctness of the algorithm is a corollary of this invariant. The invariant is pictured in Figure 3 in terms of dotted trees. We informally describe it equivalently in terms of dotted rules. A state 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 if and only if the elementary tree α derives a tree such that:

(1) $\eta_1 \cdots \eta_y \stackrel{*}{\Rightarrow} a_i \cdots a_l$
(2) $(\alpha, f) \stackrel{*}{\Rightarrow} a_{j+1} \cdots a_k$

where f is the address of the foot node of α. (2) 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 the $pos = rb$, 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.[9]

$sat? = t$ 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.[10]

[9] The algorithm will set $sat?$ to t only when $pos = rb$.

[10] The derivations in TAG disallow more than one auxiliary tree adjoined on the same node.

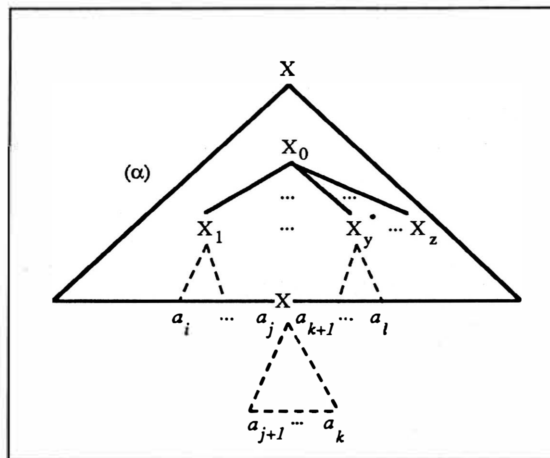

Figure 3: Invariant.

3.5 The Recognizer

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.

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

Depending on the existing states in the chart $\mathcal{C}$, new states are added to the chart by four processors until no more states can be added to the chart. The processors are: the *Predictor*, the *Completor*, the *Adjunctor* and the *Scanner*. If, in the final chart, there is a state corresponding to an initial tree completely recognized, i.e. with the dot to the right and above the root node which spans the input form position 0 to n (i.e. a state of the form $[\alpha, 0, ra, 0, -, -, n, nil]$), the input is recognized.

The recognizer for TAGs follows:

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

```
program recognizer
begin
  C = {[α,0,la,0,-,-,0,nil]| α ∈ I }
```

Apply one of the four processes on each state in the $\mathcal{C}$ until no more states can be added to the $\mathcal{C}$:
1. Scanner
2. Predictor
3. Completor
4. Adjunctor

If there is a state of the form $[\alpha, 0, ra, 0, -, -, n, nil]$ in $\mathcal{C}$ with $\alpha \in I$ then return acceptance otherwise return rejection .
end.

The initialization step
$\mathcal{C}$={ $[\alpha, 0, la, 0, -, -, 0, nil] | \alpha \in I$ }
puts all initial trees with the dot to the left and above the root node. The four processes are explained one by one in the four next sections.

3.5.1 Scanner

The *Scanner* is a bottom-up processor that scans the input string. It applies when the dot is to the left and above a terminal symbol. It consists of two cases: one when the terminal is not the empty string, and the other when it is.

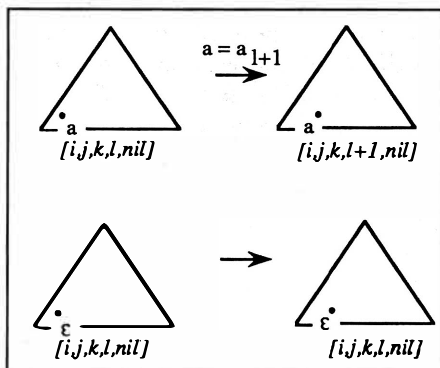

Figure 4: Scanner.

Scanner (see Figure 4):
It applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in \Sigma \cup \{\epsilon\}$.

- Case 1: $\alpha(dot) = a_{l+1}$
  The state $s = [\alpha, dot, ra, i, j, k, l+1, nil]$ is added to $\mathcal{C}$.
- Case 2: $\alpha(dot) = \epsilon$ (empty string)
  The state $s = [\alpha, dot, ra, i, j, k, l, nil]$ is added to $\mathcal{C}$.

3.5.2 Predictor

The *Predictor* constitutes the top-down processor of the parser. It predicts new states accordingly to the left context that has been been read.

The *Predictor* creates new states from a given state. It consists of three steps which are not applicable all 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 predictor that the VPP is violated since not all nodes that are predicted 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 Completor). Ruling them out during this step requires more complex data-structures and increases the complexity of the algorithm (Schabes and Joshi, 1988).

Predictor (see Figure 5):

- Step 1 applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$.

  If the conditions are satisfied, the states $\{[\beta, 0, la, l, -, -, l, nil] | \beta \in Adjunct(\alpha, dot)\ \}$ are added to $\mathcal{C}$.

- Step 2 applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$ and the node at address *dot* in α has no obligatory adjoining constraint.

  If the conditions are satisfied, the state $[\alpha, dot, lb, i, j, k, l, nil]$ is added to $\mathcal{C}$.

- Step 3 applies to a state of the form $s = [\alpha, dot, lb, i, j, k, l, nil]$ such that such that $\alpha \in A$, and such that *dot* is the address of the foot node of α.

  If the conditions are satisfied, for all elementary trees $\delta \in I \cup A$ and for all addresses dot' in δ such that $\alpha \in Adjunct(\delta, dot')$, the state $[\delta, 0, lb, l, -, -, l, nil]$ is added to $\mathcal{C}$.

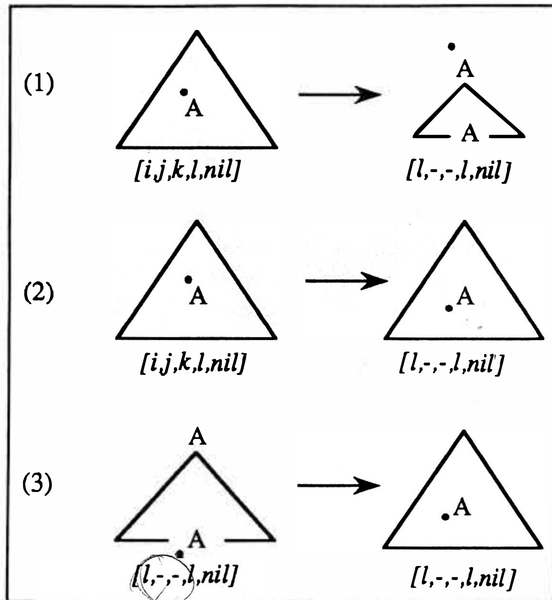

Figure 5: Predictor.

3.5.3 Completor

The *Completor* is a bottom-up processor that combines two states to form another state that spans a bigger portion of the input.

It consists of three 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. Steps 2 and 3 try to further recognize the same tree and concatenate boundaries of two states.

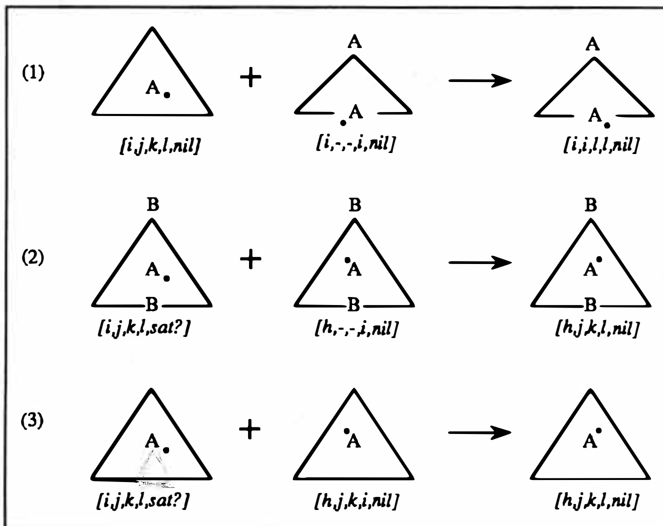

Figure 6: Completor.

Completor (see Figure 6):

- Step 1 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$, with a state $s_2 = [\beta, dot', lb, i, -, -, i, nil]$ such that $\beta \in Adjunct(\alpha, dot)$ and such that dot' is the address of the foot node of β. It adds the state $[\beta, dot', rb, i, i, l, l, nil]$ to $\mathcal{C}$.
- Step 2 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, sat?]$ such that $\alpha(dot) \in NT$, and s.t. the node at address dot in α subsumes the foot node of α, with a state $s_2 = [\alpha, dot, la, h, -, -, i, nil]$. It adds the state $[\alpha, dot, ra, h, j, k, l, nil]$ to $\mathcal{C}$.
- Step 3 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, sat?]$ such that $\alpha(dot) \in NT$ and s.t. the node at address dot in α does not subsume the foot node of α with a state $s_2 = [\alpha, dot, la, h, j, k, i, nil]$. It adds the state $[\alpha, dot, ra, h, j, k, l, nil]$ to $\mathcal{C}$.

3.5.4 Adjunctor

The *Adjunctor* is a bottom-up processor that combines two states by adjunction to form a state that spans a bigger portion of the input.

It consists of a single step.

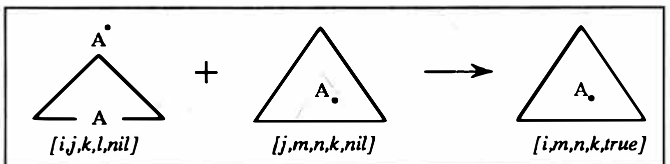

Figure 7: Adjunctor.

Adjunctor (see Figure 7):

It combines combines a state of the form $s_1 = [\beta, 0, rb, i, j, k, l, nil]$ and a state $s_2 = [\alpha, dot, rb, j, m, n, k, nil]$ such that $\beta \in Adjunct(\alpha, dot)$.

It adds the state $[\alpha, dot, rb, i, m, n, k, true]$ to $\mathcal{C}$.

3.6 An Example

We give an example that illustrates how the recognizer works. The grammar used for the example (see Figure 8) generates the language $L = \{a^n b^n e c^n d^n | n \geq 0\}$. The grammar consists of an initial tree α and an auxiliary tree β. There is a

null adjoining constraint on the root node and the foot node of β.

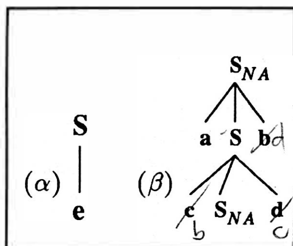

Figure 8: TAG generating $L = \{a^n b^n e c^n d^n | n \geq 0\}$

The input string given to the recognizer is: *aabbeccdd*. The corresponding chart is shown in Figure 9. For purpose of explanation, we have preceded each state with a number that uniquely identifies the state and we followed the state with the operation(s) that caused it to be placed into the chart. We used the following abbreviations: *init* for the initialization step, *pred(k)* for the Predictor applied to the state numbered by k, *sc(k)* for the Scanner applied to the state numbered by k, *compl(k+l)* for the combination with the Completor of the states numbered by k and l and *adj(k+l)* for the combination with the Adjunctor of the states numbered by k and l. With this convention, one can trace step by step the building of the chart. For example,

31. $[\beta, dot : 2, rb, 1, 4, 5, 8, t]$ *adj(30+24)*
stands for the state $[\beta, dot : 2, rb, 1, 4, 5, 8, t]$ uniquely identified by the number 31 which was placed into the chart by combining with the Adjunctor the states identified by the numbers 30 and 24.

The input is recognized since $[\alpha, 0, right, above, 0, -, -, 9, nil]$ is in the chart $\mathcal{C}$.

3.7 Implementation

The algorithm described in Section 3.5 can be implemented to follow an arbitrary search space strategy by using a priority function that ranks the states to be processed. The ranking function can also be defined to obtain a left to right behavior as in (Earley, 1968). Such a function may also very well be of statistical nature as for example in (Magerman and Marcus, 1991).

In order to bound the worst case complexity as stated in the next section, arrays must be used to implement efficiently the different processors. Due to the lack of space, we do not include the details of such implementation in this paper but they are found in (Schabes, 1991).

3.8 Correctness and Complexity

The algorithm is a general parser for TAGs with constraints on adjunction that takes in worst case $O(|G|^2 N n^6)$ time and $O(|G| N n^4)$ space, n being the length of the input string, $|G|$ the number of elementary trees in the grammar and N the maximum number of nodes in an elementary tree. The worst case complexity comes from the *Adjunctor* processor. An intuition of the validity of this result can be obtained by observing that that this processor (see Section 3.5.4) may be called at most $|G|^2 N n^6$ time since there are at most n^6 instances of the indices (i, j, k, l, m, n) and at most $|G|^2 N$ pairs of dotted trees to combine (α, β, dot). When it is used with unambiguous tree-adjoining grammars, the algorithm takes at most $O(|G|^2 N n^4)$-time[11] and linear time on a large class of grammars.

The proof of correctness consists in the proof of the invariant stated in Section 3.4.

Due to the lack of space, the details of the proofs of correctness and complexity are not given in this paper, but they are found in Schabes (1991).

3.9 The Parser

The algorithm that we described in section 3.5 is a recognizer. However, if we include pointers from a state to the other states (to a pair of states for the Completor and the Adjunctor or to a state for the Scanner and the Predictor) which caused it to be placed in the chart (in a similar manner to that shown in Figure 9), the recognizer can be modified to record all parse trees of the input string. The representation is similar to a shared forest. The worst case time complexity for the parser is the same as for the recognizer ($O(|G|^2 N n^6)$-time) but the worst case space complexity increases to $O(|G|^2 N n^6)$-space.

3.10 Extensions

The algorithm has been modified to handle extensions of TAGs such as substitution (Schabes et al., 1988), unification based TAGs (Vijay-Shanker and Joshi, 1988; Vijay-Shanker, 1991) and a version of multiple component TAG (see Schabes, 1990, for details on how to modify the parser to handle these extensions). It can also take advantage of lexicalized TAGs (Schabes and Joshi, 1989).

[11] This is a new upper-bound of the complexity of unambiguous TAG.

Input read	States 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)*	6. $[\beta, dot : 0, la, 1, -, -, 1, nil]$ *pred(5)*
a	7. $[\beta, dot : 2.1, la, 1, -, -, 1, nil]$ *pred(5)*	8. $[\beta, dot : 1, la, 1, -, -, 1, nil]$ *pred(6)*
aa	9. $[\beta, dot : 2, la, 1, -, -, 2, nil]$ *sc(8)*	10. $[\beta, dot : 0, la, 2, -, -, 2, nil]$ *pred(9)*
aa	11. $[\beta, dot : 2.1, la, 2, -, -, 2, nil]$ *pred(9)*	12. $[\beta, dot : 1, la, 2, -, -, 2, nil]$ *pred(10)*
aab	13. $[\beta, dot : 2.2, la, 2, -, -, 3, nil]$ *sc(11)*	14. $[\beta, dot : 2.2, lb, 3, -, -, 3, nil]$ *pred(13)*
aab	15. $[\beta, dot : 2.1, la, 3, -, -, 3, nil]$ *pred(14)*	16. $[\alpha, dot : 1, la, 3, -, -, 3, nil]$ *pred(14)*
aabb	17. $[\beta, dot : 2.2, la, 3, -, -, 4, nil]$ *sc(15)*	18. $[\beta, dot : 2.2, lb, 4, -, -, 4, nil]$ *pred(17)*
aabb	19. $[\beta, dot : 2.1, la, 4, -, -, 4, nil]$ *pred(18)*	20. $[\alpha, dot : 1, la, 4, -, -, 4, nil]$ *pred(18)*
aabbe	21. $[\alpha, dot : 0, rb, 4, -, -, 5, nil]$ *sc(20)*	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)*	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)*	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)*	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)*	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)*	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)*	36. $[\alpha, dot : 0, ra, 0, -, -, 9, nil]$ *compl(35+1)*

Figure 9: States constituting the chart for the input: $_0\, a\, _1\, a\, _2\, b\, _3\, b\, _4\, e\, _5\, c\, _6\, c\, _7\, d\, _8\, d\, _9$

4 Conclusion

We have shown that maintaining the valid prefix property for TAG parsing is costly because of the context-freeness of the path set of TAG derived trees.

In 1988, Schabes and Joshi introduced an Earley-style parser that satisfies the VPP however at a cost to its complexity ($O(n^9)$-time in the worst case but linear on some grammars). To our knowledge, it is the only known polynomial-time parser for TAG which satisfies the valid prefix property.

We have introduced a predictive left to right parser for TAGs which does not maintain the valid prefix property but takes at most $O(n^6)$-time in the worst case, $O(n^4)$-time for unambiguous grammars, and can behave linearly on some classes of grammars. The parser which we introduced is a practical parser since it often behaves better than its worst case complexity. It has been extended to handle extensions of TAGs such as unification based TAG and a restricted version of multiple component TAGs.

This predictive left to right parser can be adapted to other grammatical formalisms weakly equivalent to tree-adjoining languages (Joshi et al., Forthcoming 1990) such as linear index grammar, head grammars and a version of combinatory categorial grammars.

Bibliography

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

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

Saul Gorn. 1965. Explicit definitions and linguistic dominoes. In John Hart and Satoru Takasu, editors, *Systems and Computer Science.* University of Toronto Press, Toronto, Canada.

Karen Harbusch. 1990. An efficient parsing algorithm for Tree Adjoining Grammars. In 28th *Meeting of the Association for Computational Linguistics (ACL'90)*, Pittsburgh.

Aravind K. Joshi, K. Vijay-Shanker, and David Weir. Forthcoming, 1990. The convergence of mildly context-sensitive grammatical formalisms. In Peter Sells, Stuart Shieber, and Tom Wasow, editors, *Foundational Issues in Natual Language Processing.* MIT Press, Cambridge MA.

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

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.

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.

M. David Magerman and Mitchell P. Marcus. 1991. Pearl, a probabilistic chart parser. In *Proceedings of the Second International Workshop on Parsing Technologies*, Cancun, Mexico, February.

Giorgio Satta and Alberto Lavelli. 1990. A head-driven bidirectional recognizer for lexicalized TAGs. Unpublished manuscript.

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.

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

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.

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.

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.

Yves Schabes. 1991. A predictive left to right parser for tree-adjoining grammars. Technical report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia. In preparation.

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.

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

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

K. Vijay-Shanker and David J. Weir. 1990. Parsing constrained grammar formalisms. In preparation.

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

K. Vijay-Shanker. 1991. An unification based approach to Tree Adjoining Grammars. In preparation.

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

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