Parsing Some Constrained Grammar Formalisms

K. Vijay-Shanker*
University of Delaware

David J. Weir†
University of Sussex

In this paper we present a scheme to extend a recognition algorithm for Context-Free Grammars (CFG) that can be used to derive polynomial-time recognition algorithms for a set of formalisms that generate a superset of languages generated by CFG. We describe the scheme by developing a Cocke-Kasami-Younger (CKY)-like pure bottom-up recognition algorithm for Linear Indexed Grammars and show how it can be adapted to give algorithms for Tree Adjoining Grammars and Combinatory Categorial Grammars. This is the only polynomial-time recognition algorithm for Combinatory Categorial Grammars that we are aware of.

The main contribution of this paper is the general scheme we propose for parsing a variety of formalisms whose derivation process is controlled by an explicit or implicit stack. The ideas presented here can be suitably modified for other parsing styles or used in the generalized framework set out by Lang (1990).

1. Introduction

This paper presents a scheme to extend known recognition algorithms for Context-Free Grammars (CFG) in order to obtain recognition algorithms for a class of grammatical formalisms that generate a strict superset of the set of languages generated by CFG. In particular, we use this scheme to give recognition algorithms for Linear Indexed Grammars (LIG), Tree Adjoining Grammars (TAG), and a version of Combinatory Categorial Grammars (CCG). These formalisms belong to the class of *mildly context-sensitive grammar formalisms* identified by Joshi (1985) on the basis of some properties of their generative capacity. The parsing strategy that we propose can be applied to the formalisms listed as well as others that have similar characteristics (as outlined below) in their derivational process. Some of the main ideas underlying our scheme have been influenced by the observations that can be made about the constructions used in the proofs of the equivalence of these formalisms and Head Grammars (HG) (Vijay-Shanker 1987; Weir 1988; Vijay-Shanker and Weir 1993).

There are similarities between the TAG and HG derivation processes and that of Context-Free Grammars (CFG). This is reflected in common features of the parsing algorithms for HG (Pollard 1984) and TAG (Vijay-Shanker and Joshi 1985) and the CKY algorithm for CFG (Kasami 1965; Younger 1967). In particular, what can happen at each step in a derivation can depend only on which of a finite set of "states" the derivation is in (for CFG these states can be considered to be the nonterminal symbols). This property, which we refer to as the **context-freeness property**, is important because it allows one to keep only a limited amount of context during the recognition process,

* Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716. E-mail: vijay@udel.edu.
† School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, U.K. E-mail: davidw@cogs.susx.ac.uk.

which results in polynomial time algorithms. In the recognition algorithms mentioned above for CFG, HG, and TAG this is reflected in the fact that the recognizer can encode intermediate stages of the derivation with a bounded number of states. An array is used whose entries are associated with a given component of the input. In the case of the CKY algorithm, the presence of a particular nonterminal in an array entry is used to encode the fact that the nonterminal derives the associated substring of the input. The context-freeness of CFG has the consequence that there is no need to encode the way, or ways, in which a nonterminal came to be placed in an array entry.

In this respect, the derivation processes of CCG and LIG would *appear* to differ from that of CFG. In these systems unbounded stacklike structures replace the role played by nonterminals in controlling derivation choices. This would seem to suggest that the context-freeness property of CFG, HG, and TAG derivations no longer holds. Unbounded stacks can encode an unbounded number of earlier derivation choices. In fact, while the path sets[1] of CFG, HG, and TAG derivation trees are regular languages, the path sets of CCG and LIG are context-free languages. With respect to recognition algorithms, this suggests that the array (whose entries contain nonterminals in the case of CFG) would need to contain complete encodings of unbounded stacks giving an exponential time algorithm.

However, in LIG and CCG, the use of stacks to control derivations is limited in that different branches of a derivation cannot share stacks. Thus, despite the above observations, the context-freeness property does in fact hold. A detailed explanation of why this is so will be presented below. We propose a method to extend the CKY algorithm to handle the limited use of stacks found in CCG and LIG. We have chosen to adapt the CKY algorithm since it is the simplest form of bottom-up parsing. A similar approach using Earley algorithm is also possible, although not considered here. Since the use of the stacks is most explicit in the LIG formalism we describe our approach in detail by developing a recognition algorithm for LIG (Sections 2 and 3). We then show how the general approach suggested in the parser for LIG can be tailored to CCG (in Section 4). In the above discussion TAG has been grouped with HG. However, TAG can also be viewed as making use of stacks in the same way as LIG and CCG. In Section 5 we show how the LIG algorithm presented in Section 3 can be adapted for TAG.

2. Linear Indexed Grammars

An Indexed Grammar (Aho 1968) can be viewed as a CFG in which objects are nonterminals with an associated stack of symbols. In addition to rewriting nonterminals, the rules of the grammar can have the effect of pushing or popping symbols on top of the stacks that are associated with each nonterminal. Gazdar (1988) discussed a restricted form of Indexed Grammars in which the stack associated with the nonterminal on the left of each production can only be associated with one of the occurrences of nonterminals on the right of the production. Stacks of bounded size are associated with other occurrences of nonterminals on the right of the production. We call this Linear Indexed Grammars (LIG).[2]

1 The path set of a tree is the set of strings labeling paths from the root to the frontier of the tree. The path set of a tree set is the union of path sets of trees in the set.

2 The name Linear Indexed Grammars is used by Duske and Parchmann (1984) to refer to a different restriction on Indexed Grammars in which production was restricted to have only a single nonterminal on their right-hand side.

Definition 2.1
A LIG, G, is denoted by (V_N, V_T, V_I, S, P) where

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

We adopt the convention that α, β (with or without subscripts and primes) denote members of V_I^*, and γ denotes a stack symbol. As usual, A, B, C will denote nonterminals, a, b, c will denote terminals, and u, v, w will denote members of V_T^*.

Definition 2.2
A pair consisting of a nonterminal, say A, and a string of stack symbols, say α, will be called an **object** of the grammar and will be written as $A(\alpha)$. Given a grammar, G, we define the set of objects $V_C(G) = \{ A(\alpha) \mid A \in V_N, \alpha \in V_I^* \}$.

We use Υ to denote strings in $(V_C(G) \bigcup V_T)^*$. We write $A(\cdot\cdot\,\alpha)$ to denote the nonterminal A associated with an arbitrary stack α with the string on top. Also, we use $A()$ to denote that an empty stack is associated with A. The general form of a production in a LIG is:
$A(\cdot\cdot\,\alpha) \rightarrow w_1 A_1(\alpha_1) w_2 \ldots A_{i-1}(\alpha_{i-1}) w_i A_i(\cdot\cdot\,\alpha_i) w_{i+1} A_{i+1}(\alpha_{i+1}) \ldots A_n(\alpha_n) w_{n+1}$ for $n \geq 0$ and $w_1 \ldots, w_{n+1}$ are members of V_T^*.

Definition 2.3
The derivation relation, $\Longrightarrow$, is defined below. If the above production is used then for any $\beta \in V_I^*$, $\Upsilon_1, \Upsilon_2 \in (V_C(G) \bigcup V_T)^*$:

$$\Upsilon_1 A(\beta\alpha)\Upsilon_2 \Longrightarrow \Upsilon_1 w_1 A_1(\alpha_1) w_2 \ldots A_{i-1}(\alpha_{i-1}) w_i A_i(\beta\alpha_i) w_{i+1} A_{i+1}(\alpha_{i+1}) \ldots A_n(\alpha_n) w_{n+1} \Upsilon_2.$$

We use $\overset{*}{\Longrightarrow}$ as the reflexive, transitive closure of $\Longrightarrow$. As a result of the *linearity* in the general form of the rules, we can observe that the stack $\beta\alpha$ associated with the object in the left-hand side of the derivation and $\beta\alpha_i$ associated with one object in the right-hand side have the initial part β in common. In the derivation above, we will say that this object $A_i(\beta\alpha_i)$ is the **distinguished child** of $A(\beta\alpha)$. Given a derivation, the **distinguished descendant** relation is the reflexive, transitive closure of the distinguished child relation.

The language generated by a LIG, G, $L(G) = \{ w \mid S() \underset{G}{\overset{*}{\Longrightarrow}} w \}$.

Example 2.1
The LIG, $G = (\{S, T\}, \{a, b, c\}, \{\gamma_a, \gamma_b\}, S, P)$ generates $\{ wcw \mid w \in \{a, b\}^+ \}$ where P contains the following productions.

$$S(\cdot\cdot) \rightarrow aS(\cdot\cdot\,\gamma_a) \qquad S(\cdot\cdot) \rightarrow bS(\cdot\cdot\,\gamma_b) \qquad S(\cdot\cdot) \rightarrow T(\cdot\cdot)$$

$$T(\cdot\cdot\,\gamma_a) \rightarrow T(\cdot\cdot)a \qquad T(\cdot\cdot\,\gamma_b) \rightarrow T(\cdot\cdot)b \qquad T() \rightarrow c$$

A derivation tree for the string *abbcabb* is given in Figure 1.

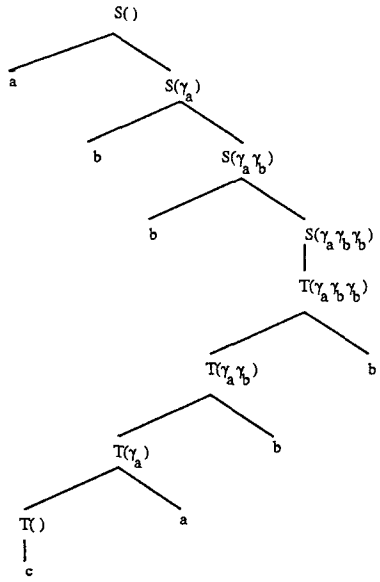

Figure 1
Derivation tree for LIG.

In this paper rather than adopting the general form of rules as given above, we restrict our attention to grammars whose rules have the following form. In fact, this can be easily seen to constitute a normal form for LIG.

1. $A(\alpha) \rightarrow \varepsilon$ where $\varepsilon \in V_T \cup \{\epsilon\}$ and length of α, $len(\alpha) \geq 1$.
2. $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot \gamma_p) A_s(\alpha_s)$ where $m \geq 0$.
3. $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot \gamma_p)$ where $m \geq 0$.
4. $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot \gamma_p)$ where $m \geq 0$.

We allow at most two symbols in the right-hand side of productions because we intend to develop CKY-style algorithms. In the above rules we say that $A_p(\cdot\cdot \gamma_p)$ is the **primary constituent** and $A_s(\alpha_s)$ is the **secondary constituent**. Notice also that in a derivation using such a rule, the primary constituent yields the distinguished child. (In grammatical theories that use a stack of subcategorized arguments, the top of the stack in the primary constituent determines which secondary constituent it can combine with.)

2.1 Terminators

Let us consider how we may extend the CKY algorithm for the recognition of LIG. Given a fixed grammar G and an input $a_1 \ldots a_n$, the recognition algorithm will complete an $n \times n$ array P such that an encoding of $A(\alpha)$ is stored in $P[i, d]$ if and only if $A(\alpha) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$. The algorithm will operate bottom-up. For example, if G contains the rule $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot \gamma_p) A_s(\alpha_s)$ and we find an encoding of $A_p(\alpha_p \gamma_p)$ in $P[i, d_p]$ and an encoding of $A_s(\alpha_s)$ in $P[i + d_p, d_s]$ then an encoding of $A(\alpha_p \gamma_1 \ldots \gamma_m)$ will be stored

in $P[i, d_p + d_s]$. What encoding scheme should be used? The most straightforward possibility would be to store a complete encoding of $A(\alpha_p \gamma_1 \ldots \gamma_m)$ in $P[i, d_p + d_s]$. However, in general, if an object $A(\alpha)$ derives a string of length d then the length of α is $\mathcal{O}(d)$.[3] Hence there can be $\mathcal{O}(k^d)$ objects that derive a substring of the input (of length d), for some constant k. Hence, the space and time complexity of this algorithm is exponential in the worst case.[4]

The inefficiency of this approach can be seen by drawing an analogy with the following algorithm for CFG. Suppose rather than storing sets of nonterminals in each array entry, we store a set of trees containing all derivation subtrees that yield the corresponding substring. The problem with this is that the number of derivation trees is exponential with respect to the length of the string spanned. However, there is no need to store derivation trees since in considering the combination of subderivation trees in the CFG, only the nonterminals at the root of the tree are relevant in determining whether there is a production that licenses the combination.

Likewise because of the last-in first-out behavior in the manipulation of stacks in LIG, we will argue that it is not necessary to store the entire stack. For instance, consider the derivation (depicted by the tree shown in Figure 2) from the point of view of recording the derivation in a bottom-up parser (such as CKY). Let a node η_1 labeled $B(\beta\gamma_1 \ldots \gamma_k \ldots \gamma_m)$ be a *distinguished descendant* of a node η labeled $A(\beta\gamma_1 \ldots \gamma_k)$ as shown in the figure. Viewing the tree bottom-up, let the node η, labeled $A(\beta\gamma_1 \ldots \gamma_k)$, be the first node above the node η_1, labeled $B(\beta\gamma_1 \ldots \gamma_k \ldots \gamma_m)$, where γ_k gets exposed as the top of the stack. Because of the last-in first-out behavior, every *distinguished* descendant of η above η_1 will have a label of the form $A'(\beta\gamma_1 \ldots \gamma_k\alpha)$ where $len(\alpha) \geq 1$. In order to record the derivation from $A(\beta\gamma_1 \ldots \gamma_k)$ it would be sufficient to store A and $\gamma_1 \ldots \gamma_k$ if we could also access the entry that records the derivation from $A_t(\beta\gamma_t)$. In the entry for η, using a pointer to the entry for $A_t(\beta\gamma_t)$ would enable the recovery of the stack below the top k symbols, $\gamma_1 \ldots \gamma_k$. However, this scheme works well only when $k \geq 2$. For instance, when $k = 1$, suppose we recorded only A, γ_1, and a pointer to entry for $A_t(\beta\gamma_t)$. Suppose that we are looking for the symbol below γ_1, i.e., the top of β. Then it is possible that in a similar way the latter entry could also record just A_t, γ_t, and a pointer to some other entry to retrieve β. This situation can occur arbitrarily many times.

Consider the derivation depicted in Figure 3. In this derivation we have indicated the branch containing only the distinguished descendants. We will assume that the node labeled $D(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1 \ldots \gamma'_{m'})$ is the closest distinguished descendant of $C(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1)$ such that every node between them will have a label of the form $C'(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1\alpha')$ where $len(\alpha') \geq 1$. Therefore, any node between that labeled $C(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1)$ and $B(\beta\gamma_1 \ldots \gamma_m)$ will have a label of the form $C''(\beta\gamma_1 \ldots \gamma_{k-1}\alpha'')$ where $len(\alpha'') \geq 1$. Now the entries representing derivations from both $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k)$ and $C(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1)$ could point back to the entry for the derivation from $A_t(\beta\gamma_t)$, whereas the entry for $C'(\beta\gamma_1 \ldots \gamma_{k-1}\gamma'_1\alpha')$ will point back to the entry for $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k)$.

We shall now formalize these notions by defining a **terminator**.

3 For instance, consider the grammar in Example 2.1 and the derivation in Figure 1. In general we can have derivations of the form $T(\gamma_a\gamma_b^n) \stackrel{*}{\Longrightarrow} cab^n$. However, if there exists productions of the form $A(\alpha) \rightarrow \epsilon$ then the length of the stack in objects is not even bounded by the length of strings they derive.

4 The CCG parsing algorithms that have been proposed so far follow this strategy (Pareschi and Steedman 1987; Tomita 1988).

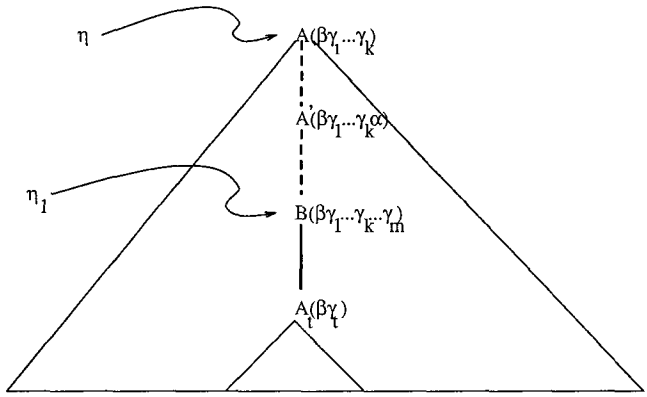

Figure 2
Recovering the rest of stack-1.

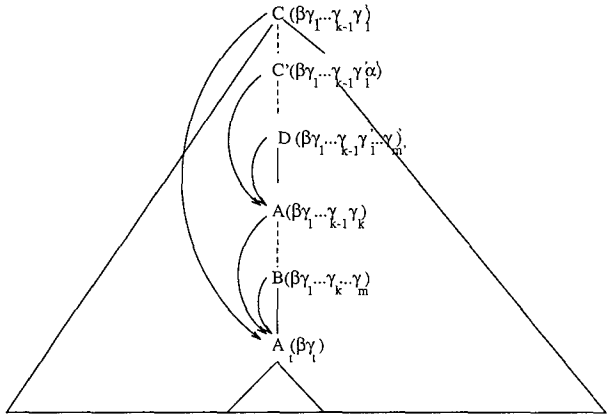

Figure 3
Recovering the rest of stack-2.

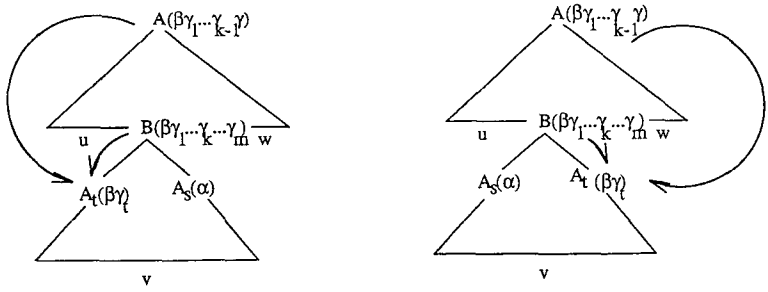

Figure 4
Definition of a Terminator.

Definition 2.4
Suppose that we have the derivation tree in Figure 4 that depicts the following derivation:

$$\begin{aligned} A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right) &\stackrel{*}{\Longrightarrow} uB\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k \ldots \gamma_m\right) w \\ &\Longrightarrow uA_t\left(\beta\gamma_t\right) A_s\left(\alpha_s\right) w \\ &\stackrel{*}{\Longrightarrow} uvw \end{aligned}$$

or similarly:

$$\begin{aligned} A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right) &\stackrel{*}{\Longrightarrow} uB\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_i \ldots \gamma_m\right) w \\ &\Longrightarrow uA_s\left(\alpha_s\right) A_t\left(\beta\gamma_t\right) w \\ &\stackrel{*}{\Longrightarrow} uvw \end{aligned}$$

where the following conditions hold

- $2 \leq k \leq m$
- The nodes labeled $B\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k \ldots \gamma_m\right)$ and $A_t\left(\beta\gamma_t\right)$ are distinguished descendants of the node labeled $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right)$ in the respective trees.
- For any distinguished descendent labeled $C\left(\alpha'\right)$ between the nodes labeled $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right)$ and $B\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k \ldots \gamma_m\right)$, α' is of the form $\beta\gamma_1 \ldots \gamma_k\alpha$ where $len\left(\alpha\right) \geq 1$. Note that the nodes labeled $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right)$ and $B\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k \ldots \gamma_m\right)$ need not be different.

The node labeled $A_t\left(\beta\gamma_t\right)$ is the ***k*-terminator** of the node labeled $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right)$.

When it is clear from context, rather than saying that a node is a terminator of another we will assume that terminators have been defined on objects that participate in a derivation as well. For instance, in the above *derivations*, we will say that $A_t\left(\beta\gamma_t\right)$ is the k-terminator of $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma\right)$. Also when the derivation is clear from context, we will omit the mention of the derivation (or derivation tree). Additionally, we will say that a node (object) has a terminator, if it has a k-terminator for some k.

We will now state some properties of terminators that influence the design of our recognition algorithm.

Definition 2.5
Given a grammar, G, define MCL(G) (Maximum Change in Length) as:
$\mathrm{MCL(G)} = max\left\{\, \mathrm{m} \mid A\left(\cdot\cdot\, \gamma_1 \ldots \gamma_m\right) \rightarrow \Upsilon_1 A_p\left(\cdot\cdot\, \gamma_p\right) \Upsilon_2 \text{ is a production of } G \,\right\}$

Henceforth, we will write MCL since the grammar in question will always be known from context.

Observation 2.1
In a derivation tree, if a node (say η) has a k-terminator (say η_t) then η_t is a distinguished descendant of η. If the node η is labeled $A\left(\beta\alpha\right)$ (where $len\left(\alpha\right) = k$) then the node η_t must be labeled $A_t\left(\beta\gamma_t\right)$ for some $A_t \in V_N$ and $\gamma_t \in V_I$. Furthermore, $2 \leq k \leq \mathrm{MCL}$.

Observation 2.2
In a derivation tree, if a node has a k-terminator then it has a unique terminator.

If η is the node in question then we are claiming here that not only does it have a unique k-terminator but also that there does not exist k' with $k' \neq k$ such that η has a k'-terminator. To see why this is the case, let some node η have a k-terminator (for some k), say η_t. Using Observation 2.1 we can assume that they are labeled $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma)$ and $A_t(\beta\gamma_t)$, respectively, where we have $(k-1) \geq 1$. From the definition of terminators we can assume that the parent of the terminator, η_t, is a node (say η') that has a label of the form $B(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k \ldots \gamma_m)$. Since (from the definition of terminators) every node between η and η' (inclusive) must have a label of the form $C(\beta\gamma_1 \ldots \gamma_{k-1}\alpha')$ where $len(\alpha') \geq 1$, it immediately follows that η_t is the closest *distinguished* descendant of η such that the length of the stack in the object labeling η_t is strictly less than the length of the stack in the object labeling η. From this, the uniqueness of terminators follows.

Observation 2.3
Consider the derivation $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma) \stackrel{*}{\Longrightarrow} uA_t(\beta\gamma_t)w \stackrel{*}{\Longrightarrow} uvw$ where $A_t(\beta\gamma_t)$ is the k-terminator of $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma)$. Then for any β' and v', if $A_t(\beta'\gamma_t) \stackrel{*}{\Longrightarrow} v'$ then we have the derivation $A(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma) \stackrel{*}{\Longrightarrow} uA_t(\beta'\gamma_t)w \stackrel{*}{\Longrightarrow} uv'w$ where $A_t(\beta'\gamma_t)$ is the k-terminator of $A(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma)$.

This follows from the fact that the derivation of $uA_t(\beta\gamma_t)w$ from $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma)$ is independent of β. Therefore we can replace $A_t(\beta\gamma_t) \stackrel{*}{\Longrightarrow} v$ by $A_t(\beta'\gamma_t) \stackrel{*}{\Longrightarrow} v'$. This is a very important property that is crucial for obtaining polynomial-time algorithm.

Note that not all nodes have terminators. For example, if a node labeled $A(\alpha)$ is the parent of a node labeled a (i.e., corresponding to the use of the production $A(\alpha) \rightarrow a$ where a is a terminal symbol) then obviously this node does not have a terminator.

Definition 2.6
Given a grammar, G, we define MTL(G) (Maximum Length in Terminal production) as:

$$\text{MTL(G)} = max\{\, len(\alpha) \mid \text{A}(\alpha) \rightarrow \varepsilon \text{ is a production of } G \text{ where } \varepsilon \in V_T \bigcup \{\epsilon\} \,\}.$$

As in the case of MCL, we will use MTL rather than MTL(G).

Observation 2.4
In the derivation $A(\alpha) \stackrel{*}{\Longrightarrow} w$ if $len(\alpha) >$ MTL then $A(\alpha)$ has a terminator.

There must be at least two steps in the above derivation since $len(\alpha) >$ MTL. However, we can assume that the node (say η) in question labeled by the object $A(\alpha)$ has a distinguished descendant, say η', with label $B(\beta)$ such that $B(\beta) \stackrel{1}{\Longrightarrow} \varepsilon$. Therefore, $len(\beta) \leq$ MTL and we may rewrite w as $u\varepsilon v$. Since $len(\alpha) > len(\beta)$ we can find the closest distinguished descendant of η labeled $C(\alpha')$ for some C, α' such that $len(\alpha') < len(\alpha)$. That node is the terminator of η from the arguments made in Observation 2.2.

The above observations will be used in the following sections to explain the way in which we represent derivations in the parsing table. We conclude this section with an observation that has a bearing on the steps of the recognition algorithm.

Observation 2.5
Consider the following derivation.

$$\begin{aligned} A\left(\beta\gamma_1 \ldots \gamma_{k-1}\right) &\stackrel{1}{\Longrightarrow} \Upsilon_1 A_p\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k\right)\Upsilon_2 \\ &\stackrel{*}{\Longrightarrow} u_1 A_p\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k\right) u_2 \\ &\stackrel{*}{\Longrightarrow} u_1 v_1 A_t\left(\beta\gamma_t\right) v_2 u_2 \\ &\stackrel{*}{\Longrightarrow} u_1 v_1 w v_2 u_2 \end{aligned}$$

where $A_p\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k\right)$ is the distinguished child of $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\right)$ and $A_t\left(\beta\gamma_t\right)$ is the k-terminator of $A_p\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k\right)$. $A_t\left(\beta\gamma_t\right)$ is the $(k-1)$-terminator of $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\right)$ if and only if $k > 2$. If $k = 2$ then $A\left(\beta\gamma_1\right)$ has a terminator if and only if $A_t\left(\beta\gamma_t\right)$ does. In fact, in this case, if $A_t\left(\beta\gamma_t\right)$ has a k'-terminator then that terminator is also the k'-terminator of $A\left(\beta\gamma_t\right)$.

This can be seen by considering the derivation shown in Figure 3 and noting the sharing of the terminator of $C\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_1'\right)$ and $A\left(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k\right)$.

3. Recognition Algorithms

As in the CKY algorithm we will use a two-dimensional array, P, such that if $A\left(\alpha\right) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ then a representation of this derivation will be recorded with an encoding of $A\left(\alpha\right)$ in $P\left[i,d\right]$. Here we assume that the given input is $a_1 \ldots a_n$. We start our discussion by considering the data structures we use to record such objects and derivations from them.

3.1 Anatomy of an Entry

We mentioned earlier that the stack in an object can be unboundedly large. We must first find a compact way to store encodings of such objects whose size is not bounded by the grammar. In this section we provide some motivation for the encoding scheme used in the recognition algorithm by considering the bottom-up application of the rule and the encoding of the primary constituent:

$$A\left(\cdot\cdot\, \gamma_1 \ldots \gamma_m\right) \rightarrow A_p\left(\cdot\cdot\, \gamma_p\right) A_s\left(\alpha_s\right)$$

The Head. An object with nonterminal A_p and top of stack γ_p will match the primary category of this rule. Thus, the first requirement is that at least this much of the object must be included in every entry since it is needed to determine if the rule can apply. This component is denoted $\langle A_p, \gamma_p \rangle$ and called the **head** of the entry. Thus, in general, an entry in $P\left[i,d\right]$ with the head $\langle A, \gamma \rangle$ encodes derivations of $a_i \ldots a_{i+d-1}$ from an object of the form $A\left(\beta\gamma\right)$ for some $\beta \in V_I^*$.

Terminator-pointer. An encoding of the object $A_p\left(\beta\gamma_p\right)$ (the primary constituent) that derives the substring $a_i \ldots a_{i+d_p-1}$ (of the input string $a_1 \ldots a_n$) will be stored in the array element $P\left[i,d_p\right]$ in our CKY-style recognition algorithms. Now consider the encoding of $A_p\left(\beta\gamma_p\right)$ for some sufficiently long $\beta\gamma_p$. While the head, $\langle A_p, \gamma_p \rangle$, of the entry is sufficient to determine whether the object in question can match the primary category of the rule, we will need to store more information in order that we can determine the content of the rest of the stack. In the above production, if $m = 0$ then the combination of $A_p\left(\beta\gamma_p\right)$ and $A_s\left(\alpha_s\right)$ results in $A\left(\beta\right)$. In order to record the derivation from $A\left(\beta\right)$, we need to know the top symbol in the stack β, i.e., the symbol below the top of the stack associated with the primary constituent. We need to recover the identity of

this symbol from the encoding of the primary category. This is why we introduced the notion of terminators. As mentioned in Section 2.1, terminators can be used to access information about the rest of the stack. In the encoding of $A_p(\beta\gamma_p)$, we will store information that allows us to access the encoding of its terminator. The part of the entry encoding the terminator will be called **terminator pointer**.

The Middle. Note that the object $A_p(\beta\gamma_p)$ (in the derivation $A_p(\beta\gamma_p) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1}$) can have a k-terminator where k is between 2 and MCL. Therefore, from Observation 2.1 it follows that the terminator-pointer can only be used to determine the $(k+1)^{st}$ symbol from the top. Therefore, assuming that $\beta = \beta'\gamma_1 \ldots \gamma_{k-1}$, the terminator-pointer will allow us to access β'. (Recall from the definition, a k-terminator of $A(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma_p)$ will have the form $A_t(\beta'\gamma_t)$. Thus the $(k+1)^{st}$ symbol from the top in $A(\beta\gamma_p)$ is the same as the symbol below the top of the stack of the terminator.) Thus, we will need to record the string $\gamma_1 \ldots \gamma_{k-1}$ in the encoding of $A_p(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma_p)$ as well. This part of the entry will be called the **middle**.

To summarize, the entry stored in $P[i,d_p]$ (where $\beta'\gamma_1 \ldots \gamma_{k-1}\gamma_p$ is assumed to be sufficiently long that we know $A_p(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma_p)$ is guaranteed to have a terminator) will have a **head**, (A_p, γ_p); and a **tail** comprised of a **middle**, $\gamma_1 \ldots \gamma_{k-1}$; and a **terminator-pointer**. Note that the length of the middle must be at least one, but at most MCL $-$ 1, since from Observation 2.1, we know $2 \leq k \leq$ MCL. We will call an entry of this kind a **terminator-type** entry.

We will now discuss what we need to store in order to point to the terminator. Suppose we would like to record in $P[i,d]$ the derivation of $a_i \ldots a_{i+d-1}$ from $A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma)$ as shown below. We assume that $A_t(\beta\gamma_t)$ is the terminator in this derivation.

$$\begin{aligned} A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1}A_t(\beta\gamma_t)\, a_{t+d_t} \ldots a_{i+d-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1}a_t \ldots a_{t+d_t-1}a_{t+d_t} \ldots a_{i+d-1} \\ &= a_i \ldots a_{i+d-1} \end{aligned}$$

From Observation 2.3, it follows that it would be sufficient to use $(\langle A_t, \gamma_t\rangle, [t,d_t])$ as the terminator-pointer. This is because any entry with the head $\langle A_t, \gamma_t\rangle$ in $P[t,d_t]$ will represent in general a derivation $A_t(\beta'\gamma_t) \stackrel{*}{\Longrightarrow} a_t \ldots a_{t+d_t-1}$. This not only matches the above case, but even if $\beta' \neq \beta$, from the Observation 2.1, we have

$$A(\beta'\gamma_1 \ldots \gamma_{k-1}\gamma) \stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1}A_t(\beta'\gamma_t)\, a_{t+d_t} \ldots a_{i+d-1} \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}.$$

Thus, the use of the head information (plus the two indices) in the terminator-pointer captures the essence of Observation 2.3. It is this structure-sharing that allows us to achieve polynomial bounds for space and time. Note that the string derived from the terminator, $a_t \ldots a_{t+d_t-1}$, is a substring of $a_i \ldots a_{i+d-1}$. In such a case, i.e., when $i \leq t$ and $i + t \geq t + d_t$, we will say that $\langle t,d_t\rangle \leq \langle i,d\rangle$. We define $\langle t,d_t\rangle < \langle i,d\rangle$ if $\langle t,d_t\rangle \leq \langle i,d\rangle$ and $\langle t,d_t\rangle \neq \langle i,d\rangle$. Since any terminator-type entry in $P[i,d]$ can only have terminator-pointers of the form $(\langle A_t, \gamma_t\rangle, [t,d_t])$ where $\langle t,d_t\rangle \leq \langle i,d\rangle$, the number of terminator-type entries in $P[i,d]$ is $O(d^2)$.

Definition 3.1
Given a grammar, G, define MSL(G) (Maximum Secondary constituent's stack Length) as MSL(G) = $max\{\, len(\alpha_s) \mid A_s(\alpha_s)$ is the secondary constituent of a production $\}$

Henceforth we will use MSL rather than MSL(G).

We now consider the question of when a terminator-type entry is appropriate. Of course, if $A(\alpha) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ we could store such an entry in $P[i, d_p]$ only when $A(\alpha)$ has a terminator in this derivation. From Observation 2.4 we know that if $len(\alpha) >$ MTL then there exists a terminator of $A(\alpha)$ in this derivation. However, it is possible that for some grammar MSL $>$ MTL. Therefore even when $len(\alpha) >$ MTL (i.e., the object has a terminator) $A(\alpha)$ can still match the secondary category of a rule if $len(\alpha) \leq$ MSL. In order to verify that an object matches the secondary category of a rule we need to consider the entire stack in the object. When $A(\alpha) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ and length of α does not exceed MSL, it would be convenient to store A as well as the entire stack α because such an object can potentially match a secondary category of a rule. To be certain that such an object is stored in its entirety when $len(\alpha) \leq$ MSL, the terminator-type entry can only be used when $len(\alpha) > max(\text{MSL}, \text{MTL})$. However, we prefer to use the terminator-type entry for representing a derivation from $A(\alpha)$ only when its terminator, say $A_t(\beta)$, is such that $len(\beta) \geq max(\text{MSL}, \text{MTL})$ rather than when $len(\alpha) > max(\text{MSL}, \text{MTL})$. Again, we point out that this choice is made only for convenience and because we feel it leads to a simpler algorithm. The alternate choice could also be made, which would lead to a slightly different algorithm.

Definition 3.2
Define the constant TTC (Terminal-Type Case) as TTC = max(MSL, MTL). In a derivation $A(\beta\gamma_1 \ldots \gamma_k) \stackrel{*}{\Longrightarrow} w$ we will say that $A(\beta\gamma_1 \ldots \gamma_k)$ has the TC-property iff it has a k-terminator, say $A_t(\beta\gamma_t)$, such that $len(\beta\gamma_t) \geq$ TTC.

If $A(\beta\gamma_1 \ldots \gamma_k) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$, where $A(\beta\gamma_1 \ldots \gamma_k)$ does not have the TC-property then we record the object in its entirety in $P[i, d]$. In order for such an entry to have the same format as the terminator-type entry, we say that the entry has a head $\langle A, \gamma_k \rangle$; a tail with a middle $\gamma_1 \ldots \gamma_{k-1}$ and a *nil* terminator-pointer. Note that in this case the middle can be an empty string; for instance, when we encode $A(\gamma) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$. In general, if $\alpha = \beta\gamma$ then we say $top(\alpha) = \gamma$ and $rest(\alpha) = \beta$. If $\alpha = \epsilon$ then we say that $top(\alpha) = rest(\alpha) = \epsilon$.

To summarize, the structure of an entry in $P[i, d]$ is described by the following rules.

- An entry consists of a head and a tail.
- A head consists of a nonterminal and a stack symbol.
- A tail consists of a middle and a terminator-pointer. The exact nature of the middle and the terminator-pointer are as given below.
  - The terminator-pointer may be of the form $(\langle A_t, \gamma_t \rangle, [t, d_t])$ where $A_t \in V_N, \gamma_t \in V_I$ and $\langle t, d_t \rangle \leq \langle i, d \rangle$. In this case, the middle is a string of stack symbols of length at least one. This form of a terminator pointer is used in the encoding of a derivation from an object if its terminator has a stack length greater than or equal to TTC. Recall that we had called this type of an entry a **terminator-type entry**.
  - A terminator-pointer can be a *nil*. Then the middle is a (possibly empty) string of stack symbols. However, the length of the middle is less than TTC + MCL − 1. This form of a terminator pointer is used in the encoding of a derivation from an object if it does not satisfy the TC-property; i.e., either it has no

terminator or if the terminator exists then its stack length is less than TTC.

3.2 Recognition Algorithms for LIG

Since the full algorithm involves a number of cases, we develop it in stages by restricting the forms of productions. The first algorithm that considers the most restricted form of productions introduces much of what lies at the core of our approach. Next we relax these restrictions to some degree. After giving the algorithm at this stage, we switch to discuss how this algorithm can be adapted to yield one for CCG. Later, in Section 5, we consider further relaxation of the restrictions on the form of LIG productions, which can help us produce an algorithm for TAG.

Regardless of which set of restrictions we consider, in every algorithm we shall establish that the following proposition holds.

Proposition 3.1

- $(\langle A, \gamma_k\rangle\ (\gamma_1 \ldots \gamma_{k-1}, (\langle A_t, \gamma_t\rangle, [t, d_t])))) \in P[i, d]$ if and only if for some $\beta \in V_I^*$,

$$\begin{aligned} A(\beta\gamma_1 \ldots \gamma_{k-1}\gamma_k) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A(\beta\gamma_t) a_{t+d_t-1} \ldots a_{i+d-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1} \end{aligned}$$

  where $A_t(\beta\gamma_t)$ is the k-terminator of $A(\beta\gamma_1 \ldots \gamma_k)$ and $len(\beta\gamma_t) \geq$ TTC.

- $(\langle A, \gamma_k\rangle\ (\gamma_1 \ldots \gamma_{k-1}, nil)) \in P[i, d]$ if and only if

$$A(\gamma_1 \ldots \gamma_{k-1}\gamma_k) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$$

  where in this derivation $A(\gamma_1 \ldots \gamma_{k-1}\gamma_k)$ does not have the TC-property.

3.2.1 Algorithm 1. Recall that the general form of rules that are to be considered are as follows.

1. $A(\alpha) \rightarrow \varepsilon$ where $\varepsilon \in \{\epsilon\} \cup V_T$, and $len(\alpha) \geq 1$.
2. $A(\cdot\cdot\, \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\, \gamma_p)\, A_s(\alpha_s)$
3. $A(\cdot\cdot\, \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s)\, A_p(\cdot\cdot\, \gamma_p)$.
4. $A(\cdot\cdot\, \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\, \gamma_p)$.

At this stage we assume that the following restrictions hold of the above rules.

In the first type of production we assume that $\varepsilon \in V_T$ and $len(\alpha) \geq 1$. Thus MTL ≥ 1.

$len(\alpha_s) \geq 1$ in productions of type 2 and type 3, i.e., MSL ≥ 1.

There are no productions of type 4.

We will now give the following rules that specify how entries get added in the parsing array. The control structure of the algorithm (a CKY-style dynamic programming structure) will be added later. We assume that the input given is $a_1 \ldots a_n$, where $n \geq 1$.

Initialization Phase
In the initialization phase of the algorithm we store lexical objects (objects deriving a terminal symbol in one step) entirely in a single entry. In other words,

Rule 1.L

$$\frac{A(\alpha) \rightarrow a \quad a = a_i \quad 1 \leq i \leq n}{(\langle A, top(\alpha)\rangle\ (rest(\alpha), nil)) \in P[i,1]}$$

Inductive phase
Here productions of type 2 and type 3 will be considered. Let us assume the presence of the following production in the grammar: $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\gamma_p)\, A_s(\alpha_s)$.[5]

Suppose that while considering which entries are to be included in $P[i,d]$ we find the following for some d_p, d_s such that $d_p + d_s = d$.

- The entry $\left(\langle A_p, \gamma_p\rangle\ \left(\beta_p, tp_p\right)\right) \in P[i,d_p]$. This is consistent with the rule's primary constituent. Regardless of whether $tp_p = nil$ or not, for some $\beta \in V_t^*$: $A_p(\beta\beta_p\gamma_p) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1}$. That is, when $tp_p = nil$ we have $\beta = \epsilon$.
- The entry $(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)) \in P[i+d_p, d_s]$. This is consistent with the rule's secondary object. Thus if $d = d_p + d_s$ we may assume $A_s(\alpha_s) \stackrel{*}{\Longrightarrow} a_{i+d_p} \ldots a_{i+d-1}$.

From the presence of the two entries specified above (and the derivations they represent) we have $A(\beta\beta_p\gamma_1 \ldots \gamma_m) \Longrightarrow A_p(\beta\beta_p\gamma_p)\, A_s(\alpha_s) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$. This derivation must be recorded with an entry in $P[i,d]$. The content of the entry depends on several factors: the value of m; whether or not the terminator-pointer in the entry for the primary constituent (i.e., tp_p) is *nil*; and the length of the middle in this entry (i.e., β_p). These determine whether or not the new entry will be a terminator-type entry. We have cases for $m = 0$, $m = 1$ and $m \geq 2$.

CASE WHEN $m = 0$
The new object to be stored is $A(\beta\beta_p)$. The top of the stack in this object can be obtained from the stack associated with the primary constituent. How this is done depends on whether the entry encoding the primary constituent is of terminator type or not.

When $m = 0$ **and** $tp_p = nil$
This means that the primary constituent has been represented in its entirety; i.e., the primary constituent is $A_p(\beta_p\gamma_p)$. Since $tp_p = nil$ the primary constituent does not satisfy the TC-property (i.e., it does not have a terminator with a stack of length greater than or equal to TTC), the new constituent too cannot be encoded using a terminator-type entry. Therefore,

Rule 2.ps.L

$$\frac{(\langle A_p, \gamma_p\rangle\ (\beta_p, nil)) \in P[i,d_p] \quad (\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)) \in P[i+d_p, d-d_p]}{(\langle A, top(\beta_p)\rangle\ (rest(\beta_p), nil)) \in P[i,d]}$$

5 Similar arguments can be used when we consider the production: $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s)\, A_p(\cdot\cdot\gamma_p)$.

The following rule is the counterpart of Rule2.ps.L[6] that corresponds to the use of the production $A(\cdot\cdot) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot\gamma_p)$.

Rule 2.sp.L

$$\frac{(\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \in P[i, d_s] \quad (\langle A_p, \gamma_p\rangle (\beta_p, nil)) \in P[i + d_s, d - d_s]}{(\langle A, top(\beta_p)\rangle (rest(\beta_p), nil)) \in P[i, d]}$$

When $m = 0$ and $tp_p \neq nil$

Let the entry for the primary constituent be $(\langle A_p, \gamma_p\rangle (\beta_p, (\langle A_t, \gamma_t\rangle, [t, d_t])))$. Since the primary constituent is $A_p(\beta\beta_p\gamma_p)$ we will assume that its terminator is $A_t(\beta\gamma_t)$ where $len(\beta\gamma_t) \geq$ TTC. Note also that $len(\beta_p\gamma_p) \geq 2$. The entry for the new object $(A(\beta\beta_p))$ is determined based on whether $len(\beta_p) = 1$ or $len(\beta_p) > 1$. In the latter case the $len(\beta_p\gamma_p)$-terminator of the primary constituent is the $len(\beta_p)$-terminator of the new object. This is not so in the former case, as noted in Observation 2.5.

Considering the latter case first, i.e., $len(\beta_p) > 1$, we may write β_p as $\gamma_1 \ldots \gamma_{k-1}\gamma_k$ where $k \geq 2$. Since in this case the new object and the primary constituent have the same terminator and since the primary constituent has the TC–property ($tp_p \neq nil$), the new object must also be encoded with a terminator-type entry. Thus we have the following rule:

Rule 3.ps.L

$$\frac{\begin{array}{l}(\langle A_p, \gamma_p\rangle (\gamma_1 \ldots \gamma_k, tp_p)) \in P[i, d_p] \\ tp_p = (\langle A_t, \gamma_t\rangle, [t, d_t]), k \geq 2 \end{array} \qquad (\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \in P[i + d_p, d - d_p]}{(\langle A, \gamma_k\rangle (\gamma_1 \ldots \gamma_{k-1}, tp_p)) \in P[i, d]}$$

Henceforth we shall give the *ps* versions of the rules only and omit *sp* versions.

Now let us consider the case when $len(\beta_p) = 1$. Rewriting β_p as γ_1, the entries represent derivation for $\beta \in V_I^*$ ($len(\beta\gamma_1) = len(\beta\gamma_t) \geq$ TTC).

$$\begin{aligned} A(\beta\gamma_1) &\stackrel{*}{\Longrightarrow} A_p(\beta\gamma_1\gamma_p) A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\beta\gamma_t) a_{t+d_t} \ldots a_{i+d_p-1} A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} a_t \ldots a_{t+d_t-1} a_{t+d_t} \ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \end{aligned}$$

where $A_t(\beta\gamma_t)$ is the 2-terminator of $A_p(\beta\gamma_1\gamma_p)$. From Observation 2.5 it follows that if $A_t(\beta\gamma_t)$ has a terminator then the terminator of $A(\beta\gamma_1)$ in this derivation is the same as the terminator of $A_t(\beta\gamma_t)$; and if $A_t(\beta\gamma_t)$ has no terminator then neither does $A(\beta\gamma_1)$. Additionally, in this derivation $A(\beta\gamma_1)$ satisfies the TC-property if and only if $A_t(\beta\gamma_t)$ has the TC-property. That is, we should use a terminator-type entry to record this derivation from $A(\beta\gamma_1)$ if and only if a terminator-type entry has been used for $A_t(\beta\gamma_t)$. Since these two objects share the same terminator (if it exists) the terminator-pointer must be the same when we record derivations from them. Therefore, suppose we use the terminator-pointer of $(\langle A_p, \gamma_p\rangle (\beta_p, (\langle A_t, \gamma_t\rangle, [t, d_t])))$ to locate an entry $(\langle A_t, \gamma_t\rangle (\beta_t, tp_t)) \in P[t, d_t]$. This would suggest the addition of the entry

6 Here *L* indicates a rule we use in LIG parsing; *ps* indicates that the primary constituent appears before the secondary constituent. Similarly, *sp* will be used to indicate that the secondary constituent appears before the primary constituent.

$(\langle A, \gamma_1\rangle (\beta_t, tp_t))$ to $P[i,d]$, regardless of whether or not $tp_t = nil$. However, we give the two cases ($tp_t = nil$ or $tp_t = (\langle A_r, \gamma_r\rangle, [r, d_r])$ for some A_r, γ_r, r, d_r) in the form of two different rules. This is because (as we shall see later) these two rules will have to appear in different points of the control-structure of the parsing algorithm.

Rule 4.ps.L

$$\frac{\begin{array}{c}(\langle A_p, \gamma_p\rangle (\gamma_1, (\langle A_t, \gamma_t\rangle, [t, d_t]))) \\ \in P[i, d_p]\end{array} \quad \begin{array}{c}(\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \\ \in P[i + d_p, d - d_p]\end{array} \quad \begin{array}{c}(\langle A_t, \gamma_t\rangle (\beta_t, nil)) \\ \in P[t, d_t]\end{array}}{(\langle A, \gamma_1\rangle (\beta_t, nil)) \in P[i,d]}$$

Rule 5.ps.L

$$\frac{\begin{array}{c}(\langle A_p, \gamma_p\rangle (\gamma_1, (\langle A_t, \gamma_t\rangle, [t, d_t]))) \\ \in P[i, d_p]\end{array} \quad \begin{array}{c}(\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \\ \in P[i + d_p, d - d_p]\end{array} \quad \begin{array}{c}(\langle A_t, \gamma_t\rangle (\beta_t, tp_t)) \\ \in P[t, d_t] \\ tp_t = (\langle A_r, \gamma_r\rangle, [r, d_r])\end{array}}{(\langle A, \gamma_1\rangle (\beta_t, tp_t)) \in P[i,d]}$$

CASE WHEN $m = 1$

The length of the stack in the new object is equal to that of the primary object. In fact, the terminator of the primary object (if it exists) is the same as the terminator of the new object, and when the primary object has no terminator neither does the new object. Therefore the encoding of the new object can easily be derived from that of the primary object by simply modifying the head (to change the top of the stack symbol). Thus we have:

Rule 6.ps.L

$$\frac{(\langle A_p, \gamma_p\rangle (\beta_p, nil)) \in P[i, d_p] \quad (\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \in P[i + d_p, d - d_p]}{(\langle A, \gamma_1\rangle (\beta_p, nil)) \in P[i,d]}$$

Rule 7.ps.L

$$\frac{(\langle A_p, \gamma_p\rangle (\beta_p, (\langle A_t, \gamma_t\rangle, [t, d_t]))) \in P[i, d_p] \quad (\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \in P[i + d_p, d - d_p]}{(\langle A, \gamma_1\rangle (\beta_p, (\langle A_t, \gamma_t\rangle, [t, d_t]))) \in P[i,d]}$$

CASE WHEN $m \geq 2$

If the primary constituent is $A_p (\beta\beta_p\gamma_p)$ then the new constituent is $A (\beta\beta_p\gamma_1 \ldots \gamma_m)$. In fact, in this case, we have the primary constituent being the m-terminator of $A (\beta\beta_p\gamma_1 \ldots \gamma_m)$. Of course, this does not mean that the derivation from the new object should be recorded with the use of a terminator-type entry. We use the terminator-type entry only when $len(\beta_p\gamma_p) \geq$ TTC. In order to determine the length of this stack we have to use the entry for the primary constituent (i.e., $(\langle A_p, \gamma_p\rangle (\beta_p, tp_p)) \in P[i, d_p]$) and consider whether this is a terminator-type entry or not (i.e., whether $tp_p = nil$ or not).

When $m \geq 2$ and $tp_p \neq nil$
Therefore the length of the stack of the terminator of the primary constituent is greater than or equal to TTC. This means that stack length of the primary constituent (the terminator of the new object) exceeds TTC. Thus we have the following rule:

Rule 8.ps.L

$$\frac{\begin{array}{c}\left(\langle A_p, \gamma_p\rangle \left(\beta_p, tp_p\right)\right) \in P[i, d_p] \\ tp_p = \left(\langle A_t, \gamma_t\rangle, [t, d_t]\right) \qquad \left(\langle A_s, top(\alpha_s)\rangle \left(rest(\alpha_s), nil\right)\right) \in P[i + d_p, d - d_p]\end{array}}{\left(\langle A, \gamma_m\rangle \left(\gamma_1 \ldots \gamma_{m-1}, \left(\langle A_p, \gamma_p\rangle, [i, d_p]\right)\right)\right) \in P[i, d]}$$

When $m \geq 2$ and $tp_p = nil$
The primary constituent (which is the terminator of the new object) should be represented in its entirety. Therefore, in order to determine whether we have to encode the new object with a terminator-type entry or not, we have to look at the entry for the primary constituent. Thus we obtain the following rules:

Rule 9.ps.L

$$\frac{\begin{array}{c}len\left(\beta_p\gamma_p\right) < \text{TTC} \\ \left(\langle A_p, \gamma_p\rangle \left(\beta_p, nil\right)\right) \in P[i, d_p] \qquad \left(\langle A_s, top(\alpha_s)\rangle \left(rest(\alpha_s), nil\right)\right) \in P[i + d_p, d - d_p]\end{array}}{\left(\langle A, \gamma_m\rangle \left(\beta_p\gamma_1 \ldots \gamma_{m-1}, nil\right)\right) \in P[i, d]}$$

Rule 10.ps.L

$$\frac{\begin{array}{c}len\left(\beta_p\gamma_p\right) \geq \text{TTC} \\ \left(\langle A_p, \gamma_p\rangle \left(\beta_p, nil\right)\right) \in P[i, d_p] \qquad \left(\langle A_s, top(\alpha_s)\rangle \left(rest(\alpha_s), nil\right)\right) \in P[i + d_p, d - d_p]\end{array}}{\left(\langle A, \gamma_m\rangle \left(\gamma_1 \ldots \gamma_{m-1}, \left(\langle A_p, \gamma_p\rangle, [i, d_p]\right)\right)\right) \in P[i, d]}$$

In the discussions that follow, we find it convenient to refer to the entries mentioned in the above rules as either **antecedent** entries (or entries that appear in the antecedent) of a rule or **consequent** entry (or entry that appears in the consequent) of a rule. For example, $\left(\langle A_p, \gamma_p\rangle \left(\beta_p, nil\right)\right)$ in $P[i, d_p]$ and $\left(\langle A_s, top(\alpha_s)\rangle \left(rest(\alpha_s), nil\right)\right)$ in $P[i + d_p, d - d_p]$ are the antecedent entries of Rule 10.ps.L and $\left(\langle A, \gamma_m\rangle \left(\gamma_1 \ldots \gamma_{m-1}, \left(\langle A_p, \gamma_p\rangle, [i, d_p]\right)\right)\right)$ that is added to $P[i, d]$ is the entry in the consequent of Rule 10.ps.L.

3.3 The Control Structure

We will start by giving a simple control structure for the recognition algorithm that follows the dynamic programming style used in the CKY algorithm.

In this section we modify the notation for entries slightly. In the above discussion, the terminator-pointer of a terminator-type entry contains a pair of indices representing input positions. Thus, in effect, P is a four-dimensional array. As an alternative to saying that $(\langle A,\gamma\rangle\ (\beta,(\langle A',\gamma'\rangle,[t,d'_t])))$ is in $P[i,d]$ we will sometimes say $(\langle A,\gamma\rangle\ (\beta,\langle A',\gamma'\rangle))$ is in $P[i,d]\,[t,d_t]$. Also as an alternative to saying $(\langle A,\alpha\rangle\ (\beta,nil))$ is in $P[i,d]$ we will sometimes say $(\langle A,\alpha\rangle\ (\beta,nil))$ is in $P[i,d]\,[0,0]$. Thus P can be considered to be an array of size $n\times n\times(n+1)\times(n+1)$.

In the specification of the algorithm (Figure 5) we will not restate all the rules we discussed in the previous section. Instead we will only indicate where in the control structure each rule fits. As an example, when we state "Use Rule 2.ps.L with $d_p=d'$" within the i,d, and d' loops we mean the following: for current values of i,d, and d' (and hence d_p,d_s) consider every production of the form $A(\cdot\cdot\,\gamma_1\ldots\gamma_m)\rightarrow A_p\,(\cdot\cdot\,\gamma_p)\ A_s\,(\alpha_s)$ with $m=0$. For each such production, look for entries of the form $(\langle A_p,\gamma_p\rangle\ (\beta_p,nil))\in P[i,d_p]\,[0,0]$ for some β_p and $(\langle A_s,top\,(\alpha_s)\rangle\ (rest\,(\alpha_s),nil))\in P[i+d_p,d-d_p]\,[0,0]$. In the event we find such entries, we add $(\langle A,top\,(\beta_p)\rangle\ (rest\,(\beta_p),nil))$ to $P[i,d]\,[0,0]$ if it is not already there.

Since the entries in $P[i,d]$ have the form $(\langle A,\gamma\rangle\ (\beta,(\langle A_t,\gamma_t\rangle,[t,d_t])))$ (where $\langle t,d_t\rangle\leq\langle i,d\rangle$) or the form $(\langle A,\gamma\rangle\ (\beta,nil))$, there are $O(d^2)$ many entries in $P[i,d]$ (where $1\leq i<n$ and $1\leq d\leq n-d$). Thus space complexity of this algorithm is $O(n^4)$. Note that within the body within the r loop will be attempted for all possible values of i,d,d',t,d_t,r,d_r. Since the range of each loop is $O(n)$, the time complexity is $O(n^7)$.

The asymptotic complexity of the above algorithm can be improved to $O(n^6)$ with a simple rearrangement of the control structure. The key point here is that the steps involving the use of rules 5.ps.L and 5.sp.L can be split into two parts each. Consider, for example, the use of the Rule 5.ps.L, which is repeated below.

Rule 5.ps.L

$$\frac{\begin{array}{ccc}(\langle A_p,\gamma_p\rangle\ (\gamma_1,(\langle A_t,\gamma_t\rangle,[t,d_t])))\\ \in P[i,d_p]\end{array}\qquad\begin{array}{c}(\langle A_s,top\,(\alpha_s)\rangle\ (rest\,(\alpha_s),nil))\\ \in P[i+d_p,d-d_p]\end{array}\qquad\begin{array}{c}(\langle A_t,\gamma_t\rangle\ (\beta_t,tp_t))\\ \in P[t,d_t]\\ tp_t=(\langle A_r,\gamma_r\rangle,[r,d_r])\end{array}}{(\langle A,\gamma_1\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle,[r,d_r])))\in P[i,d]}$$

This rule corresponds to the use of the production $A(\cdot\cdot)\rightarrow A_p\,(\cdot\cdot\,\gamma_p)\ A_s\,(\alpha_s)$. The values of i,d,d',t,d_t are necessary to determine the span of the substrings derived from the primary constituent and the secondary constituent, and the values of i,d,t,d_t,r,d_r are needed to locate the entry for the terminator, i.e., $(\langle A_t,\gamma_t\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle,[r,d_r])))$ and to place the new entry in the appropriate parsing table element. That is, the values of r and d_r are not required for the first part and the value of d' need not be known for the second part. This indicates that the second part need not be done within the loop for d'. Therefore, we can modify the control structure in the following way. Within the t loop (which appears within the loops for d,i,d',d_t) we find the entries for the primary and secondary constituents. Having found the two relevant entries, we must record the head of the new entry $\langle A,\beta_p\rangle$ and the terminator-pointer of the primary constituent, i.e., $(\langle A_t,\gamma_t\rangle,[t,d_t])$. We can do this by using a two-dimensional array called TEMP where we store $\langle A,\gamma_1,A_t,\gamma_t\rangle$. Outside the d' loop (and hence outside the loops for t and d_t as well), but within the loops for i and d, we can have the loops that vary t,d_t,r,d_r (note $\langle r,d_r\rangle<\langle t,d_t\rangle$) in order to locate the entry for the terminator by using the information recorded in TEMP. Finally, having found the entry for the

Algorithm 1

```
begin
    for i:= 2 to n do
        Initialization phase
        Use Rule 1
    for d := 2 to n do % d loop
        for i := 1 to n − d + 1 do % i loop
            begin
                for d' := 1 to d − 1 do % d' loop
                    begin
                        Use Rule 2.ps.L, 6.ps.L, 9.ps.L, 10.ps.L with d_p = d'.
                        for d_t := (d' − 1) to 1 do % d_t loop
                            for t := i to (i + d' − d_t) do % t' loop
                                begin
                                    Use Rule 3.ps.L, 4.ps.L, 7.ps.L, 8.ps.L with d_p = d'
                                    for d_r := d_t to 1 do
                                        for r := t to t + d_t − d_r do
                                            begin
                                                Use Rule 5.ps.L with d_p = d'
                                            end
                                        % end of d_r loop
                                    % end of r loop
                                end
                            % end of t loop
                        % end of d_t loop
                        for d_t := (d − d' − 1) to 1 do % d_t loop
                            for t := (i + d') to (i + d − d_t) do % t' loop
                                begin
                                    Use Rule 3.ps.L, 4.ps.L, 7.ps.L, 8.ps.L with d_s = d'
                                    for d_r := d_t − 1 to 1 do
                                        for r := t to (t + d_t − d_r) do
                                            begin
                                                Use Rule 5.sp.L with d_s = d'
                                            end
                                        % end of r loop
                                    % end of d_r loop
                                end
                            % end of t loop
                        % end of d_t loop
                    end
                % end of d' loop
            end
        % end of i loop
    % end of d loop
```

Figure 5
Algorithm 1.

terminator we then store the resulting entry in $P[i,d]$. These steps are captured by the following rules. For a specific value of $\langle i,d\rangle$ we have

Rule 5.i.ps.L

$$\frac{\begin{array}{c}(\langle A_p,\gamma_p\rangle\ (\gamma_1,(\langle A_t,\gamma_t\rangle\ ,[t,d_t])))\\ \in P[i,d_p]\end{array}\qquad\begin{array}{c}(\langle A_s,top(\alpha_s)\rangle\ (rest(\alpha_s),nil))\\ \in P[i+d_p,d-d_p]\end{array}}{\langle A,\gamma_1,A_t,\gamma_t\rangle \in \mathrm{TEMP}[\mathrm{t},\mathrm{d_t}]}$$

Rule 5.ii.ps.L

$$\frac{\langle A,\gamma_1,A_t,\gamma_t\rangle \in \mathrm{TEMP}[\mathrm{t},\mathrm{d_t}]\quad (\langle A_t,\gamma_t\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r]))) \in P[t,d_t]}{(\langle A,\gamma_1\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r]))) \in P[i,d]}$$

Similarly, we assume we have the pair Rule 5.i.sp.L and Rule 5.ii.sp.L corresponding to Rule 5.sp.L. This leads to the algorithm given in Figure 6. In this algorithm we drop the *sp* rules and specify the *ps* rules only for the sake of simplicity.

The correctness of Algorithm 2 can be established from the correctness of Algorithm 1 (which is established in Appendix A) and the following Lemma.

Lemma 3.1
Given a grammar G and an input $a_1 \ldots a_n$ an entry $(\langle A,\gamma\rangle\ (\beta,tp))$ is added to $P[i,d]$ by Algorithm 1 if and only if $(\langle A,\gamma\rangle\ (\beta,tp))$ is added to $P[i,d]$ by Algorithm 2.

Outline of Proof: Using induction on d. The base case corresponding to $d = 1$ involves only the initialization step, which is the same in the two algorithms. The only difference between the two algorithms (apart from the control structure) is the use of Rule 5.ps.L (and Rule 5.sp.L) by Algorithm 1 versus the use of Rule 5.i.ps.L and Rule 5.ii.ps.L (Rule 5.i.sp.L and Rule 5.i.sp.L) in Algorithm 2. Rule 5.ps.L is used to add entries of the form $(\langle A,\gamma_1\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r])))$. We can establish that $(\langle A,\gamma_1\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r])))$ is added to $P[i,d]$ due to the application of Rule 5.ps.L if and only if there exist entries of the form $(\langle A_p,\gamma_p\rangle\ (\gamma_1,(\langle A_t,\gamma_t\rangle\ ,[t,d_t])))$ in $P[i,d_p]$; $(\langle A_s,top(\alpha_s)\rangle\ (rest(\alpha_s),nil))$ in $P[i+d_p,d-d_p]$; $(\langle A_t,\gamma_t\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r])))$ in $P[t,d_t]$; and the production $A(\cdot\cdot) \rightarrow A_p\ (\cdot\cdot\ \gamma_p)\ A_s\ (\alpha_s)$. Using induction, we can establish that these entries exist if and only if $\langle A,\gamma_1,A_t,\gamma_t\rangle$ is added to $\mathrm{TEMP}[\mathrm{t},\mathrm{d_t}]$ using Rule 5.ps.i.L (or Rule 5.sp.i.L) and $(\langle A,\gamma_1\rangle\ (\beta_t,(\langle A_r,\gamma_r\rangle\ ,[r,d_r])))$ is added to $P[i,d]$ using Rule 5.ii.ps.L.

4. Combinatory Categorial Grammars

Combinatory Categorial Grammars (CCG) (Steedman 1985, 1986) are extensions of Classical Categorial Grammars in which both function composition and function application are allowed. In addition, forward and backward slashes are used to place conditions concerning the relative ordering of adjacent categories that are to be combined.

Definition 4.1
The set of **categories** generated from a set, V_N, of atomic categories is defined as the smallest set such that all members of V_N are categories, and if c_1, c_2 are categories then so are (c_1/c_2) and $(c_1\backslash c_2)$.

Algorithm 2

```
begin
    for i:= 1 to n do
        Initialization phase
        Use Rule 1
    for d := 2 to n do % d loop
        for i := 1 to n − d + 1 do % i loop
            begin
                Initialize TEMP[t, d_t] to φ for all ⟨t, d_t⟩ ≤ ⟨i, d⟩
                for d_p := 1 to d − 1 do % d_p loop
                        Use Rule 2.ps.L, 6.ps.L, 9.ps.L, 10.ps.L
                    for d_t := d_p − 1 to 1 do % d_t loop
                        for t := i to i + d_p − d_t do % t loop
                                Use Rule 3.ps.L, 4.ps.L, 5.i.ps.L, 7.ps.L, 8.ps.L
                        % end of t loop
                    % end of d_t loop
                % end of d_p loop
                for d_t := d − 1 to 1 do % d_t loop
                    for t := i to i + d − d_t do % t loop
                        for d_r := d_t − 1 to 1 do
                            for r := t to t + d_t − d_r do
                                begin
                                    Use Rule 5.ii.ps.L
                                end
                            % end of r loop
                        % end of d_r loop
                    % end of d_t loop
                % end of t loop
            end
        % end of i loop
    % end of d loop
```

Figure 6
Algorithm 2.

Definition 4.2
A CCG, G, is denoted by (V_T, V_N, S, f, R) where

V_T is a finite set of terminals (lexical items),
V_N is a finite set of nonterminals (atomic categories),
S is a distinguished member of V_N,
f is a function that maps each element of V_T to a finite set of categories,
R is a finite set of combinatory rules, where combinatory rules have the following form.

1. A forward rule has the following form where $m \geq 0$.

$$(x/y) \quad (y|_1 z_1|_2 \ldots |_m z_m) \rightarrow (x|_1 z_1|_2 \ldots |_m z_m)$$

2. A backward rule has the following form where $m \geq 0$.

$$(y|_1 z_1|_2 \ldots |_m z_m) \quad (x\backslash y) \rightarrow (x|_1 z_1|_2 \ldots |_m z_m)$$

Here $x, y, z_1, \ldots, z_m$ are meta-variables and $|_1, \ldots, |_m \in \{\backslash, /\}$. For $m = 0$ these rules correspond to function application and for $m > 0$ to function composition. Note that the set R contains a finite subset of these possible forward and backward rules; i.e., for a given CCG only some of the combinatory rules will be available.

Definition 4.3
In the forward and backward rules given above, we say that (x/y) (resp. $(x\backslash y)$) is the **primary** constituent of the forward (resp. backward) rules and $(y|_1 z_1|_2 \ldots |_m z_m)$ is the **secondary** constituent of the rule. The notion of a distinguished child is defined as in the case of LIG, i.e., a category is the distinguished child of its parent if it corresponds to the primary constituent of the rule used. As before, the distinguished descendant is the reflexive, transitive closure of the distinguished child relation.

In discussing CCG we use the notational conventions that the variables $|$ and c (when used with or without primes and subscripts) range over the forward and backward slashes and categories, respectively. We use x, y, z for meta-variables; α, β for strings of directional categories (i.e., a string of the form $|_1 c_1 |_2 \ldots |_n c_n$ from some $n \geq 0$); and A, B, C for atomic categories (i.e., members of V_N).

Derivations in a CCG, $G = (V_T, V_N, S, f, R)$, involve the use of the combinatory rules in R. Let $\underset{G}{\Longrightarrow}$ be defined as follows, where Υ_1 and Υ_2 are strings of categories and terminal symbols.

- If $c_1 c_2 \rightarrow c$ is an instance of a rule in R, then $\Upsilon_1 c \Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 c_1 c_2 \Upsilon_2$.
- If $c \in f(a)$ for some $a \in V_T$ and c is a category, then $\Upsilon_1 c \Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 a \Upsilon_2$.

The string languages generated by a CCG, G, $L(G) = \{ w \mid S \underset{G}{\overset{*}{\Longrightarrow}} w \mid w \in V_T^* \}$.

Example 4.1
The following CCG generates $\{ wcw \mid w \in \{a, b\}^+ \}$. Let $G = (\{a, b, c\}, \{S, T, A, B\}, S, f, R)$ where

$$f(a) = \{A, T\backslash A/T, T\backslash A\} \qquad f(b) = \{B, T\backslash B/T, T\backslash B\} \qquad f(c) = \{S/T\}$$

The set of rules R includes the following three rules.

$$y \quad (x\backslash y) \rightarrow x \qquad (x/y) \quad (y\backslash z_1/z_2) \rightarrow (y\backslash z_1/z_2) \qquad (x/y) \quad (y\backslash z_1) \rightarrow (y\backslash z_1)$$

In each of these rules, the target of the category matched with x must be S.[7] Figure 7 shows a derivation of the string *abbcabb*.

We find it convenient to represent categories in a minimally parenthesized form (i.e., without parentheses unless they are needed to override the left associativity of the slashes), where minimally parenthesized form is defined as follows.

7 Following Steedman (1985), we allow certain very limited restrictions on the substitutions of variables in the combinatory rules. A discussion on the use of such restrictions is given in Vijay-Shanker and Weir (in press). However, we have not included this in the formal definition since it does not have a significant impact on the algorithm presented.

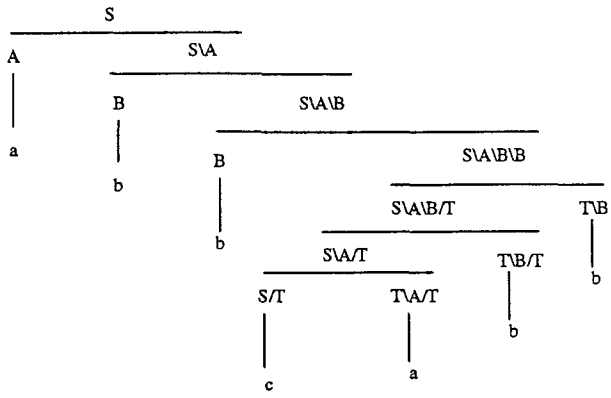

Figure 7
CCG example derivation tree.

Definition 4.4

- A is the minimally parenthesized form of A where $A \in V_N$.
- If $c_1, \ldots, c_n$ are the minimally parenthesized forms of categories $c'_1, \ldots, c'_n$ respectively, then $(A|_1c_1|_2 \ldots |_nc_n)$ is the minimally parenthesized form of $((\cdots(A|_1c'_1)|_2\cdots)|_nc'_n)$.

A category c is in minimally parenthesized form if c is the minimally parenthesized form of itself.

Definition 4.5
Let a category $c = A|_1c_1|_2 \ldots |_nc_n$ be in minimally parenthesized such that $n \geq 0$, $A \in V_N$, and $c_1, \ldots, c_n$ are minimally parenthesized categories.

- The **target** category of $c = A|_1c_1|_2 \ldots |_nc_n$ denoted by $tar(c)$ is A.
- The **arity** of $c = A|_1c_1|_2 \ldots |_nc_n$, denoted as $arity(c)$, is n.
- The argument categories of $c = A|_1c_1|_2 \ldots |_nc_n$ denoted by $args(c) = \{ c_i \mid 1 \leq i \leq n \}$.

4.1 CCG and LIG

Before showing how the general parsing scheme illustrated by the LIG recognition algorithm can be instantiated as a recognition algorithm for CCG, we show that CCG and LIG are very closely related. The details of the examination of the relationship between CCG and LIG may be found in Weir and Joshi (1988) and Weir (1988).

A minimally parenthesized category $(A|_1c_1|_2 \ldots |_nc_n)$ can be viewed as the atomic category, A, associated with a stack of directional argument categories, $|_1c_1|_2 \ldots |_nc_n$. The rule

$$(x/y) \quad (y|_1z_1|_2 \ldots |_mz_m) \rightarrow (x|_1z_1|_2 \ldots |_mz_m)$$

has as an instance $(A_p|'_1c'_1 \ldots |'_nc'_n/A_s) \quad (A_s|_1c_1|_2 \ldots |_mc_m) \rightarrow (A_p|'_1c'_1 \ldots |'_nc'_n|_1c_1|_2 \ldots |_mc_m)$ as well as $(A_p|'_1c'_1 \ldots |'_nc'_n/(A_s|'c')) \quad (A_s|'c'|_1c_1|_2 \ldots |_mc_m) \rightarrow (A_p|'_1c'_1 \ldots |'_nc'_n|_1c_1|_2 \ldots |_mc_m)$

as an instance. Thus x matches the category $(A_p|'_1c'_1 \ldots |'_nc'_n)$, y matches an atomic category A_s in the first example and a nonatomic category $(A_s|'c')$ in the second, and each z_i matches c_i for $1 \leq i \leq m$. A derivation involving the second instance (viewed bottom-up) can be seen as popping the top directional argument $/(A_s|'c')$ from the primary category and pushing the m directional arguments $|_1c_1|_2 \ldots |_mc_m$. Thus, each instance of the combinatory rule appears to closely resemble a LIG production. For example, in case of the second instance we have

$$A_p\left(\cdot\cdot|_1c_1|_2 \ldots |_mc_m\right) \rightarrow A_p\left(\cdot\cdot/(A_s|'c')\right) A_s\left(|'c'|_1c_1|_2 \ldots |_mc_m\right).$$

We now show that, like the set of stack symbols of a LIG, the set of directional argument categories that we need to be concerned with is finite.

Definition 4.6
Let c be a **useful** category with respect to a grammar G if and only if $c \stackrel{*}{\Longrightarrow} w$ for some $w \in V_T^*$. The set of **argument** categories, $args(G)$ of a CCG, $G = (V_T, V_N, S, f, R)$, is defined as $args(G) = \bigcup_{c \in f(a)} args(c)$.

Observation 4.1
If c is a useful category then $args(c) \subseteq args(G)$, a finite set determined by the grammar, G.

This observation can be shown by an induction on the length of the derivation of some string from c. The base case corresponds to a lexical assignment and hence trivially $args(c) \subseteq args(G)$. The inductive step corresponds to the use of a combination using a rule of the form

$$(x/y) \quad (y|_1z_1|_2 \ldots |_mz_m) \rightarrow (x|_1z_1|_2 \ldots |_mz_m)$$

or

$$(y|_1z_1|_2 \ldots |_mz_m) \quad (x\backslash y) \rightarrow (x|_1z_1|_2 \ldots |_mz_m)$$

By inductive hypothesis, any useful category matching either $(x/y), (x\backslash y)$ or $(y|_1z_1|_2 \ldots |_mz_m)$ must take its arguments from $args(G)$ (a finite set) and therefore the resulting useful category also shares this property.

The above property makes it possible to adapt the LIG algorithm for CCG. Note that in the CKY-style CCG recognition we only need to record the derivations from useful categories. From Observation 4.1 it follows that the lexical category assignment, f, determines the number of "stack" symbols we need to be concerned with. Therefore, only one of the variables (x) in a combinatory rule is essential in the sense that the number of categories that it can usefully match is not bound by the grammar. Therefore, it would be possible to map each combinatory rule to an equivalent finite set of instances in which ground categories (from $args(G)$) were substituted for all variables other than x; i.e., $y, z_1, \ldots z_m$ in the combinatory rule above. This would result in a grammar that was a slight notational variant of a LIG where the CCG variable x and the LIG notation $\cdot\cdot$ perform similar roles. However, for the purpose of constructing a recognition algorithm it is both unnecessary and undesirable to expand the number of rules in this way. We adapt the LIG algorithm so that it, in effect, constructs appropriate instances of the combinatory rules as needed during the recognition process.

4.2 Recognition of CCG

The first step in modifying the LIG algorithm is to define the constants MSL and MTL for the case of CCG. Let $G = (V_T, V_N, S, f, R)$ be a CCG. These definitions follow immediately from the similarities between CCG combinatory rules and LIG productions.

Observation 4.2
If we were to express a combinatory rule

$$(x/y) \quad (y|_1 z_1 \dots |_m z_m) \rightarrow (x|_1 z_1 \dots |_m z_m)$$

in terms of LIG production

$$A(\cdot\cdot \gamma_1 \dots \gamma_m) \rightarrow A_p(\cdot\cdot \gamma_p) A_s(\alpha_s)$$

then we have the following correspondences:

- γ_p with $/y$.
- γ_i with $= |_i z_i$ for $1 \leq i \leq m$, i.e., $\gamma_1 \dots \gamma_m$ with $|_1 z_1 \dots |_m z_m$.
- $A = A_p$.
- $A_s(\alpha_s)$ with $y|_1 z_1 \cdots |_m z_m$.

Given such a direct correspondence between combinatory rules and LIG productions, we will define the following constants to be used in the the CCG algorithm with minimal explanation.

- MTL is the maximum arity of a lexical category. Thus, $\mathrm{MTL} = max\{ arity(\mathrm{c}) \mid \mathrm{c} \in \mathrm{f(a)}, \mathrm{a} \in \mathrm{V_T} \}$.
- MSL should be the maximum arity of a useful category that can match the secondary category of a rule. Note that a category matching $(y|_1 z_1|_2 \dots |_m z_m)$ will have an arity that is the sum of m and the arity of the category matching y. Furthermore, note that since y is an argument of the primary category it must be bound to a member of $args(G)$. Thus, $\mathrm{MSL} = max\{ \mathrm{m} \mid (\mathrm{y}|_1 \mathrm{z}_1|_2 \dots |_\mathrm{m} \mathrm{z}_\mathrm{m}) \}$ is the secondary category of a rule in $R + max\{ arity(c) \mid c \in args(G) \}$.
- Note that in the case of CCG, MCL need not be defined independently of MSL.
- As before, we define TTC as $\mathrm{TTC} = max\{ \mathrm{MSL}, \mathrm{MTL} \}$.

Since directional categories play the same role that stack symbols have in LIG, we revise the notions of length *top*( ) and *rest*( ) as follows. We say that the string of directional arguments categories $|_1 c_1|_2 \dots |_n c_n$ has a length n, i.e., $len(|_1 c_1|_2 \dots |_n c_n) = n$. Note that $arity((A|_1 c_1|_2 \dots |_n c_n)) = len(|_1 c_1|_2 \dots |_n c_n) = n$. We define $top((|_1 c_1|_2 \dots |_n c_n)) = |_n c_n$ and $rest((|_1 c_1|_2 \dots |_n c_n)) = |_1 c_1|_2 \dots |_{n-1} c_{n-1}$. Additionally, $top(\epsilon) = rest(\epsilon) = \epsilon$.

4.2.1 Terminators in CCG. We can define a k-terminator in essentially the same way as in the case for LIG. Note that a category shares its target category with all of its distinguished descendants.

Definition 4.7
Suppose that we have the following derivation:

$$\begin{array}{rcl} A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c & \stackrel{*}{\Longrightarrow} & u \quad A\beta|_1c_1\ldots|_{k-1}c_{k-1}|_kc_k\ldots|_mc_m \quad w \\ & \Longrightarrow & u \quad A\beta/c_p \quad c_p|_1c_1\ldots|_kc_k\ldots|_mc_m \quad w \\ & \stackrel{*}{\Longrightarrow} & uvw \end{array}$$

or similarly

$$\begin{array}{rcl} A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c & \stackrel{*}{\Longrightarrow} & u \quad A\beta|_1c_1\ldots|_{k-1}c_{k-1}|_kc_k\ldots|_mc_m \quad w \\ & \Longrightarrow & u \quad c_p|_1c_1\ldots|_kc_k\ldots|_mc_m \quad A\beta\backslash c_p \quad w \\ & \stackrel{*}{\Longrightarrow} & uvw \end{array}$$

where the following conditions hold

- β is a string of direction categories, i.e., $\beta \in (\{/,\backslash\}args(G))^*$.
- $k-1 \geq 1$,
- $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|_kc_k\ldots|_mc_m$ and $A\beta/c$ are distinguished descendents of $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$
- any distinguished descendent between $A\beta/c_p$ and $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$ can be expressed in the form $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|\alpha$ where $len(\alpha) \geq 1$

We say that $A\beta/c_p$ is the $len\left(|_1c_1\ldots|_{k-1}c_{k-1}|c\right)$-terminator of $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$.

Note that c_p need not be atomic. Hence if we write the secondary category as $c_p|_1c_1\ldots|_mc_m$ we are not necessarily expressing it in minimal parenthesis form.

4.2.2 Anatomy of a CCG Entry. In the CCG algorithm we will use entries that have a form similar to that of the entries in the LIG algorithm. The choices we make are based on Observation 4.2. For a derivation $A|_1c_1\ldots|_jc_j \stackrel{*}{\Longrightarrow} a_1\ldots a_{i+d-1}$ (where the input is $a_1\ldots a_n$), we will have an entry in $P[i,d]$ with a head $\langle A,|_jc_j\rangle$, where $A \in V_N$, $|_j \in \{\backslash,/\}$, and $c_j \in args(G)$.

First consider the case when a terminator-type entry is used. The terminator-type entry is applicable when $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$ has a k-terminator, say $A\beta|_tc_t$ where $len\left(\beta|_tc_t\right) \geq \text{TTC}$. As before we say that in such a case $A\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$ satisfies the TC-property. Assuming the terminator derives the substring $a_t\ldots a_{t+d_t-1}$, we can use the terminator-pointer $\left(\langle|c_t\rangle, [t,d_t]\right)$ and a middle $|_1c_1\ldots|_{k-1}c_{k-1}$. Notice that since the target of the category $A|\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$ as well as the target of its terminator is A and since A is already noted in the head, it is not recorded in the terminator-pointer.

For entries that are not terminator-pointer, the entire category is noted in the entry. Such an entry has the form $\left(\langle A,|_jc_j\rangle\left(|_1c_1\ldots|_{j-1}c_{j-1},nil\right)\right)$ assuming that $j \geq 1$. However, it is possible that $j=0$. In this case the category being represented is A, and the entry will be written as $\left(\langle A,\epsilon\rangle\left(\epsilon,nil\right)\right)$. In general, we use the non–terminator-type entry for recording a derivation from $A\alpha$ when it has no terminator or when the terminator, say $A\beta|_tc_t$ (rewriting α as $\beta|_1c_1\ldots|_{k-1}c_{k-1}|c$) is such that $len\left(\beta|_tc_t\right) \leq \text{TTC}$; i.e., when the category $A\alpha$ does not satisfy the TC-property.

4.2.3 CCG Algorithm. It is straightforward to derive the rules for the CCG recognition algorithm from those used in LIG algorithm. Using Observation 4.2, we can now give the rules for the CCG algorithm with no explanation.

Rule 1.C

$$\frac{A\alpha \in f(a) \quad a = a_i \quad 1 \leq i \leq n}{\left(\langle A_p, top\,(\alpha)\rangle \; (rest\,(\alpha), nil)\right) \in P[i, 1]}$$

Assume the combinatory rule $(x/y) \quad (y|_1 z_1 \ldots |_m z_m) \rightarrow (x|_1 z_1 \ldots |_m z_m)$.

When $m = 0$ **and** $tp_p = nil$

Rule 2.ps.C

$$\frac{\left(\langle A_p, /c_p\rangle \; (\beta_p, nil)\right) \in P[i, d_p] \qquad \begin{matrix} A_s\alpha_s = c_p \\ \left(\langle A_s, top\,(\alpha_s)\rangle \; (rest\,(\alpha_s), nil)\right) \in P[i + d_p, d - d_p] \end{matrix}}{\left(\langle A_p, top\,(\beta_p)\rangle \; (rest\,(\beta_p), nil)\right) \in P[i, d]}$$

When $m = 0$ **and** $tp_p \neq nil$

Rule 3.ps.C

$$\frac{\begin{matrix} tp_p = \left(\langle |_t c_t\rangle, [t, d_t]\right) \\ k \geq 2 \\ \left(\langle A_p, /c_p\rangle \; \left(|_1 c_1 \ldots |_k c_k, tp_p\right)\right) \in P[i, d_p] \end{matrix} \qquad \begin{matrix} A_s\alpha_s = c_p \\ \left(\langle A_s, top\,(\alpha_s)\rangle \; (rest\,(\alpha_s), nil)\right) \in P[i + d_p, d - d_p] \end{matrix}}{\left(\langle A_p, |_k c_k\rangle \; \left(|_1 c_1 \ldots |_{k-1} c_{k-1}, tp_p\right)\right) \in P[i, d]}$$

Rule 4.ps.C

$$\frac{\begin{matrix} \left(\langle A_p, /c_p\rangle \; \left(|_1 c_1, \left(\langle |_t c_t\rangle, [t, d_t]\right)\right)\right) \\ \in P[i, d_p] \end{matrix} \qquad \begin{matrix} A_s\alpha_s = c_p \\ \left(\langle A_s, top\,(\alpha_s)\rangle \; (rest\,(\alpha_s), nil)\right) \\ \in P[i + d_p, d - d_p] \end{matrix} \qquad \begin{matrix} \left(\langle A_p, |_t c_t\rangle \; (\beta_t, nil)\right) \\ \in P[t, d_t] \end{matrix}}{\left(\langle A_p, |_1 c_1\rangle \; (\beta_t, nil)\right) \in P[i, d]}$$

Rule 5.ps.C

$$\frac{\begin{matrix} \left(\langle A_p, /c_p\rangle \; \left(|_1 c_1, \left(\langle |_t c_t\rangle, [t, d_t]\right)\right)\right) \\ \in P[i, d_p] \end{matrix} \qquad \begin{matrix} A_s\alpha_s = c_p \\ \left(\langle A_s, top\,(\alpha_s)\rangle \; (rest\,(\alpha_s), nil)\right) \\ \in P[i + d_p, d - d_p] \end{matrix} \qquad \begin{matrix} tp_t = \left(\langle |_r c_r\rangle, [r, d_r]\right) \\ \left(\langle A_p, |_t c_t\rangle \; (\beta_r, tp_t)\right) \\ \in P[t, d_t] \end{matrix}}{\left(\langle A_p, |_1 c_1\rangle \; (\beta_r, tp_t)\right) \in P[i, d]}$$

When $m = 1$

Rule 6.ps.C

$$\frac{\left(\langle A_p, /c_p\rangle \; (\beta_p, nil)\right) \in P[i, d_p] \qquad \begin{matrix} A_s\alpha_s = c_p|_1 c_1 \\ \left(\langle A_s, top\,(\alpha_s)\rangle \; (rest\,(\alpha_s), nil)\right) \in P[i + d_p, d - d_p] \end{matrix}}{\left(\langle A_p, |_1 c_1\rangle \; (\beta_p, nil)\right) \in P[i, d]}$$

Rule 7.ps.C

$$\frac{\left(\langle A_p, /c_p\rangle\ (\beta_p, (\langle|_t c_t\rangle, [t, d_t]))\right) \in P[i, d_p] \qquad \overset{A_s\alpha_s = c_p|_1 c_1}{\left(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)\right) \in P[i + d_p, d - d_p]}}{\left(\langle A_p, |_1 c_1\rangle\ (\beta_p, (\langle|_t c_t\rangle, [t, d_t]))\right) \in P[i, d]}$$

When $m \geq 2$ **and** $tp_p \neq nil$

Rule 8.ps.C

$$\frac{\begin{array}{c} tp_p = (\langle|_t c_t\rangle, [t, d_t]) \\ \left(\langle A_p, /c_p\rangle\ (\beta_p, tp_p)\right) \in P[i, d_p] \end{array} \qquad \overset{A_s\alpha_s = c_p|_1 c_1 \ldots |_m c_m}{\left(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)\right) \in P[i + d_p, d - d_p]}}{\left(\langle A_p, |_m c_m\rangle\ (|_1 c_1 \ldots |_{m-1} c_{m-1}, (\langle/c_p\rangle, [i, d_p]))\right) \in P[i, d]}$$

When $m \geq 2$ **and** $tp_p = nil$

Rule 9.ps.C

$$\frac{\begin{array}{c} len(\beta_p/c_p) < \text{TTC} \\ \left(\langle A_p, /c_p\rangle\ (\beta_p, nil)\right) \in P[i, d_p] \end{array} \qquad \overset{A_s\alpha_s = c_p|_1 c_1 \ldots |_m c_m}{\left(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)\right) \in P[i + d_p, d - d_p]}}{\left(\langle A_p, |_m c_m\rangle\ (\beta_p|_1 c_1 \ldots |_{m-1} c_{m-1}, nil)\right) \in P[i, d]}$$

Rule 10.ps.C

$$\frac{\begin{array}{c} len(\beta_p/c_p) \geq \text{TTC} \\ \left(\langle A_p, /c_p\rangle\ (\beta_p, nil)\right) \in P[i, d_p] \end{array} \qquad \overset{A_s\alpha_s = c_p|_1 c_1 \ldots |_m c_m}{\left(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil)\right) \in P[i + d_p, d - d_p]}}{\left(\langle A_p, |_m c_m\rangle\ (|_1 c_1 \ldots |_{m-1} c_{m-1}, (\langle/c_p\rangle, [i, d_p]))\right) \in P[i, d]}$$

Proposition 4.1
The CCG recognition algorithm can be seen to establish the following.

- $\left(\langle A_p, |c\rangle\ (\beta, (\langle|_t c_t\rangle, [t, d_t]))\right) \in P[i, d]$ if and only if there is some α such that $A\alpha\beta|c \overset{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ and the $(len(\beta) + 1)$-terminator $(A\alpha|_t c_t)$ of $A\alpha\beta|c$ derives the string $a_t \ldots a_{t+d_t-1}$ and $len(\alpha|_t c_t) \geq \text{TTC}$.
- $\left(\langle A_p, top(\alpha)\rangle, (rest(\alpha), nil,)\right) \in P[i, d]$ if and only if $A\alpha \overset{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ and either $A\alpha$ has no terminator or its terminator, say $A\alpha'$ is such that $len(\alpha') \leq \text{TTC}$.

5. TAG Recognition

We begin this section by first considering how to extend our algorithm for LIG to handle unary productions. This will be needed to show we can instantiate our scheme to give a recognition algorithm for TAG.

5.1 Handling Unary Productions and Epsilon Productions

We will now show how the LIG algorithm given earlier can be extended to consider **unary** productions of the form $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\gamma_p)$ as well as ϵ productions of

the form: $A(\alpha) \rightarrow \epsilon$. However, we will now assume that $m \leq 2$ in productions of the form $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow \Upsilon_1 A_p(\cdot\cdot\gamma_p)\Upsilon_2$. Thus, henceforth MCL ≤ 2. Note that this refers to both unary and binary productions. This additional restriction does not change the generative power. We have introduced these restrictions in order to reduce the number of cases we have to consider and also because we can restrict our attention to the productions that are used in the TAG to LIG construction.

Consider the processing of a *binary* production $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\gamma_p) A_s(\alpha_s)$. Since CKY-style parsers work bottom-up, we check to see if the primary and secondary categories derive adjacent strings (say $a_i \ldots a_{i+d_p-1}$ and $a_{i+d_p} \ldots a_{i+d_p+d_s}$, respectively) and then we store an encoding for the new object that results from the combination. The processing of unary productions is similar except that we do not have to consider a secondary constituent. The rules that express the processing of such productions will be very similar to those for the binary productions. For example, consider Rule 2.ps.L for the binary production $A(\cdot\cdot) \rightarrow A_p(\cdot\cdot\gamma_p) A_s(\alpha_s)$.

Rule 2.ps.L

$$\frac{(\langle A_p, \gamma_p\rangle(\beta_p, nil)) \in P[i, d_p] \quad (\langle A_s, top(\alpha_s)\rangle(rest(\alpha_s), nil)) \in P[i + d_p, d - d_p]}{(\langle A, top(\beta_p)\rangle(rest(\beta_p), nil)) \in P[i, d]}$$

Given a unary production $A(\cdot\cdot) \rightarrow A_p(\cdot\cdot\gamma_p)$ we have the Rule 2.u.L (where u stands for unary).

Rule 2.u.L

$$\frac{(\langle A_p, \gamma_p\rangle(\beta_p, nil)) \in P[i, d_p]}{(\langle A, top(\beta_p)\rangle(rest(\beta_p), nil)) \in P[i, d]}$$

In addition, with the introduction of ϵ productions, we have to consider derivations of strings of length $d = 0$. We shall assume that if $A(\alpha) \stackrel{*}{\Longrightarrow} \epsilon$ then an encoding of $A(\alpha)$ will be stored in $P[i, 0]$ (for all i). We must also consider the possibility that the primary constituent or the secondary constituent derive the empty string, i.e., $d_p = 0$ or $d_s = 0$. Processing of such cases becomes similar to that of unary productions.

To indicate the additional processing required due to the introduction of unary productions and the possibility of the derivation of the empty string, let us consider Rule 8.ps.L. "Use Rule 8.ps.L" can be paraphrased as follows.

> If there exists a production $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\gamma_p) A_s(\alpha_s)$ where $m \geq 2$, $e_1 = (\langle A_p, \gamma_p\rangle(\beta_p, \langle A_t, \gamma_t\rangle))$ belongs to $P[i, d][t, d_t]$ and $e_2 = (\langle A_s, top(\alpha_s)\rangle(rest(\alpha_s), nil))$ belongs to $P[i + d_p, d - d_p][0, 0]$ then add $e_3 = (\langle A, \gamma_m\rangle(\gamma_1 \ldots \gamma_{m-1}, \langle A_p, \gamma_p\rangle))$ to $P[i, d][i, d_p]$ if e_3 is not already present in this array element.

If we allow ϵ productions it is possible that $d_s = d - d_p = 0$. Consider the case where we have $A_s(\alpha_s) \stackrel{*}{\Longrightarrow} \epsilon$. That is, we expect the entry e_2 to be present in $P[i + d, 0][0, 0]$. This means that the resulting entry e_3 must be added to $P[i, d][i, d]$ since we now have $d_p = d$. Note that the addition of $e_3 = (\langle A, \gamma_m\rangle(\gamma_1 \ldots \gamma_{m-1}, \langle A_p, \gamma_p\rangle))$ (that encodes the derivation from $A(\beta\gamma_1 \ldots \gamma_p)$ for some β) can result in more entries being added to the same array element $P[i, d][i, d]$ (for instance, when we have the production $B(\cdot\cdot\gamma'_1 \ldots \gamma'_k) \rightarrow A(\cdot\cdot\gamma_m)$). This is similar to the *prediction* phase in Earley's algorithm and the state construction in LR parsing. Based on this analogy, we will define our

notion of **closure**. Closure (e, i, d, t, d_t) will add entries to $P[i,d][t,d_t]$ or $P[i,d][i,d]$ that result from the inclusion of the entry $P[i,d][t,d_t]$ by considering unary productions (or binary productions when the primary or secondary constituent derives the empty string). Before we define Closure( ) we note that for each occurrence in the algorithm of "use Rule X" is replaced by "use closure of Rule X." For example, "Use closure of Rule 8.ps.L" stands for

> If we have the production $A(\cdot\cdot\gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\gamma_p) A_s(\alpha_s)$ where $m \geq 2$,
> $e_1 = (\langle A_p, \gamma_p\rangle (\beta_p, A_t, \gamma_t))$ belongs to $P[i,d][t,d_t]$ and
> $e_2 = (\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil))$ belongs to $P[i+d_p, d-d_p][0,0]$ and
> $e_3 = (\langle A, \gamma_m\rangle (\gamma_1 \ldots \gamma_{m-1}, A_p, \gamma_p))$ does not belong to $P[i,d][i,d_p]$ then
> add e_3 to $P[i,d][i,d_p]$ and then invoke Closure (e_3, i, d, i, d_p).

Closure is defined as follows:

Closure (e, i_1, d_1, t, d_t)
begin
use closure of Rule 2.ps.L, 6.ps.L, 7.ps.L, 8.ps.L, 9.ps.L, 10.ps.L
with $d = d_p$ and the entry e as the primary constituent in the antecedent.
use closure of Rule 2.sp.L, 6.sp.L, 7.sp.L, 8.sp.L, 9.sp.L, 10.sp.L
with $d_s = d$ and the entry e as the secondary constituent in the antecedent.
use closure of Rule 2.u.L, 6.u.L, 7.u.L, 8.u.L, 9.u.L, 10.u.L
with $d = d_p$ and the entry e as the primary constituent in the antecedent.
end.

Note Rule 3 does not apply since we have to assume MCL ≤ 2 (hence any terminator is a 2-terminator and the length of the middle in a terminator-type entry is always one). We have not included Rule 4 and Rule 5 while computing the closure. These correspond directly to the completor step in Earley's algorithm and to the popping of stack elements and hence are not considered a part of the closure. They have to be applied later in the control structure.

We will now consider the effect of including unary rules on the control structure of the algorithm. Let $\langle\langle i_1, d_1\rangle, \langle i_2, d_2\rangle\rangle \prec \langle\langle i_3, d_3\rangle, \langle i_4, d_4\rangle\rangle$ if and only if (1) $\langle i_1, d_1\rangle < \langle i_3, d_3\rangle$ or (2) $\langle i_1, d_1\rangle = \langle i_3, d_3\rangle$ and $\langle i_2, d_2\rangle < \langle i_4, d_4\rangle$. The simplicity of the loop structure in the algorithms seen thus far stems from the fact that for any parsing rule if the entry in the consequent is to be added to $P[i_3, d_3][i_4, d_4]$ based on the existence of an antecedent entry in $P[i_1, d_1][i_2, d_2]$, then $\langle\langle i_1, d_1\rangle, \langle i_2, d_2\rangle\rangle \prec \langle\langle i_3, d_3\rangle, \langle i_4, d_4\rangle\rangle$. This no longer holds when we consider Rule 5.u.L or Rule 5.ps.L when the secondary constituent derives the empty string. Consider the following derivation (and the presence of the productions assumed) for a sufficiently long β:

$$A(\beta\alpha) \stackrel{1}{\Longrightarrow} A_1(\beta\gamma_1) \stackrel{1}{\Longrightarrow} A_2(\beta\gamma_1\gamma_2) \stackrel{1}{\Longrightarrow} A_3(\beta\gamma) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$$

Consider the addition of an entry e_3 to $P[i,d][t,d_t]$ (for some $\langle t, d_t\rangle$) to record the derivation from $A_3(\beta\gamma)$. Closure (e_3, i, d, t, d_t) is invoked, resulting in the addition of e_2 (corresponding to $A_2(\beta\gamma_1\gamma_2)$) to $P[i,d][i,d]$. From Rule 5.u.L and the presence of entry e_2 and e_3 we would add e_1 (corresponding to $A_1(\beta\gamma_1)$ to $P[i,d][t,d_t]$). This could result in the need to add more entries to $P[i,d][i,d]$, which in turn could cause new entries being added back to $P[i,d][t,d_t]$, and so on. Thus we have a situation where

```
                    ⋮
initialization phase
                    ⋮
for loops for d, i, d' as before
    begin
        consider closure of Rules in Rule set I
        for d_t := d' − 1 to 1 do
            for t := i to i + d' − d_t do
                repeat
                    consider closure of Rules in Rule set II
                        for d_r := d_t − 1 to 1 do
                            for r := t to t + d_t − d_r do
                                consider closure of Rules 5.ps.L and Rule 5.u.L
                until no new entries are added to P[i,d][t,d_t]
```

Figure 8
Control structure with unary productions.

an antecedent entry in $P[i,d][t,d_t]$ ($\langle t,d_t\rangle < \langle i,d\rangle$) causes an entry to be added to $P[i,d][i,d]$, which, acting as an antecedent entry, causes a new entry to be added to $P[i,d][t,d_t]$.

A simple strategy to take care of this situation would be to add another loop within the t loop (as shown in the partial control structure given in Figure 8) that is repeated until no new entries are added to $P[i,d][t,d_t]$. It is straightforward to prove the correctness of the algorithm with this additional loop and also that the asymptotic complexity remains the same. The latter is the case because only a bounded number of entries can belong to $P[i,d][t,d_t]$ for any fixed value of i, d, t, d_t, and hence the repeat loop can be iterated only a bounded number of times (as determined by the grammar). In the partially specified control structure given in Figure 8, we have not considered the *sp* rules. Also we only consider the changes that need to be made to Algorithm 1; the changes to Algorithm 2 can be made in a similar fashion. Finally, for purposes of abbreviation, we have grouped Rules 2.ps.L, 6.ps.L, 9.ps.L, and 10.ps.L together and called it the Rule set I, and Rules 3.ps.L, 4.ps.L, 7.ps.L, and 8.ps.L the Rule set II.

The repeat loop shown in Figure 8 is not needed in some situations. Consider the derivation and the sequence of addition of entries, e_3, e_2, e_1, as discussed above. Viewing this derivation as a bottom-up recognizer would, we have a "prediction" from entry e_3 followed by a "completion" that results in the entry e_1. In this case the two entries both encode objects with the same stack length. We generalize this situation and call such derivations **auxiliary** derivations (named after auxiliary trees in TAG).

$$A(\beta\gamma_1) \stackrel{1}{\Longrightarrow} \Upsilon_1 A_1(\beta\gamma_1\gamma_2)\Upsilon_2 \stackrel{*}{\Longrightarrow} \Upsilon_1 u A_t(\beta\gamma_t) w \Upsilon_2 \stackrel{*}{\Longrightarrow} u_1 u A_t(\beta\gamma_t) w w_1$$

where $A_t(\beta\gamma_t)$ is the 2-terminator of $A_1(\beta\gamma_1\gamma_2)$. We will say that this auxiliary derivation spans at least one terminal if $len(u_1 u w w_1) \geq 1$. Notice that if for a particular grammar every auxiliary derivation spans at least one terminal, then the extra repeat loop added becomes unnecessary. This is because now, with this assumption, for every parsing rule if the entry in the consequent is to be added to $P[i_3,d_3][i_4,d_4]$ based on the existence of an antecedent entry in $P[i_1,d_1][i_2,d_2]$ then $\langle\langle i_1,d_1\rangle,\langle i_2,d_2\rangle\rangle \prec \langle\langle i_3,d_3\rangle,\langle i_4,d_4\rangle\rangle$.

We end this section by noting that in the case of a lexicalized TAG, we can verify that every auxiliary derivation spans at least one terminal, and hence in the TAG algorithm we do not have to include this additional repeat loop.

5.2 Tree Adjoining Grammars

Tree Adjoining Grammars (TAG) is a tree generating formalism introduced by Joshi, Levy, and Takahashi (1975). A TAG is defined by a finite set of trees composed by means of the operation of tree adjunction.

Definition 5.1
A TAG, G, is denoted by (V_N, V_T, S, I, A) where
V_N is a finite set of nonterminals symbols,
V_T is a finite set of terminal symbols,
$S \in V_N$ is the start symbol,
I is a finite set of initial trees,
A is a finite set of auxiliary trees.

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

We will call a node in an elementary tree an **elementary node**. We can give a unique name to each elementary node by using an **elementary node address**. An elementary node address is a pair composed of the name of the elementary tree to which the node belongs and the address of the node within that tree. We will assume the standard addressing scheme where the root node has an address ϵ. If a node addressed μ has k children then the k children (in left to right order) have addresses $\mu \cdot 1, \ldots, \mu \cdot k$. Thus, if $\mathcal{N}$ is the set of natural numbers then $\mu \in \mathcal{N}^*$. In this section we will use μ to refer to addresses and η to refer to elementary node addresses. In general, we can write $\eta = \langle \gamma, \mu \rangle$ where γ is an elementary tree and $\mu \in \text{Domain}(\gamma)$. We will use $\text{Domain}(\gamma)$ for the set of addresses of the nodes in γ.

Definition 5.2
Let γ be a tree with internal node labeled by a nonterminal A. Let β be an auxiliary tree with root and foot node labeled by the same nonterminal A. The tree, γ', that results from the **adjunction** of β at the node in γ labeled A (as shown in Figure 9) is formed by removing the subtree of γ rooted at this node, inserting β in its place, and substituting it at the foot node of β.

Each elementary node is associated with a **selective adjoining** (SA) constraint that determines the set of auxiliary trees that can be adjoined at that node. In addition, when adjunction is mandatory at a node it is said to have an **obligatory adjoining** (OA) constraint. Figure 9 shows how constraints are associated with nodes in trees derived from adjunctions. Whether β can be adjoined at the node (labeled by A) in γ is determined by c, the SA constraint of the node. In γ' the nodes contributed by β

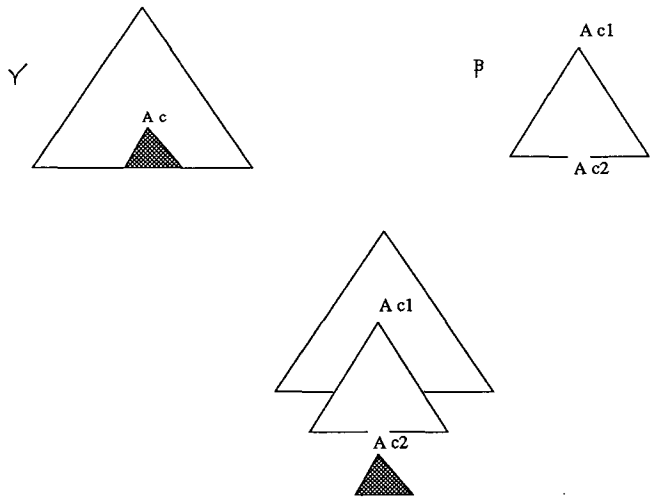

Figure 9
The operation of adjoining.

have the same constraints as those associated with the corresponding nodes in β. The remaining nodes in γ' have the constraints of the corresponding nodes in γ.

Given $\mu \in \text{Domain}(\gamma)$, by $\text{LABEL}\langle\gamma, \mu\rangle$ we refer to the label of the node addressed μ in γ. If the tree in question is clear from context, we will simply use $\text{LABEL}\langle\mu\rangle$. Similarly, we will use $\text{SA}\langle\gamma, \mu\rangle$ (or $\text{SA}\langle\mu\rangle$) and $\text{OA}\langle\gamma, \mu\rangle$ (or $\text{OA}\langle\mu\rangle$) to refer to the SA and OA constraints of a node addressed μ in a tree γ. Finally, we will use $\text{ft}(\beta)$ to refer to the address of the foot node of an auxiliary tree β.

To be precise, we define the adjunction of β at a node in γ with address μ as follows. This operation is defined when β is included in the SA constraints of node addressed μ in γ. If the operation is defined, we will use $\text{ADJ}(\gamma, \mu, \beta)$ to refer to the tree that results. Let $\gamma' = \text{ADJ}(\gamma, \mu, \beta)$. Then the nodes in γ' and their labels and adjoining constraints are defined as follows.

- $\text{Domain}(\gamma') = \{\mu_1 \mid \mu_1 \in \text{Domain}(\gamma), \mu_1 \neq \mu \cdot \mu_2, \text{ for some } \mu_2 \in \mathcal{N}^*\} \cup \{\mu \cdot \mu_1 \mid \mu_1 \in \text{Domain}(\beta)\} \cup \{\mu \cdot \text{ft}(\beta) \cdot \mu_1 \mid \mu \cdot \mu_1 \in \text{Domain}(\gamma), \text{ and } \mu_1 \neq \epsilon\}$
- When $\mu_1 \in \text{Domain}(\gamma)$ such that $\mu_1 \neq \mu \cdot \mu_1$ for some $\mu_1 \in \mathcal{N}^*\}$, i.e., the node in γ with address μ_1 is not equal to or dominated by the node addressed μ in γ:
  - $\text{LABEL}\langle\gamma', \mu_1\rangle = \text{LABEL}\langle\gamma, \mu_1\rangle$,
  - $\text{SA}\langle\gamma', \mu_1\rangle = \text{SA}\langle\gamma, \mu_1\rangle$,
  - $\text{OA}\langle\gamma', \mu_1\rangle = \text{OA}\langle\gamma, \mu_1\rangle$,
- when $\mu \cdot \mu_1 \in \text{Domain}(\gamma')$ such that $\mu_1 \in \text{Domain}(\beta)$:
  - $\text{LABEL}\langle\gamma', \mu \cdot \mu_1\rangle = \text{LABEL}\langle\beta, \mu_1\rangle$,
  - $\text{SA}\langle\gamma', \mu \cdot \mu_1\rangle = \text{SA}\langle\beta, \mu_1\rangle$,
  - $\text{OA}\langle\gamma', \mu \cdot \mu_1\rangle = \text{OA}\langle\beta, \mu_1\rangle$,
- when $\mu \cdot \text{ft}(\beta) \cdot \mu_1 \in \text{Domain}(\gamma')$ such that $\mu \cdot \mu_1 \in \text{Domain}(\gamma)$ and $\mu_1 \neq \epsilon$:
  - $\text{LABEL}\langle\gamma', \mu \cdot \text{ft}(\beta) \cdot \mu_1\rangle = \text{LABEL}\langle\gamma, \mu \cdot \mu_1\rangle$,

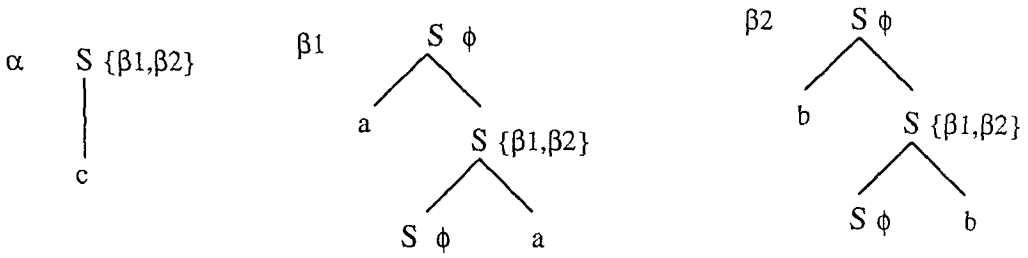

Figure 10
Example of a TAG G.

— $\mathrm{SA}\langle\gamma', \mu \cdot \mathrm{ft}(\beta) \cdot \mu_1\rangle = \mathrm{SA}\langle\gamma, \mu \cdot \mu_1\rangle$,
— $\mathrm{OA}\langle\gamma', \mu \cdot \mathrm{ft}(\beta) \cdot \mu_1\rangle = \mathrm{OA}\langle\gamma, \mu \cdot \mu_1\rangle$,

In general, if μ is the address of a node in γ then by $\langle\gamma, \mu\rangle$ we refer to the elementary node address of the node that contributes to its presence, and hence its label and constraints.

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

Example 5.1
Figure 10 gives a TAG, G, which generates the language $\{wcw \mid w \in \{a,b\}^+\}$. The constraints associated with the root and foot of β specify that no auxiliary trees can be adjoined at these nodes. This is indicated in Figure 10 by associating the empty set, ϕ, with these nodes. An example derivation of the strings *aca* and *abcab* is shown in Figure 11.

5.3 TAG and LIG

In this section, we examine bottom-up recognition of a TAG. In doing so, we construct a LIG that simulates the derivations of the TAG. Based on this construction, we derive a recognizer for TAG from the algorithms given earlier.

Consider bottom-up TAG recognition. Having recognized the substring dominated by an elementary node there are two possible actions: (1) move up the tree by combining this node with its siblings; or (2) consider adjunction at that node. In bottom-up recognition, the second action (i.e., adjunction) must be considered before the first. Therefore, there are two phases involved in the consideration of each node. On entering the **bottom** phase of a node, having just combined the derivations of its children, we predict an adjunction. On entering the **top** phase, having just finished adjunction at that node, we must now combine with any siblings in order to move up the tree. Note that in the bottom phase we may also predict that there is no adjunction at the node (if there is no OA constraint on that node) and hence move to its top phase directly.

Figure 12 shows why, because of the nature of the adjoining operation, TAG can be seen to involve stacking. Suppose, during recognition, the bottom phase of a node, η,

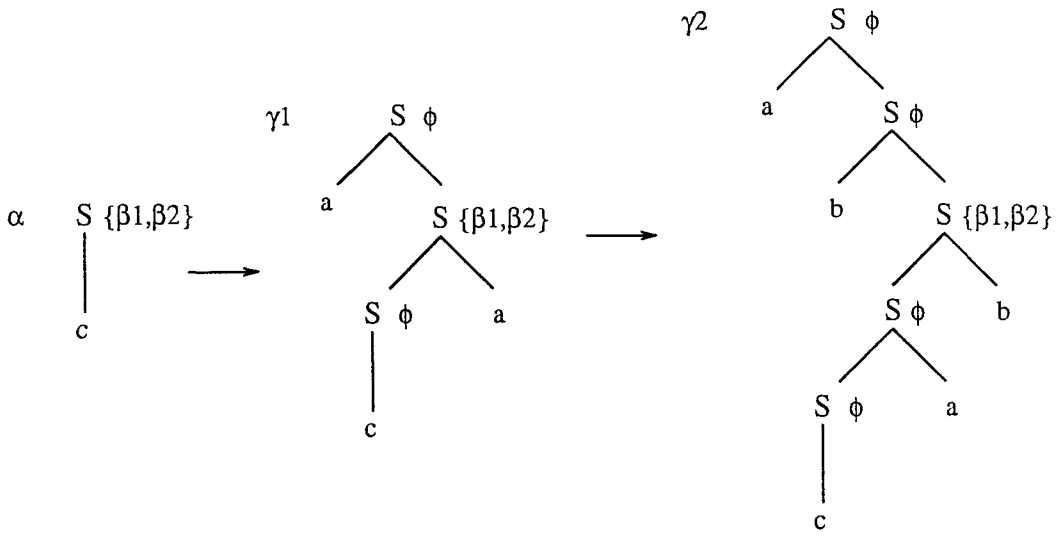

Figure 11
Sample derivations in *G*.

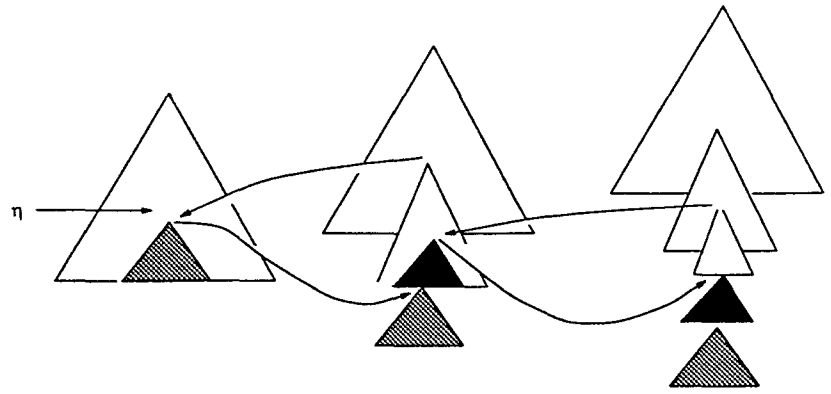

Figure 12
Stacking in a TAG.

has been reached. When adjunction by the auxiliary tree β is predicted, control shifts to the bottom phase of β's foot node. As we move up the spine of β it is necessary to remember that β was adjoined at η. On reaching the top phase of β's root we must return to (the top phase of) η. Therefore, the adjunction point, η, must be propagated up the spine of β. In general, we may need to propagate a stack of adjunction points as we move up the spine as shown in Figure 12 where γ_2 is obtained by adjoining β_1 at a node η_1 on the spine of β. From this figure, it can be seen that the information about the adjunction points (that must be propagated along the spine of an auxiliary tree) follows the stack (last-in first-out) discipline. Notice also that only the nodes on the spine participate in the propagation of adjunction points.

Consider how a LIG that simulates this process can be constructed. The details of the equivalence between LIG and TAG can be found in Vijay-Shanker (1987). In the

LIG, we use two nonterminals, t and b to capture the differences between the top and bottom phases associated with a node. The stack holds an appropriate sequence of adjunction points in the form of elementary node addresses. The top of the stack is the elementary node address of the node that is currently being visited (thus all objects have at least one element on the stack). Nodes that are not on the spine, or belong to an initial tree, do not participate in the propagation of adjunction points. Therefore, the objects for such nodes will have stacks that contain only their elementary node address.

The set of LIG productions is determined as follows. We assume that internal elementary nodes have either a single child labeled by a terminal symbol (or ϵ), or exactly two children labeled by nonterminals. In this discussion below, we will use η for a node and its elementary node address interchangeably.

1. If η is a node that is labeled ε where $\varepsilon \in V_T \bigcup \{\epsilon\}$ then we will include $\mathsf{t}(\eta) \rightarrow \varepsilon$.
2. If η_p and η_s are the children of a node η such that the left sibling η_p (and hence η) is on the spine then the following holds: (1) the object corresponding to η_p can have an unboundedly large stack, whereas the object for η_s will have a stack of size one; (2) the top of the stack in these objects will be η_p and η_s; (3) combination of these two sibling nodes is possible only after the top parts of these nodes are reached; (4) the stack in the object for η_p must be propagated to object for η, except that the top symbol η_p is replaced by η; (5) when the two sibling nodes are combined we reach the bottom part of η. Hence, we include the production $\mathsf{b}(\cdot\cdot\,\eta) \rightarrow \mathsf{t}\left(\cdot\cdot\,\eta_p\right)\ \mathsf{t}(\eta_s)$.
3. If η_p, η_s are children of η as in the previous case except that η_p is the right sibling and is on the spine, then we include the production $\mathsf{b}(\cdot\cdot\,\eta) \rightarrow \mathsf{t}(\eta_s)\ \mathsf{t}\left(\cdot\cdot\,\eta_p\right)$.
4. If η_p, η_s, and η are as before except that neither sibling is on the spine of an auxiliary tree then we include the production $\mathsf{b}(\cdot\cdot\,\eta) \rightarrow \mathsf{t}\left(\cdot\cdot\,\eta_p\right)\ \mathsf{t}(\eta_s)$.
5. If η_p is the only child of η we have $\mathsf{b}(\cdot\cdot\,\eta) \rightarrow \mathsf{t}\left(\cdot\cdot\,\eta_p\right)$.
6. If η is a node where β can be adjoined and we are at the bottom of η, then, by predicting adjunction by β, control moves to the bottom part of η_1 (the foot node of β). This is illustrated in Figure 13. In this case we add the production $\mathsf{b}(\cdot\cdot\,\eta\eta_1) \rightarrow \mathsf{b}(\cdot\cdot\,\eta)$. When there is no OA constraint at η then we can predict that no adjunction takes place. This is captured with the production $\mathsf{t}(\cdot\cdot\,\eta) \rightarrow \mathsf{b}(\cdot\cdot\,\eta)$.
7. Suppose we have reached the top part of the root node, η_2, of the auxiliary tree β. The corresponding object has the nonterminal t with η_2 on top of the stack and the node at which β was adjoined is immediately below η_2. Having reached the top of the root node of β we must return to the top of the node where β was adjoined. This is accomplished with the production $\mathsf{t}(\cdot\cdot\,) \rightarrow \mathsf{t}(\cdot\cdot\,\eta_2)$ (see Figure 13).

Figure 13 captures the essence of the connection between TAG and LIG—in particular the way the adjoining operation in TAG can be simulated in LIG. This figure is also useful in order to understand the notion of terminators. As in the case of CCG, the construction of the LIG equivalent of the given grammar is unnecessary. However, as in the case of CCG, this discussion of the connection between TAG and LIG can be

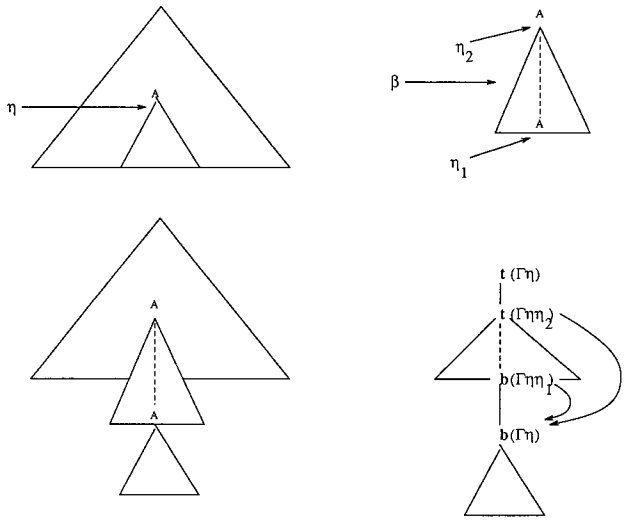

Figure 13
TAG/LIG relationship.

used to motivate the choices we make in the form of entries in TAG parser as well as the rules in the algorithm.

5.4 Recognition of TAG

We now give a CKY-style recognition algorithm for TAG. But first we shall consider the LIG constructed from a given TAG as described in Section 5.3. Given this LIG grammar, consider the objects derived and the form of entries that will be used by the LIG algorithm.

- If η is an elementary node address of a node on the spine of an auxiliary tree, say β, then any object that has η as the top symbol of its stack must be of the form $A(\eta_1 \ldots \eta_k \eta_t \eta)$ where $k \geq 0$, $A \in \{\mathsf{t}, \mathsf{b}\}$, and η_t is the elementary node address of a node where β can be adjoined. Furthermore, in any derivation, the terminator of $A(\varphi\eta_t\eta)$ will be $\mathsf{b}(\varphi\eta_t)$.
- For this LIG, MSL = MTL = TTC = 1 and MCL = 2. Hence it follows that any terminator is a 2-terminator. From the discussion above, an object $A(\varphi\eta)$ (where $A \in \{\mathsf{t}, \mathsf{b}\}$ and $len(\varphi) \geq 0$) has a terminator if and only if η is an elementary node address of a node on the spine of an auxiliary tree.
- Consider the forms of entries for a LIG in this form. First, the length of the middle in a terminator-type entry will be one always, since any terminator is a 2-terminator. Note that the terminator of $A(\varphi\eta_t\eta)$ will be $\mathsf{b}(\varphi\eta_t)$. Thus, a terminator type entry in a parsing array entry, say $P[i,d]$ will have the form $(\langle A, \eta\rangle\ (\eta_t, (\langle b, \eta_t\rangle, [t, d_t])))$ where $A \in \{\mathsf{t}, \mathsf{b}\}$ and $\langle t, d_t\rangle < \langle i, d\rangle$. Note that $\langle b, \eta_t\rangle$ in the terminator-pointer is redundant.
- From the discussion above, a non–terminator-type entry will be used to record derivations from $A(\eta)$ where $A \in \{\mathsf{t}, \mathsf{b}\}$ and η is the elementary node address of a node that belongs to an initial tree or of a node that is

not on the spine of an auxiliary tree. To record this object the entry $(\langle A, \eta\rangle, nil)$ would have been used.

From the above discussion it makes sense that terminator-type entries in the TAG parser have the form $(\langle A, \eta\rangle (\eta_t, t, d_t))$ where $A \in \{\mathsf{t}, \mathsf{b}\}$, η is an elementary node address of a node on the spine of an auxiliary tree, say β, and η_t is the elementary node address of a node where β can be adjoined in. A non–terminator-type entry has the form $(\langle A, \eta\rangle, nil)$ where $A \in \{\mathsf{t}, \mathsf{b}\}$, and η is an elementary node address of a node that is not on the spine of an auxiliary tree.

Finally, consider an auxiliary derivation in the LIG obtained from a TAG as described in Section 5.3. Recall that an auxiliary derivation has the form

$$\begin{aligned} A(\varphi\gamma_1) &\stackrel{1}{\Longrightarrow} \Upsilon_1 A_1 (\varphi\gamma_1\gamma_2)\Upsilon_2 \\ &\stackrel{*}{\Longrightarrow} \Upsilon_1 u A_t (\varphi\gamma_t) w \Upsilon_2 \end{aligned}$$

In this case we would have:

- $\gamma_1 = \gamma_t$,
- $A = A_1 = \mathsf{t}$,
- $A_t = \mathsf{b}$, and
- γ_2 is the root of an auxiliary tree that can be adjoined at the node whose elementary node address is given by η_t.

Since every auxiliary tree in a lexicalized TAG has at least one terminal node in its frontier, every auxiliary derivation spans at least one terminal in the LIG we have constructed.

5.5 Recognition Algorithm

We begin with a description of the cases involved in TAG recognition algorithm.

- **Predicting adjunction**: During the recognition phase, on reaching the bottom part of a node η, we predict adjunction by each auxiliary tree, β that can be adjoined at η as determined by its SA constraints. As given in Case 6 of the construction in Section 5.3, this prediction is captured with the LIG production $\mathsf{b}(\cdot\cdot\,\eta\eta_1) \rightarrow \mathsf{b}(\cdot\cdot\,\eta)$ where η_1 is the foot node of the auxiliary tree, β. Depending on whether η is on the spine of an auxiliary tree or not, we have the following counterparts of Rule 8.u.L and Rule 10.u.L:

  Rule 8.u.T

$$\frac{\eta_1 = \langle\beta, \mathsf{ft}(\beta)\rangle \quad \beta \in \mathrm{SA}\langle\eta\rangle \quad (\langle \mathsf{b}, \eta\rangle (\eta_t, t, d_t)) \in P[i, d]}{(\langle \mathsf{b}, \eta_1\rangle (\eta, i, d)) \in P[i, d]}$$

  Rule 10.u.T

$$\frac{\eta_1 = \langle\beta, \mathsf{ft}(\beta)\rangle \quad \beta \in \mathrm{SA}\langle\eta\rangle \quad (\langle \mathsf{b}, \eta\rangle, nil) \in P[i, d]}{(\langle \mathsf{b}, \eta_1\rangle (\eta, i, d)) \in P[i, d]}$$

As in the second part of Case 6 of the LIG construction (i.e., when there is no OA constraint at the node $\eta\rangle$ we have the following counterparts of Rule 6.u.L and Rule 7.u.L:

Rule 6.i.u.T

$$\frac{\mathrm{OA}\langle\eta\rangle = \mathit{false} \quad \left(\langle \mathsf{b},\eta\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]}{\left(\langle \mathsf{t},\eta\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]}$$

Rule 7.i.u.T

$$\frac{\mathrm{OA}\langle\eta\rangle = \mathit{false} \quad \left(\langle \mathsf{b},\eta\rangle, \mathit{nil}\right) \in P[i,d]}{\left(\langle \mathsf{t},\eta\rangle, \mathit{nil}\right) \in P[i,d]}$$

- **Left sibling on the spine**: This corresponds to Case 2 of the LIG construction. The following rule that captures this situation corresponds to Rule 7.ps.L.

Rule 7.ps.T

$$\frac{\begin{array}{ll}\eta_p \text{ is left child of } \eta & \\ \eta_p \text{ is on the spine of an auxiliary tree} & \eta_p \text{ is right child of } \eta \\ \left(\langle \mathsf{t},\eta_p\rangle\ (\eta_t, t, d_t)\right) \in P[i,d_p] & \left(\langle \mathsf{t},\eta_s\rangle, \mathit{nil}\right) \in P[i+d_p, d-d_p]\end{array}}{\left(\langle \mathsf{b},\eta\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]}$$

The following covers Case 4 of LIG construction where the two siblings are not on the spine or belong to an initial tree and corresponds to Rule 6.ps.L (or Rule 6.sp.L).

Rule 6.ps.T

$$\frac{\begin{array}{ll}\eta_p \text{ is left child of } \eta & \\ \eta \text{ is not on the spine of any auxiliary tree} & \eta_p \text{ is right child of } \eta \\ \left(\langle \mathsf{t},\eta_p\rangle, \mathit{nil}\right) \in P[i,d_p] & \left(\langle \mathsf{t},\eta_s\rangle, \mathit{nil}\right) \in P[i+d_p, d-d_p]\end{array}}{\left(\langle \mathsf{b},\eta\rangle, \mathit{nil}\right) \in P[i,d]}$$

- **Right sibling on the spine**: Corresponding to Case 3 of LIG construction and Rule 7.sp.L we have

Rule 7.sp.T

$$\frac{\begin{array}{ll} & \eta_p \text{ is right child of } \eta \\ \eta_s \text{ is left child of } \eta & \eta_p \text{ is on the spine of an auxiliary tree} \\ \left(\langle \mathsf{t},\eta_s\rangle, \mathit{nil}\right) \in P[i,d_s] & \left(\langle \mathsf{t},\eta_p\rangle\ (\eta_t, t, d_t)\right) \in P[i+d_s, d-d_s]\end{array}}{\left(\langle \mathsf{b},\eta\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]}$$

- **Single child case**: Corresponding to Case 5 of LIG construction, Rule 7.u.L and Rule 6.u.L.

Rule 7.ii.u.T

$$\frac{\begin{array}{ll}\eta_p \text{ is only child of } \eta & \\ \eta_p \text{ is on the spine of some auxiliary tree} & \left(\langle \mathsf{t},\eta_p\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]\end{array}}{\left(\langle \mathsf{b},\eta\rangle\ (\eta_t, t, d_t)\right) \in P[i,d]}$$

Rule 6.ii.u.T

$$\frac{\eta_p \text{ is only child of } \eta \quad \eta \text{ is not on the spine of any auxiliary tree} \quad \left(\langle \mathsf{t}, \eta_p \rangle, nil\right) \in P[i,d]}{\left(\langle \mathsf{b}, \eta \rangle, nil\right) \in P[i,d]}$$

- **Completing an adjunction**: Corresponding to Case 7 of the construction and depending on whether the node of adjunction is on the spine of an auxiliary tree, we have the following counterparts of Rule 4.u.L, Rule 5.u.L.

Rule 4.u.T

$$\frac{\left(\langle \mathsf{t}, \eta_p \rangle \left(\eta_t, t, d_t\right)\right) \in P[i,d] \qquad \begin{array}{c}\eta_t \text{ is not on the spine of any auxiliary tree} \\ \left(\langle \mathsf{b}, \eta_t \rangle, nil\right) \in P[t,d_t]\end{array}}{\left(\langle \mathsf{t}, \eta_t \rangle, nil\right) \in P[i,d]}$$

Rule 5.u.T

$$\frac{\left(\langle \mathsf{t}, \eta_p \rangle \left(\eta_t, t, d_t\right)\right) \in P[i,d] \qquad \begin{array}{c}\eta_t \text{ is on the spine of an auxiliary tree} \\ \left(\langle \mathsf{b}, \eta_t \rangle \left(\eta_r, r, d_r\right)\right) \in P[t,d_t]\end{array}}{\left(\langle \mathsf{t}, \eta_t \rangle \left(\eta_r, r, d_r\right)\right) \in P[i,d]}$$

From the nature of entries being created it will follow that if $\eta_p = \langle \beta, \epsilon \rangle$, for some auxiliary tree β, then β is adjoinable at η_t. Similarly, if $\eta_t = \langle \beta', \mu \rangle$ for some auxiliary tree β', then β' is adjoinable at η_r.

- **Scanning a terminal symbol**: If η is a node labeled by a terminal matching the i^{th} input symbol, a_i, then we have (corresponding to Rule 1.L):

Rule 1.T

$$\frac{\text{LABEL}(\eta) = a_i \quad 1 \leq i \leq n}{\left(\langle \mathsf{t}, \eta \rangle, nil\right) \in P[i,1]}$$

- **Scanning empty string**: If η is a node labeled by ϵ, then we have (corresponding to Rule 1.ϵ.L):

Rule 1.ϵ.T

$$\frac{\text{LABEL}(\eta) = \epsilon}{\left(\langle \mathsf{t}, \eta \rangle, nil\right) \in P[i,0]}$$

This concludes our discussion of the parsing rules for TAG. With the correspondences with the LIG parsing rules given (via the numbering of rules), these rules may be placed in the control structure as suggested in Section 5.1. As noted earlier, in the case of a lexicalized TAG, since every auxiliary derivation spans at least one terminal we do not require the repeat loop discussed in Section 5.1.

6. Conclusion

In this paper we have presented a general scheme for parsing a set of grammar formalisms whose derivation process is controlled by (explicit or implicit) stacking machinery. We have shown how this scheme can be instantiated to give polynomial time algorithms for LIG, CCG, and TAG. In the case of CCG, this provides the only polynomial parsing algorithm (apart from a slight variant of this scheme given in Vijay-Shanker and Weir (1990)) we are aware of.

The main contribution of this paper is the general recognition scheme and definitions of some notions (e.g., terminators, data structures sharing of stacks) crucial to this scheme. We believe that these ideas can be suitably adapted in order to produce parsing schemes based on other CFG parsing algorithms (such as Earley's algorithm). For instance, the definition of terminator given here was tailored for pure bottom-up parsing. In the case of Earley's algorithm, a bottom-up parser with top-down prediction, an additional notion of terminator for the top-down prediction component can be obtained in a straightforward manner.

We have also introduced a new method of representing derivations in a TAG, one that we believe is appropriate in capturing the stacking that occurs during a TAG derivation. The derivations themselves represented can be in another TAG that we call the derivation grammar (see Vijay-Shanker and Weir (1993)).

We have not discussed the extraction of parses after the recognition is complete because of space considerations. However, an algorithm to extract the parses and build a shared forest representation of all parses for CCG was proposed in Vijay-Shanker and Weir (1990). This scheme was based on the approach we have taken in our general scheme. The method of extracting parses and representing them using a shared forest given in Vijay-Shanker and Weir (1990) can be generalized in a straightforward manner to be compatible with the generalized recognition scheme given here.

Acknowledgments

This work has been partially supported by NSF Grants IRI-8909810 and IRI-9016591. We would like to thank A. K. Joshi, B. Lang, Y. Schabes, S. M. Shieber, and M. J. Steedman for many discussions. We are grateful to the anonymous reviewers for their numerous suggestions.

References

Aho, A. V. (1968). "Indexed grammars—An extension to context free grammars." *J. ACM*, 15, 647–671.

Duske, J., and Parchmann, R. (1984). "Linear indexed languages." *Theoretical Comput. Sci.*, 32, 47–60.

Gazdar, G. (1988). "Applicability of indexed grammars to natural languages." In *Natural Language Parsing and Linguistic Theories*, edited by U. Reyle and C. Rohrer. D. Reidel, 69–94.

Joshi, A. K. (1985). "How much context-sensitivity is necessary for characterizing structural descriptions—tree adjoining grammars." In *Natural Language Processing—Theoretical, Computational and Psychological Perspective*, edited by D. Dowty, L. Karttunen, and A. Zwicky. Cambridge University Press, 206–250.

Joshi, A. K.; Levy, L. S.; and Takahashi, M. (1975). "Tree adjunct grammars." *J. Comput. Syst. Sci.*, 10(1), 136–163.

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

Lang, B. (1990). "Towards a uniform formal framework for parsing." In *Current Issues in Parsing Technology*, edited by M. Tomita. Kluwer Academic Publishers, 153–171.

Pareschi, R., and Steedman, M. J. (1987). "A lazy way to chart-parse with categorial grammars." In *Proceedings, 25th Meeting of the Association for Computational Linguistics*, 81–88.

Pollard, C. (1984). *Generalized Phrase Structure Grammars, Head Grammars and Natural Language*. Doctoral dissertation, Stanford University.

Steedman, M. (1986). "Combinators and grammars." In *Categorial Grammars and*

Natural Language Structures, edited by R. Oehrle, E. Bach, and D. Wheeler. Foris, 417–442.

Steedman, M. J. (1985). "Dependency and coordination in the grammar of Dutch and English." *Language*, 61:523–568.

Tomita, M. (1988). "Graph-structured stack and natural language parsing." In *Proceedings, 26th Meeting of the Association for Computational Linguistics*, 248–257.

Vijay-Shanker, K. (1987). *A study of tree adjoining grammars*. Doctoral dissertation, University of Pennsylvania, Philadelphia, PA.

Vijay-Shanker, K., and Joshi, A. K. (1985). "Some computational properties of tree adjoining grammars." In *Proceedings, 23rd Meeting of the Association for Computational Linguistics*, 82–93.

Vijay-Shanker, K., and Weir, D. J. (In press). "The equivalence of four extensions of context-free grammars." *Mathematical Systems Theory*.

Vijay-Shanker, K., and Weir, D. J. (1990). "Polynomial parsing of combinatory categorial grammars." In *Proceedings, 28th Meeting of the Association for Computational Linguistics*, Pittsburgh, PA, 1–8.

Vijay-Shanker, K., and Weir, D. J. (1993). "The use of shared forests in TAG parsing." In *Proceedings, 6th Meeting of the European Association for Computational Linguistics*, Utrecht, The Netherlands, 384–393.

Weir, D. J. (1988). *Characterizing mildly context-sensitive grammar formalisms*. Doctoral dissertation, University of Pennsylvania, Philadelphia, PA.

Weir, D. J., and Joshi, A. K. (1988). "Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems." In *Proceedings, 26th Meeting of the Association for Computational Linguistics*, 278–285.

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

Appendix A: Correctness of Algorithm 1

We will now prove the correctness of Algorithm 1. In doing so, we will start by observing some properties of the rules and the control structure used.

Firstly, given an input is $a_1 \ldots a_n$, we can note that every entry added by a rule (i.e., consequents of rules) satisfies the requirements for the terminator-type and non-terminator-type entries; viz., if $\left(\langle A, \gamma\rangle \left(\beta, \left(\langle A_t, \gamma_t\rangle, [t, d_t]\right)\right)\right)$ is added to an array element $P[i, d]$ then

- $A, A_t \in V_N$,
- $\gamma, \gamma_t \in V_I$,
- $\beta \in V_I^+$ where $1 \leq len(\beta) \leq \text{MCL} - 1$ and
- $\langle t, d_t\rangle < \langle i, d\rangle \leq \langle 1, n\rangle$ where $d \geq 2$.

We can also note that if $\left(\langle A, \gamma\rangle \left(\beta, nil\right)\right)$ is added to $P[i, d]$ then

- $A \in V_N$,
- $\gamma \in V_I$,
- $\beta \in V_I^*$ where $0 \leq len(\beta) < \text{TTC} + \text{MCL} - 1$ and
- $d \geq 1$.

These can be verified from noting the form of the rules and by simple induction on $\langle i, d\rangle$. We can also observe from the control structure given that entries to $P[i_1, d_1][i_2, d_2]$ are added before entries are added to $P[i_3, d_3][i_4, d_4]$ if and only if $\langle i_1, d_1\rangle < \langle i_3, d_3\rangle$ or $\langle i_1, d_1\rangle = \langle i_3, d_3\rangle$ and $\langle i_2, d_2\rangle > \langle i_4, d_4\rangle$. This observation can be used to show that when

a rule is considered for the purposes of adding an entry to $P[i_1, d_1][i_2, d_2]$ then the array elements specified in the antecedent of that rule would have already been filled. Verifying these properties of the algorithm enables us to establish the correctness of the algorithm more easily.

Theorem A.1

- $$(\langle A, \gamma\rangle (\alpha, (\langle A_t, \gamma_t\rangle, [t, d_t]))) \in P[i, d]$$

  if and only if

  $$\begin{aligned} A(\beta\alpha\gamma) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\beta\gamma_t) a_t \ldots a_{i+d-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1} \end{aligned}$$

  for some β such that $A_t(\beta\gamma_t)$ is the $len(\alpha\gamma)$-terminator of $A(\beta\alpha\gamma)$ in this derivation and $len(\beta\gamma_t) \geq$ TTC.

- $$(\langle A, \gamma\rangle (\alpha, nil)) \in P[i, d]$$

  if and only if

  $$A(\alpha\gamma) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$$

  where $A(\alpha\gamma)$ does not have the TC–property, i.e., $A(\alpha\gamma)$ has no terminator in this derivation or the terminator, say $A_t(\beta\gamma_t)$, is such that $len(\beta\gamma_t) <$ TTC.

Proof of Soundness:
We prove the soundness by inducting on d. The base case corresponds to $d = 1$. We have to consider only entries of the form $(\langle A, \gamma\rangle (\alpha, nil))$ in $P[i, 1]$. Such entries are added only by the application of Rule 1. Therefore, we have $A(\alpha\gamma) \rightarrow a$ and $a = a_i$. Hence $A(\alpha\gamma) \stackrel{*}{\Longrightarrow} a_i$ as required.

Now, for the inductive step, let $d \geq 2$. Any entry $(\langle A, \gamma\rangle (\alpha, tp))$ added to $P[i, d]$ where $d \geq 2$ must be due to a rule other than Rule 1.L. This means that we have either a production $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot \gamma_p) A_s(\alpha_s)$ or $A(\cdot\cdot \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot \gamma_p)$. Let us assume that the first production was used. We will discuss the cases for $m = 0$, $m = 1$, and $m \geq 2$ separately.

- Let $m = 0$. In this case the production is $A(\cdot\cdot) \rightarrow A_p(\cdot\cdot \gamma_p) A_s(\alpha_s)$. Then the entry $(\langle A, \gamma\rangle (\alpha, tp))$ should have been added by using one of rules 1.ps.L through 5.ps.L. We take Rule 4.ps.L as a representative. If $(\langle A, \gamma_1\rangle (\beta_t, nil))$ were to be added as a result of this rule, then we have to show that $A(\beta_t\gamma_1) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ where $A(\beta_t\gamma_1)$ does not meet the TC-property. Since $\langle i, d_p\rangle < \langle i, d\rangle$, $\langle i + d_p, d - d_s\rangle < \langle i, d\rangle$, and $\langle t, d_t\rangle < \langle i, d\rangle$ the inductive hypothesis applies to the three entries in the antecedent. Thus, we have for some α the following derivations:

  $$\begin{aligned} A_t(\beta_t\gamma_t) &\stackrel{*}{\Longrightarrow} a_t \ldots a_{t+d_t-1} \\ A_s(\alpha_s) &\stackrel{*}{\Longrightarrow} a_{i+d_p} \ldots a_{i+d-1} \\ A_p(\alpha\gamma_1\gamma_p) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\alpha\gamma_t) a_{t+d_t} \ldots a_{i+d_p-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1} \end{aligned}$$

such that $A_t(\beta_t\gamma_t)$ does not meet the TC-property. However, $A_p(\alpha\gamma_1\gamma_p)$ satisfies the TC-property and furthermore $A_t(\alpha\gamma_t)$ is the 2-terminator of $A_p(\alpha\gamma_1\gamma_p)$. From Observation 2.1, we can also infer the existence of the following derivation.

$$\begin{aligned} A_p(\beta_t\gamma_1\gamma_p) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\beta_t\gamma_t)\, a_{t+d_t} \ldots a_{i+d_p-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1} \end{aligned}$$

Combining this derivation with the derivation from $A_s(\alpha_s)$ we have

$$\begin{aligned} A(\beta_t\gamma_1) &\stackrel{*}{\Longrightarrow} A_p(\beta_t\gamma_1\gamma_p)\, A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\beta_t\gamma_t)\, a_{t+d_t} \ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1} \end{aligned}$$

From Observation 2.5, we know that the terminator of $A_t(\beta_t\gamma_t)$ in this derivation is also the terminator of $A(\beta_t\gamma_1)$ (and if $A_t(\beta_t\gamma_t)$ has no terminator then neither does $A(\beta_t\gamma_1)$). Since $A_t(\beta_t\gamma_t)$ does not satisfy the TC-property (i.e., it does not have a terminator with stack length greater than or equal to TTC), $A(\beta_t\gamma_1)$ does not satisfy the TC-property either. Thus we have shown the existence of the required derivation.

- Let $m = 1$. Therefore the production may be written as $A(\cdot\cdot\,\gamma_1) \rightarrow A_p(\cdot\cdot\,\gamma_p)\, A_s(\alpha_s)$. This time we will take Rule 6.ps.L as a representative. Hence, we can assume that the entry added to $P[i,d]$ has the form $(\langle A, \gamma_1\rangle\, (\beta_p, nil))$. Since $\langle i, d_p\rangle < \langle i, d\rangle$, and $\langle i + d_p, d - d_s\rangle < \langle i, d\rangle$, the inductive hypothesis applies to the two entries in the antecedent. Thus, we have the following derivations:

$$\begin{aligned} A_p(\beta_p\gamma_p) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1} \\ A_s(\alpha_s) &\stackrel{*}{\Longrightarrow} a_{i+d_p} \ldots a_{i+d-1} \end{aligned}$$

Therefore we have the derivation:

$$\begin{aligned} A(\beta_p\gamma_1) &\stackrel{*}{\Longrightarrow} A_p(\beta_p\gamma_p)\, A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots\ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \\ &= a_i \ldots a_{i+d-1} \end{aligned}$$

Note that any terminator of $A_p(\beta_p\gamma_p)$ is also the terminator of $A(\beta_p\gamma_1)$ (and if $A_p(\beta_p\gamma_p)$ has no terminator then neither has $A(\beta_p\gamma_1)$). Since $A_p(\beta_p\gamma_p)$ does not meet the TC-property in this derivation (from inductive hypothesis), neither does $A(\beta_p\gamma_p)$. Thus we have shown the existence of the required derivation.

- Let $m \geq 2$. We will consider the application of Rule 10.ps.L as a representative. Again, applying the inductive hypothesis we have the following derivations:

$$\begin{aligned} A_p(\beta_p\gamma_p) &\stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d_p-1} \\ A_s(\alpha_s) &\stackrel{*}{\Longrightarrow} a_{i+d_p} \ldots a_{i+d-1} \end{aligned}$$

where $len(\beta_p\gamma_p) \geq$ TTC. Combining the two derivations, we have:

$$\begin{aligned} A(\beta_p\gamma_1 \ldots \gamma_m) &\stackrel{*}{\Longrightarrow} A_p(\beta_p\gamma_p) A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots\ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \\ &= a_i \ldots a_{i+d-1} \end{aligned}$$

Since $m \geq 2$, $A_p(\beta_p\gamma_p)$ is the m-terminator of $A(\beta_p\gamma_1 \ldots \gamma_m)$ in the above derivation. Since $len(\beta_p\gamma_p) \geq$ TTC, we have shown the existence of the required derivation and that $A(\beta_p\gamma_1 \ldots \gamma_m)$ satisfies the TC-property.

In a similar manner we can consider other rules (including those that assume a production of the form $A(\cdot\cdot\ \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot\ \gamma_p)$) as well.

Proof of Completeness:
We will now show the completeness of Algorithm 1. This time we use induction on the number of steps in a derivation. Suppose $A(\beta) \stackrel{l}{\Longrightarrow} a_i \ldots a_{i+d-1}$; we have to show that there is a corresponding entry (as specified in Theorem A.1) in $P[i,d]$.

The base case corresponds to $l = 1$. From the form of the productions being considered we can assume that $d = 1$ and that there exists a production $A(\alpha) \rightarrow a_i$. Rule 1 would apply and thus we have the required entry.

Let $A(\beta) \stackrel{l+1}{\Longrightarrow} a_i \ldots a_{i+d-1}$ where $l \geq 1$. The first production used in this derivation must have the form $A(\cdot\cdot\ \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\ \gamma_p) A_s(\alpha_s)$ or $A(\cdot\cdot\ \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot\ \gamma_p)$. We will only assume that the production is $A(\cdot\cdot\ \gamma_1 \ldots \gamma_m) \rightarrow A_p(\cdot\cdot\ \gamma_p) A_s(\alpha_s)$. Arguments similar to the one given below can be used when the production of the form $A(\cdot\cdot\ \gamma_1 \ldots \gamma_m) \rightarrow A_s(\alpha_s) A_p(\cdot\cdot\ \gamma_p)$ is involved as the first step of the derivation.

Case $m = 0$**:** We begin by considering the case when $m = 0$. Since the first production used in $A(\beta) \stackrel{*}{\Longrightarrow} a_i \ldots a_{i+d-1}$ is $A(\cdot\cdot\ ) \rightarrow A_p(\cdot\cdot\ \gamma_p)\ A_s(\alpha_s)$, we can write the derivation as

$$\begin{aligned} A(\beta) &\stackrel{1}{\Longrightarrow} A_p(\beta\gamma_p) A_s(\alpha_s) \\ &\stackrel{l_p}{\Longrightarrow} a_i \ldots a_{i+d_p-1} A_s(\alpha_s) \\ &\stackrel{l_s}{\Longrightarrow} a_i \ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \end{aligned}$$

for some $1 \leq d_p < d$ and $l_p + l_s = l$. Applying the inductive hypothesis to the derivation $A_s(\alpha_s) \stackrel{l_s}{\Longrightarrow} a_{i+d_p} \ldots a_{i+d-1}$, we can assume the existence of the entry

$$(\langle A_s, top(\alpha_s)\rangle\ (rest(\alpha_s), nil))$$

in $P[i+d_p, d-d_p]$.

In order to show the existence of the appropriate type of entry corresponding to the derivation of $a_i \ldots a_{i+d-1}$ from $A(\beta)$, we need to consider whether $A(\beta)$ satisfies the TC-property in this derivation. This could depend on whether the primary constituent $A_p(\beta_p\gamma_p)$ does. Since the inductive hypothesis applies for the derivation $A_p(\beta\gamma_p) \stackrel{l_p}{\Longrightarrow} a_i \ldots a_{i+d_p-1}$. Let us start by assuming that $A(\beta)$ satisfies the TC-property. This means that it has a (say) $(k+1)$-terminator whose stack length is greater than or equal to TTC. Expressing β as $\beta_t\gamma_1 \ldots \gamma_k$, we can then rewrite the derivation from $A(\beta)$ as follows.

$$\begin{aligned} A(\beta_t\gamma_1 \ldots \gamma_k) &\Longrightarrow A_p(\beta_t\gamma_1 \ldots \gamma_k\gamma_p) A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots a_{t-1} A_t(\beta_t\gamma_t) a_{t+d_t} \ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \\ &\stackrel{l_s}{\Longrightarrow} a_i \ldots a_{t-1} a_t \ldots a_{t+d_t-1} a_{t+d_t} \ldots a_{i+d-1} \end{aligned}$$

where $A_t(\beta_t\gamma_t)$ is the terminator of $A_p(\beta_t\gamma_1 \ldots \gamma_k\gamma_p)$. Thus, $len(\beta_t\gamma_t) \geq$ TTC and $k \geq 1$. Now, $A_t(\beta_t\gamma_t)$ is the terminator of $A(\beta_t\gamma_1 \ldots \gamma_k)$ if and only if $k > 1$ (from Observation 2.5).

- Let $k > 1$. $A_t(\beta_t\gamma_t)$ is the terminator of $A(\beta_t\gamma_1 \ldots \gamma_k)$ and $len(\beta_t\gamma_t) \geq$ TTC. Thus, $A(\beta_t\gamma_1 \ldots \gamma_k)$ satisfies the TC-property. Therefore we must show that the entry $(\langle A, \gamma_k\rangle (\gamma_1 \ldots \gamma_{k-1}, (\langle A_t, \gamma_t\rangle, [t, d_t])))$ belongs to $P[i, d]$. By inductive hypothesis we may assume $(\langle A_p, \gamma_p\rangle (\gamma_1 \ldots \gamma_k, (\langle A_t, \gamma_t\rangle, [t, d_t])))$ belongs to $P[i, d_p]$. Now all the conditions in the antecedent of Rule 3.ps.L have been met and thus we have shown the existence of the appropriate entry to record the derivation of $a_i \ldots a_{i+d-1}$ from $A(\beta)$.
- Let $k = 1$. From Observation 2.5 it follows that the k'-terminator of $A_t(\beta_t\gamma_t)$ (if it exists) is also the k'-terminator of $A(\beta_t\gamma_1)$, and if $A_t(\beta_t\gamma_t)$ has no terminator then neither does $A(\beta_t\gamma_1)$. Therefore $A(\beta) = A(\beta_t\gamma_1)$ satisfies the TC-property if and only if $A_t(\beta_t\gamma_t)$ does. Suppose $A_t(\beta_t\gamma_t)$ satisfies the TC-property; then all conditions stated in the antecedent of Rule 5.ps.L are met and the appropriate entry is added to record the derivation from $A(\beta)$. On the other hand, if $A_t(\beta_t\gamma_t)$ does not satisfy the TC-property then all conditions stated in the antecedent of Rule 4.ps.L are met and the appropriate entry is added to record the derivation from $A(\beta)$.

Case $m = 1$**:** Here we are concerned with the situation where $A(\cdot\cdot\gamma_1) \rightarrow A_p(\cdot\cdot\gamma_p)\ A_s(\alpha_s)$ is the first production used in the derivation of $a_i \ldots a_{i+d-1}$ from $A(\beta)$. Rewriting β as $\beta_p\gamma_1$ we have

$$\begin{aligned} A(\beta_p\gamma_1) &\stackrel{*}{\Longrightarrow} A_p(\beta_p\gamma_p)\, A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots\ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \end{aligned}$$

Applying the inductive hypothesis we have

$$(\langle A_s, top(\alpha_s)\rangle (rest(\alpha_s), nil)) \in P[i + d_p, d - d_p].$$

Now any k-terminator of $A_p(\beta_p\gamma_p)$ is also the k-terminator of $A(\beta_p\gamma_1)$ (and if $A_p(\beta_p\gamma_p)$ has no terminator then neither does $A(\beta_p\gamma_1)$). That is, $A(\beta_p\gamma_1)$ satisfies the TC-property in this derivation if and only if $A_p(\beta_p\gamma_p)$ does. If $A_p(\beta_p\gamma_p)$ does not satisfy the TC-property, then, by inductive hypothesis, we have $(\langle A_p, \gamma_p\rangle (\beta_p, nil)) \in P[i, d_p]$. Thus the entries corresponding to the antecedents of Rule 6.ps.L exist and the algorithm would have added the entry $(\langle A, \gamma_1\rangle (\beta_p, nil)) \in P[i, d]$ as desired. If $A_p(\beta_p\gamma_p)$ does satisfy the TC-property then Rule 7.ps.L would add the required entry to record the derivation from $A(\beta)$.

Case $m \geq 2$**:** Finally, consider that case when $m \geq 2$. The given derivation may be expressed as

$$\begin{aligned} A(\beta_p\gamma_1 \ldots \gamma_m) &\stackrel{*}{\Longrightarrow} A_p(\beta_p\gamma_p)\, A_s(\alpha_s) \\ &\stackrel{*}{\Longrightarrow} a_i \ldots\ldots a_{i+d_p-1} a_{i+d_p} \ldots a_{i+d-1} \\ &= a_i \ldots a_{i+d-1} \end{aligned}$$

Applying the inductive hypothesis we have

$$(\langle A_s, top(\alpha_S)\rangle (rest(\alpha_s), nil)) \in P[i + d_p, d - d_p].$$

Since $A_p\,(\beta_p\gamma_p)$ is the m-terminator of $A\,(\beta_p\gamma_1 \ldots \gamma_m)$, we have to consider its length in order to know whether $A\,(\beta_p\gamma_1 \ldots \gamma_m)$ satisfies the TC-property, i.e., how it must be represented. Suppose $len\,(\beta_p\gamma_p) <$ TTC, then by inductive hypothesis we have the entry $(\langle A_p, \gamma_p\rangle\,(\beta_p, nil)) \in P[i, d_p]$. Thus all antecedents of Rule 9.ps.L have been found. Since the terminator of $A\,(\beta_p\gamma_1 \ldots \gamma_m)$ has a stack of length less than TTC, the required entry, $(\langle A, \gamma_m\rangle\,(\beta_p\gamma_1 \ldots \gamma_{m-1}, nil))$, is added by the algorithm by the application of Rule 9.ps.L. Suppose $len\,(\beta_p\gamma_p) \geq$ TTC, then $A_p\,(\beta_p\gamma_p)$ may or may not be represented as a terminator-type entry. Let us take the case where $A_p\,(\beta_p\gamma_p)$ does not satisfy the TC-property. Again by inductive hypothesis, we have the entry $(\langle A_p, \gamma_p\rangle\,(\beta_p, nil)) \in P[i, d_p]$. Since $len\,(\beta_p\gamma_p) \geq$ TTC and the antecedents entries of Rule 10.ps.L exist, the algorithm would add $(\langle A, \gamma_m\rangle\,(\gamma_1 \ldots \gamma_{m-1}, (\langle A_p, \gamma_p\rangle, [i, d_p])))$ to $P[i, d]$ as desired. If we had assumed $A_p\,(\beta_p\gamma_p)$ satisfies the TC-property, then by applying the inductive hypothesis we can guarantee the existence of the entries corresponding to the antecedent of Rule 8.ps.L, and therefore the algorithm would have added

$$(\langle A, \gamma_m\rangle\,(\gamma_1 \ldots \gamma_{m-1}, (\langle A_p, \gamma_p\rangle, [i, d_p])))$$

to $P[i, d]$ as desired.