Recognition of Combinatory Categorial Grammars and Linear Indexed Grammars

K. Vijay-Shanker
Department of CIS
University of Delaware
Delaware, DE 19716

David J. Weir
Department of EECS
Northwestern University
Evanston, IL 60208

1 Introduction

In recent papers [14,15,3] we have shown that Combinatory Categorial Grammars (CCG), Head Grammars (HG), Linear Indexed Grammars (LIG), and Tree Adjoining Grammars (TAG) are weakly equivalent; i.e., they generate the same class of string languages. Although it is known that there are polynomial-time recognition algorithms for HG and TAG [7,11], there are no known polynomial-time recognition algorithms that work *directly* with CCG or LIG. In this paper we present polynomial-time recognition algorithms for CCG and LIG that resemble the CKY algorithm for Context-Free Grammars (CFG) [4,16].

The tree sets derived by a CFG can be recognized by *finite state* tree automata [10][1]. This is reflected in CFL bottom-up recognition algorithms such as the CKY algorithm. Intermediate configurations of the recognizer can be encoded by the states of these finite state automata (the nonterminal symbols of the grammar). The similarity of TAG, CCG, and LIG can be seen from the fact that the tree sets derived by these formalisms can be recognized by *pushdown* (rather than finite state) based tree automata. We give recognition algorithms for these formalisms by extending the CKY algorithm so that intermediate configurations are encoded using stacks. In [6] a chart parser for CCG is given where copies of stacks (derived categories) are stored explicitly in each chart entry. In Section 4 we show that storing stacks in this way leads to exponential run-time. In the algorithm we present here the stack is encoded by storing its top element together with information about where the remainder of the stack can be found. Thus, we avoid the need for multiple copies of parts of the same stack through the sharing of common substacks. This reduces the number of possible elements in each entry in the chart and results in a polynomial time algorithm since the time complexity is related to the number of elements in each chart entry.

It is not necessary to derive separate algorithms for CCG, LIG, and TAG. In proving that these formalisms are equivalent, we developed constructions that map grammars between the different formalisms. We can make use of these constructions to adapt an algorithm for one formalism into an algorithm for another. First we present a discussion of the recognition algorithm for LIG in Section 2[2].

[1] A bottom-up finite state tree automaton reads a tree bottom-up. The state that the automaton associates with each node that it visits will depend on the states associated with the children of the node.

[2] We consider LIG that correspond to the Chomsky normal form for CFG although we do not prove that all LIG have an equivalent grammar in this form. A discussion of the recognition algorithm for LIG in this form is sufficient to enable us to adapt it to give a recognition algorithm for CCG, which is the primary purpose of this paper.

We present the LIG recognition algorithm first since it appeares to be the clearest example involving the use of the notion of stacks in derivations. In Section 3 we give an informal description of how to map a CCG to an equivalent LIG. Based on this relationship we adapt the recognition algorithm for LIG to one for CCG.

2 Linear Indexed Grammars

An Indexed Grammar [1] can be viewed as a CFG in which each nonterminal is associated with a stack of symbols. In addition to rewriting nonterminals, productions can have the effect of pushing or popping symbols on top of the stacks that are associated with each nonterminal. A LIG [2] is an Indexed Grammar in which the stack associated with the nonterminal of the LHS of each production can only be associated with one of the occurrences of nonterminals on the RHS of the production. Empty stacks are associated with other occurrences of nonterminals on the RHS of the production. We write $A[\cdot\cdot]$ (or $A[\cdot\cdot\gamma]$) to denote the nonterminal A associated with an arbitrary stack (or an arbitrary stack whose top symbol is γ). A nonterminal A with an empty stack is written $A[\,]$.

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, having one of the following forms.

$$A[\cdot\cdot\gamma] \rightarrow A_1[\,]\ldots A_i[\cdot\cdot]\ldots A_n[\,] \qquad A[\cdot\cdot] \rightarrow A_1[\,]\ldots A_i[\cdot\cdot\gamma]\ldots A_n[\,] \qquad A[\,] \rightarrow a$$

where $A, A_1, \ldots, A_n \in V_N$ and $a \in \{\epsilon\} \cup V_T$.

The relation $\underset{G}{\Longrightarrow}$ is defined as follows where $\alpha \in V_I^*$ and Υ_1, Υ_2 are strings of nonterminals with associated stacks.

- If $A[\cdot\cdot\gamma] \rightarrow A_1[\,]\ldots A_i[\cdot\cdot]\ldots A_n[\,] \in P$ then

$$\Upsilon_1 A[\alpha\gamma]\Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 A_1[\,]\ldots A_i[\alpha]\ldots A_n[\,]\Upsilon_2$$

- If $A[\cdot\cdot] \rightarrow A_1[\,]\ldots A_i[\cdot\cdot\gamma]\ldots A_n[\,] \in P$ then

$$\Upsilon_1 A[\alpha]\Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 A_1[\,]\ldots A_i[\alpha\gamma]\ldots A_n[\,]\Upsilon_2$$

In each of these two cases we say that A_i is the **distinguished** child of A in the derivation.

- If $A[\,] \rightarrow a \in P$ then

$$\Upsilon_1 A[\,]\Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 a \Upsilon_2$$

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

2.1 Recognition of LIG

In considering the recognition of LIG, we assume that the underlying CFG is in Chomsky Normal Form; i.e., either two nonterminals (with their stacks) or a single terminal can appear on the RHS of a rule. Although we have not confirmed whether this yields a normal form, a recognition algorithm for LIG in this form of LIG is sufficient to enable us to develop a recognition algorithm for CCG. We use an array L consisting of n^2 elements where the string to be recognized is $a_1 \ldots a_n$. In the case of the CKY algorithm for CFG recognition each array element $L_{i,j}$ contains that subset of the nonterminal symbols that can derive the substring $a_i \ldots a_j$. In our algorithm the elements stored in $L_{i,j}$ will encode those nonterminals *and associated stacks* that can derive the string $a_i \ldots a_j$.

In order to obtain a polynomial algorithm we must encode the stacks efficiently. With each nonterminal we store only the top of its associated stack and an indication of the element in L where the next part of the stack can be found. This is achieved by storing sets of tuples of the form $(A, \gamma, A', \gamma', p, q)$ in the array elements. Roughly speaking, a tuple $(A, \gamma, A', \gamma', p, q)$ is stored in $L_{i,j}$ when $A[\alpha\gamma'\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$ and $A'[\alpha\gamma'] \stackrel{*}{\Longrightarrow} a_p \ldots a_q$ where α is a string of stack symbols and A' is the unique distinguished descendent of A in the derivation of $a_i \ldots a_j$.

Note that tuples, as defined above, assume the presence of at least two stack symbols. We must also consider two other cases in which a nonterminal is associated with either a stack of a single element, or with the empty stack. Suppose that A is associated with a stack containing only the single symbol γ. This case will be represented using tuples of the form $(A, \gamma, A', -, p, q)$ ("$-$" indicates that an empty stack is associated with A'). When an empty stack is associated with A we will use the tuple $(A, -, -, -, -, -)$. In discussing the general case for tuples we will use the form $(A, \gamma, A', \gamma', p, q)$ with the understanding that: $A' \in V_N$ or $-$; $\gamma, \gamma' \in V_I$ or $-$; and p, q are integer between 1 and n or $-$. The algorithm can be understood by verifying that at each step the following invariant holds.

Proposition 2.1 $(A, \gamma, A', \gamma', p, q) \in L_{i,j}$ if and only if one of the following holds.

If $\gamma' \neq -$ then $A[\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_{p-1} A'[]a_{q+1} \ldots a_j$ and $A'[\alpha\gamma'] \stackrel{*}{\Longrightarrow} a_p \ldots a_q$ for some $\alpha \in V_I^*$ where A' is a distinguished descendent of A. Note that this implies that for all $\beta \in V_I^*$, $A[\beta\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_{p-1} A'[\beta]a_{q+1} \ldots a_j$. Thus, for $\beta = \alpha\gamma'$, $A[\alpha\gamma'\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_{p-1} A'[\alpha\gamma']a_{q+1} \ldots a_j$ which implies $A[\alpha\gamma'\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$.

If $\gamma' = - \neq A'$ then $A[\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$ and $A'[] \stackrel{*}{\Longrightarrow} a_p \ldots a_q$.

If $A' = -$ then $A[] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$.

We now describe how each entry $L_{i,j}$ is filled. As the algorithm proceeds, the gap between i and j increases until it spans the entire input. The input, $a_1 \ldots a_n$, is accepted if $(S, -, -, -, -, -) \in L_{1,n}$. New entries are added to the array elements according to the productions of the grammar as follows.

1. The production $A[\cdot\cdot\gamma] \rightarrow A_1[]A_2[\cdot\cdot]$ is used while filling the array element $L_{i,j}$ as follows. For every k where $i \leq k \leq j$, check the previously completed array elements $L_{i,k}$ and $L_{k+1,j}$ for $(A_1, -, -, -, -, -)$ and some $(A_2, \gamma_2, A_3, \gamma_3, p, q)$, respectively. If these entries are found add $(A, \gamma, A_2, \gamma_2, k+1, j)$ to $L_{i,j}$. If $\gamma_2 = \gamma_3 = A_3 = p = q = -$ we place $(A, \gamma, A_2, -, k+1, j)$ in $L_{i,j}$. From these entries in $L_{i,k}$ and $L_{k+1,j}$ we know by Proposition 2.1 that $A_1[] \stackrel{*}{\Longrightarrow} a_i \ldots a_k$

and $A_2[\alpha] \stackrel{*}{\Longrightarrow} a_{k+1} \ldots a_j$ for some $\alpha \in V_f^*$. Thus, $A[\alpha\gamma] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$. The production $A[\cdots\gamma] \rightarrow A_1[\cdots]A_2[\,]$ is handled similarly.

2. Suppose $A[\cdots] \rightarrow A_1[\,]A_2[\cdots\gamma]$ is a production. When filling $L_{i,j}$ we must check whether the tuple $(A_1, -, -, -, -, -)$ is in $L_{i,k}$ and $(A_2, \gamma, A_3, \gamma_3, p, q)$ is in $L_{k+1,j}$ for some k between i and j. If we do find these tuples then we check in $L_{p,q}$ for some $(A_3, \gamma_3, A_4, \gamma_4, r, s)$. In this case we add $(A, \gamma_3, A_4, \gamma_4, r, s)$ to $L_{i,j}$. If $\gamma_3 = -$ then the stack associated with A_3 is empty, $\gamma_4 = A_4 = r = s = -$, and we add the tuple $(A, -, -, -, r, s)$ to $L_{i,j}$. The above steps can be related to Proposition 2.1 as follows.

   (a) If $\gamma_3 \neq -$ then for some $\alpha \in V_f^*$, $A_4[\alpha\gamma_4] \stackrel{*}{\Longrightarrow} a_r \ldots a_s$ a subderivation of $A_3[\alpha\gamma_4\gamma_3] \stackrel{*}{\Longrightarrow} a_p \ldots a_q$ a subderivation of $A_2[\alpha\gamma_4\gamma_3\gamma] \stackrel{*}{\Longrightarrow} a_{k+1} \ldots a_j$. Combining this with $A_1[\,] \stackrel{*}{\Longrightarrow} a_i \ldots a_k$ we have $A[\alpha\gamma_4\gamma_3] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$.

   (b) If $\gamma_3 = -$ then $A_3[\,] \stackrel{*}{\Longrightarrow} a_p \ldots a_q$ is a subderivation of $A_2[\gamma] \stackrel{*}{\Longrightarrow} a_{k+1} \ldots a_j$. Combining with $A_1[\,] \stackrel{*}{\Longrightarrow} a_i \ldots a_k$, we get $A[\,] \stackrel{*}{\Longrightarrow} a_i \ldots a_j$.

   Productions of the form $A[\cdots] \rightarrow A_1[\cdots\gamma]A_2[\,]$ are handled similarly.

3. Suppose $A[\,] \rightarrow a$ is a production. This is used by the algorithm in the initialization of the array L. If the terminal symbol a is the same as the i^{th} symbol in the input string, i.e., $a = a_i$, then we include $(A, -, -, -, -, -)$ in the array element $L_{i,i}$.

2.2 Complete Algorithm

For $i := 1$ to n do

  $L_{i,i} := \{(A, -, -, -, -, -) \mid A[\,] \rightarrow a_i\}$

For $i := n$ to 1 do

  For $j := i$ to n do

    For $k := i$ to $j - 1$ do

      Step 1a. For each production $A[\cdots\gamma] \rightarrow A_1[\,]A_2[\cdots]$

        if $(A_1, -, -, -, -, -) \in L_{i,k}$ and $(A_2, \gamma_2, A_3, \gamma_3, p, q) \in L_{k+1,j}$

        then $L_{i,j} := L_{i,j} \cup \{(A, \gamma, A_2, \gamma_2, k+1, j)\}$

      Step 1b. For each production $A[\cdots\gamma] \rightarrow A_1[\cdots]A_2[\,]$

        if $(A_1, \gamma_1, A_3, \gamma_3, p, q) \in L_{i,k}$ and $(A_2, -, -, -, -, -) \in L_{k+1,j}$

        then $L_{i,j} := L_{i,j} \cup \{(A, \gamma, A_1, \gamma_1, i, k)\}$

      Step 2a. For each production $A[\cdots] \rightarrow A_1[\,]A_2[\cdots\gamma]$

        if $(A_2, \gamma, A_3, \gamma_3, p, q) \in L_{k+1,j}$, $(A_3, \gamma_3, A_4, \gamma_4, r, s) \in L_{p,q}$, and $(A_1, -, -, -, -, -) \in L_{i,k}$

        then $L_{i,j} := L_{i,j} \cup \{(A, \gamma_3, A_4, \gamma_4, r, s)\}$

      Step 2b. For each production $A[\cdots] \rightarrow A_1[\cdots\gamma]A_2[\,]$

        if $(A_1, \gamma, A_3, \gamma_3, p, q) \in L_{i,k}$, $(A_3, \gamma_3, A_4, \gamma_4, r, s) \in L_{p,q}$, and $(A_2, -, -, -, -, -) \in L_{k+1,j}$

        then $L_{i,j} := L_{i,j} \cup \{(A, \gamma_3, A_4, \gamma_4, r, s)\}$

2.3 Complexity of the Algorithm

Any array element, say $L_{i,j}$, is a set of tuples of the form $(A, \gamma, A', \gamma', p, q)$ where p and q are either integers between i and j, or $i = j = -$. The number of possible values for A, A', γ, and γ' are each bounded by a constant. Thus the number of tuples in $L_{i,j}$ is at most $O((j-i)^2)$. For a fixed value of i, j, k, steps 1a and 1b will attempt to place at most $O((j-i)^2)$ tuples in $L_{i,j}$. Before adding any tuple to $L_{i,j}$ we first check whether the tuple is already present in that array element. This can be done in constant time on a RAM by assuming that each array element $L_{i,j}$ is itself an $(i+1) \times (j+1)$ array. A tuple of the form $(A, \gamma, A', \gamma', p, q)$ will be in the $\langle p, q \rangle^{th}$ element of $L_{i,j}$ and a tuple of the form $(A, -, -, -, -, -)$ will be in the $\langle i+1, j+1 \rangle^{th}$ element of $L_{i,j}$. Thus these steps take at most $O((j-i)^2)$ time. Similarly, for a fixed value of i, j, and k, steps 2a and 2b can add at most $O((j-i)^2)$ distinct tuples. However, in these steps $O((j-i)^4)$ not necessarily distinct tuples may be considered. There are $O((j-i)^4)$ such tuples because the integers p, q, r, s can take values in the range between i and j. Thus steps 2a and 2b may each take $O((j-i)^4)$ time for a fixed value of i, j, k. Since we have three initial loops for i, j, and k, the time complexity of the algorithm is $O(n^7)$ where the length of the input is n.

3 Combinatory Categorial Grammars

CCG [9,8] is an extension 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 3.1 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 elements of $V_T \cup \{\epsilon\}$ to finite subsets of $C(V_N)$, the set of categories,[3] where $C(V_N)$ is the smallest set such that $V_N \subseteq C(V_N)$ and $c_1, c_2 \in C(V_N)$ implies $(c_1/c_2), (c_1 \backslash c_2) \in C(V_N)$,
R is a finite set of combinatory rules.

There are four types of combinatory rules involving variables $x, y, z, z_1, \ldots$ over $C(V_N)$ and where $|_i \in \{\backslash, /\}$[4].

1. forward application: $(x/y) \quad y \rightarrow x$

2. backward application: $y \quad (x \backslash y) \rightarrow x$
   For these rules we say that (x/y) is the primary category and y the secondary category.

3. generalized forward composition for some fixed $n \geq 1$:

$$(x/y) \quad (\ldots(y|_1 z_1)|_2 \ldots|_n z_n) \rightarrow (\ldots(x|_1 z_1)|_2 \ldots|_n z_n)$$

[3] Note that f can assign categories to the empty string, ϵ, though, to our knowledge, this feature has not been employed in the linguistic applications of CCG.

[4] There is no type-raising rule although its effect can be achieved to a limited extent since f can assign type-raised categories to lexical items.

4. generalized backward composition for some $n \geq 1$:

$$(\ldots(y|_1 z_1)|_2 \ldots|_n z_n) \quad (x\backslash y) \rightarrow (\ldots(x|_1 z_1)|_2 \ldots|_n z_n)$$

For these rules (x/y) is the primary category and $(\ldots(y|_1 z_1)|_2 \ldots|_n z_n)$ the secondary category.

Restrictions can be associated with the use of each combinatory rule in R. These restrictions take the form of constraints on the instantiations of variables in the rules.

1. The leftmost nonterminal (**target category**) of the primary category can be restricted to be in a given subset of V_N.

2. The category to which y is instantiated can be restricted to be in a given finite subset of $C(V_N)$.

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 $\Upsilon_1, \Upsilon_2 \in (C(V_N) \cup V_T)^*$ and $c, c_1, c_2 \in C(V_N)$.

- If R contains a combinatory rule that has $c_1 c_2 \rightarrow c$ as an instance then

$$\Upsilon_1 c \Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 c_1 c_2 \Upsilon_2$$

- If $c \in f(a)$ for some $a \in V_T \cup \{\epsilon\}$ and $c \in C(V_N)$ then

$$\Upsilon_1 c \Upsilon_2 \underset{G}{\Longrightarrow} \Upsilon_1 a \Upsilon_2$$

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

In the present discussion of CCG recognition we make the following assumptions concerning the form of the grammar.

1. In order to simplify our presentation we assume that the categories are parenthesis-free. *The algorithm that we present can be adapted in a straightforward way to handle parenthesized categories and this more general algorithm is given in [12].*

2. We will assume that the function f does not assign categories to the empty string. This is consistent with the linguistic use of CCG although we have not shown that this is a normal form for CCG.

3.1 The LIG/CCG Relationship

In this section, we describe the relationship between LIG and CCG by discussing how we can construct from any CCG a weakly equivalent LIG. The weak equivalence of LIG and CCG was established in [15]. The purpose of this section is to show how a CCG recognition algorithm can be derived from the algorithm given above for LIG.

Given a CCG, $G = (V_T, V_N, S, f, R)$, we construct an equivalent LIG, $G' = (V_T, V_N, V_N \cup \{/, \backslash\}, S, P)$, as follows. Each category in $c \in C(V_N)$ can be represented in G' as a nonterminal and associated stack $A[\alpha]$ where A is the target category of c and $\alpha \in (\{/, \backslash\} V_N)^*$ such that $A\alpha = c$. Note that we are assuming that categories are parenthesis-free.

We begin by considering the function, f, which assigns categories to each element of V_T. Suppose that $c \in f(a)$ where $c \in C(V_N)$ and $a \in V_T$. We should include the production $A[\alpha] \rightarrow a$ where $c = A\alpha$ in P. For each combinatory rule in R we may include a number of productions in P. From the definition of CCG it follows that the length of all secondary categories in the rules R is bounded by some constant. Therefore there are a finite number of possible ground instantiations of the secondary category in each rule. Thus we can remove variables in secondary categories by expanding the number of rules in R. The rules that result will involve a secondary category $c \in C(V_N)$ and a primary category of the form x/A or $x\backslash A$ where $A \in V_N$ is the target category of c. The rule may also place a restriction on the value of the target category of x. In the case of the primary categories of the combinatory rules there is no bound on their length and we cannot remove the variable that will be bound to the unbounded part of the category (the variable x above). Therefore the rules contain a single variable and are linear with respect to this variable; i.e., it appears once on either side of the rule.

It is straightforward to convert combinatory rules in this form into corresponding LIG productions. We illustrate how this can be done with an example. Suppose we have the following combinatory rule.

$$x/A \quad A/B\backslash C\backslash B \rightarrow x/B\backslash C\backslash B$$

where the target category of x must be either C or D. This is converted into the following two productions in P.

$$C[\cdot\cdot/B\backslash C\backslash B] \rightarrow C[\cdot\cdot/A] \quad A[/B\backslash C\backslash B] \qquad D[\cdot\cdot/B\backslash C\backslash B] \rightarrow D[\cdot\cdot/A] \quad A[/B\backslash C\backslash B]$$

Notice that these LIG productions do not correspond precisely to our earlier definition. We are pushing and popping more that one symbol on the stack and we have not associated empty stacks with all but one of the RHS nonterminals. Although this clearly does not affect weak generative power, as we will see in the next section, it will require a modification to the recognition algorithm given earlier for LIG.

3.2 Recognition of CCG

In order to produce a CCG recognition algorithm we extend the LIG recognition algorithm given in Section 2.2. From the previous section it should be clear that the CCG and LIG algorithms will be very similar. Therefore we do not present a detailed description of the CCG algorithm. We use an array, C, with n^2 elements, $C_{i,j}$ for $1 \leq i \leq j \leq n$. The tuples in the array will have a slightly different form from those of the LIG algorithm. This is because each derivation step may depend on more than one symbol of the category (stack). The number of such symbols is bounded by the grammar and is equal to the number of symbols in the longest secondary category. We define this bound for a CCG, $G = (V_T, V_N, S, f, R)$ as follows. Let $l(c) = k$ if $c \in (\{/, \backslash\}V_N)^k$. Let $s(G)$ be the maximum $l(c)$ of any category $c \in C(V_N)$ such that c can be the secondary category of a combinatory rule in R.

As in the LIG algorithm we do not store the entire category explicitly. However, rather than storing only the top symbol locally, as in the LIG algorithm, we store some bounded number of symbols locally together with a indication of where in C the remainder of the category can be found. This modification is needed since at each step in the recognition algorithm we may have to examine the top $s(G)$ symbols of a category. Without this extension we would be required to trace through $c(G)$ entries in C in order to examine the top $c(G)$ symbols of a category and the algorithm's time complexity would increase.

An entry in C will be a six-tuple of the form $(A, \alpha, \beta, \gamma, p, q)$ where $A \in V_N$,, $\alpha, \beta \in (\{/, \backslash\} V_N)^*$ and one of the two cases applies.

$$2 \leq l(\alpha) \leq s(G) - 1, \quad l(\beta) = s(G) - 1, \quad \gamma \in \{/, \backslash\} V_N, \quad 1 \leq p \leq q \leq n$$

or

$$0 \leq l(\alpha) < 2s(G) - 2, \quad \beta = \epsilon, \quad \gamma = p = q = -$$

An entry $(A, \alpha, \beta, \gamma, p, q)$ is placed in $C_{i,j}$ when

- If $\beta = \epsilon$ and $\gamma = p = q = -$ then $A\alpha \stackrel{*}{\underset{G}{\Longrightarrow}} a_i \ldots a_j$.
- If $\beta \neq \epsilon$ then for some $\alpha' \in (\{/, \backslash\} V_N)^*$, $A\alpha'\beta\alpha \stackrel{*}{\underset{G}{\Longrightarrow}} a_i \ldots a_j$ and $A\alpha'\beta\gamma \stackrel{*}{\underset{G}{\Longrightarrow}} a_p \ldots a_q$.

The steps of the algorithm that apply for examples of forward application and forward composition are as follows.

- $x/A \quad A \rightarrow x \in R$
  For each k between i and j, we look for $(B, \alpha, \beta, \gamma, p, q) \in C_{i,k}$ and $(A, \epsilon, \epsilon, -, -, -) \in C_{k+1,j}$ where B is a possible target category of x and the string $\beta\alpha$ has $/A$ as a suffix. If we find these tuples then do the following.

  If $l(\alpha) \geq 3$ or $\beta = \epsilon$ then include $(B, \alpha', \beta, \gamma, p, q)$ in $C_{i,j}$ where $\alpha = \alpha'/A$

  If $l(\alpha) = 2$ and $\beta \neq \epsilon$ then look in $C_{p,q}$ for some $(B, \alpha', \beta', \gamma', r, s)$ such that β is a suffix of $\beta'\alpha'$, and include $(B, \alpha'''\alpha'', \beta', \gamma', r, s)$ in $C_{i,j}$ where $\alpha = \alpha''/A$ and $\alpha' = \alpha'''\gamma$.

  If $l(\alpha) = /A$ then we know that $\beta = \epsilon$ and $\gamma = p = q = -$, and we should add $(B, \epsilon, \epsilon, -, -, -)$ in $C_{i,j}$.

- $x/A \quad A\backslash B/C \rightarrow x\backslash B/C \in R$
  For each k between i and j, we look for $(A', \alpha, \beta, \gamma, p, q) \in C_{i,k}$ and $(A, \backslash B/C, \epsilon, -, -, -) \in C_{k+1,j}$ where A' is a possible target category of x and $/A$ is a suffix of $\beta\alpha$. If we find these tuples then do the following.

  If $l(\beta) = s(G) - 1$ or $l(\alpha) = 2s(G) - 3$ then include $(A', \backslash B/C, \beta', /A, i, k)$ in $C_{i,j}$ where β'/A is a suffix of $\beta\alpha$ such that $l(\beta') = s(G) - 1$.

  If $l(\beta) = 0$ and $l(\alpha) < 2s(G) - 3$) then include $(A', \backslash B/C\alpha', \epsilon, -, -, -)$ in $C_{i,j}$ where $\alpha'/A = \beta\alpha$.

Each of the other forms of combinatory rules can be treated in a similar way yielding an algorithm that closely resembles the LIG algorithm presented in Section 2.2. Note that in a complete algorithm, the forward composition example that we have considered here would have to be made more general since the number of cases that must be considered depends on the length of the secondary category in the rule. The time complexity of the full CCG recognition algorithm is the same as that of the LIG algorithm; i.e., $\mathcal{O}(n^7)$.

4 Importance of Linearity

The recognition algorithms given here have polynomial-time complexity because each array element (e.g., $L_{i,j}$ in LIG recognition) contains a polynomial number of tuples (with respect to the difference between j and i). These tuples encode the top symbol of the stack (or top symbols of the category) together with an indication of where the next part of the stack (category) can be found. If we had stored the entire stack in the array elements[5], then each array entry could include exponentially many elements. The recognition complexity would then be exponential.

It is interesting to consider why it is not necessary to store the entire stack in the array elements. Suppose that $(A, \gamma, A', \gamma', p, q) \in L_{i,j}$. This indicates the existence of a tuple, say $(A', \gamma', A'', \gamma'', r, s)$, in $L_{p,q}$. It is crucial to note that when we are adding the first tuple to $L_{i,j}$ we are not concerned about how the second tuple came to be put in $L_{p,q}$. This is because the productions in LIG (combinatory rules in CCG) are *linear* with respect to their unbounded stacks (categories). Hence the derivations from different nonterminals and their associated stacks (categories) are *independent* of each other. In Indexed Grammars, productions can have the form $A[\cdots\gamma] \rightarrow A_1[\cdots] A_2[\cdots]$. In such productions there is no single *distinguished* child that inherits the unbounded stack from the nonterminal in the LHS of the production. In a bottom-up recognition algorithm the identity of the entire stacks associated with A_1 and A_2 has to be verified. This nullifies any advantage from the sharing of stacks since we would have to examine the complete stacks. A similar situation arises in the case of coordination schema used to handle certain forms of coordination in Dutch. A coordination schema has been used by Steedman [9] that has the form $x\ conj\ x \rightarrow x$ where the variable x can be any category. With this schema we have to check the identity of two derived categories. This results in the loss of *independence* among paths in derivation trees. In [13] we have discussed the notion of independent paths in derivation trees with respect to a range of grammatical formalisms. We have shown [12] that when CCG are extended with this coordination schema the recognition problem becomes NP-complete.

5 Conclusion

We have presented a general scheme for polynomial-time recognition of languages generated by a class of grammatical formalisms that are more powerful than CFG. This class of formalisms, which includes LIG, CCG, and TAG, derives more complex trees than CFG due the use of an additional stack-manipulating mechanism. Using constructions given in [15,3], we have described how a recognition algorithm presented for LIG can be adapted to give an algorithm for CCG. These are the first polynomial recognition algorithms that work directly with these formalisms. This approach can also be used to yield TAG recognition algorithm, although the TAG algorithm is not discussed in this paper. A similar approach has been independently taken by Lang [5] who presents a Earley parser for TAG that appears to be very closely related to the algorithms presented here.

[5] In the chart parser for CCG given by Pareschi and Steedman [6] the entire category is stored explicitly in each chart entry.

References

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

[2] G. Gazdar. *Applicability of Indexed Grammars to Natural Languages.* Technical Report CSLI-85-34, Center for Study of Language and Information, 1985.

[3] A. K. Joshi, K. Vijay-Shanker, and D. J. Weir. The convergence of mildly context-sensitive grammar formalisms. In T. Wasow and P. Sells, editors, *The Processing of Linguistic Structure*, MIT Press, 1989.

[4] T. Kasami. *An Efficient Recognition and Syntax Algorithm for Context-Free Languages.* Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[5] B. Lang. *Nested Stacks and Structure Sharing in Earley Parsers.* In preparation.

[6] R. Pareschi and M. J. Steedman. A lazy way to chart-parse with categorial grammars. In 25^{th} *meeting Assoc. Comput. Ling.*, 1987.

[7] C. Pollard. *Generalized Phrase Structure Grammars, Head Grammars and Natural Language.* PhD thesis, Stanford University, 1984.

[8] M. Steedman. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, Foris, Dordrecht, 1986.

[9] M. J. Steedman. Dependency and coordination in the grammar of Dutch and English. *Language*, 61:523–568, 1985.

[10] J. W. Thatcher. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *J. Comput. Syst. Sci.*, 5:365–396, 1971.

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

[12] K. Vijay-Shanker and D. J. Weir. The computational properties of constrained grammar formalisms. In preparation.

[13] K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In 25^{th} *meeting Assoc. Comput. Ling.*, 1987.

[14] K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. Tree adjoining and head wrapping. In 11^{th} *International Conference on Comput. Ling.*, 1986.

[15] D. J. Weir and A. K. Joshi. Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems. In 26^{th} *meeting Assoc. Comput. Ling.*, 1988.

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