POLYNOMIAL TIME PARSING OF COMBINATORY CATEGORIAL GRAMMARS*

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

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

Abstract

In this paper we present a polynomial time parsing algorithm for Combinatory Categorial Grammar. The recognition phase extends the CKY algorithm for CFG. The process of generating a representation of the parse trees has two phases. Initially, a shared forest is build that encodes the set of all derivation trees for the input string. This shared forest is then pruned to remove all spurious ambiguity.

1 Introduction

Combinatory Categorial Grammar (CCG) [7, 5] is an extension of Classical Categorial Grammar in which both function composition and function application are allowed. In addition, forward and backward slashes are used to place conditions on the relative ordering of adjacent categories that are to be combined. There has been considerable interest in parsing strategies for CCG [4, 11, 8, 2]. One of the major problems that must be addressed is that of spurious ambiguity. This refers to the possibility that a CCG can generate a large number of (exponentially many) derivation trees that assign the same function argument structure to a string. In [9] we noted that a CCG can also generate exponentially many genuinely ambiguous (non-spurious) derivations. This constitutes a problem for the approaches cited above since it results in their respective algorithms taking exponential time in the worst case. The algorithm we present is the first known polynomial time parser for CCG.

The parsing process has three phases. Once the recognizer decides (in the first phase) that an input can be generated by the given CCG the set of parse trees can be extracted in the second phase. Rather than enumerating all parses, in Section 3, we describe how they can be encoded by means of a shared forest (represented as a grammar) with which an exponential number of parses are encoded using a polynomially bounded structure. This shared forest encodes all derivations including those that are spuriously ambiguous. In Section 4.1, we show that it is possible to modify the shared forest so that it contains no spurious ambiguity. This is done (in the third phase) by traversing the forest, examining two levels of nodes at each stage, detecting spurious ambiguity locally. The three stage process of recognition, building the shared forest, and eliminating spurious ambiguity takes polynomial time.

1.1 Definition of CCG

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 to finite sets of categories, R is a finite set of combinatory rules. Combinatory rules have the following form. In each of the rules $x, y, z_1, \ldots$ are variables and $|_i \in \{\backslash, /\}$.

1. Forward application: $x/y \quad y \rightarrow x$
2. Backward application: $y \quad x\backslash y \rightarrow x$
3. Forward composition (for $n \geq 1$):
   $x/y \quad y|_1 z_1|_2 \ldots|_n z_n \rightarrow x|_1 z_1|_2 \ldots|_n z_n$
4. Backward composition (for $n \geq 1$):
   $y|_1 z_1|_2 \ldots|_n z_n \quad x\backslash y \rightarrow x|_1 z_1|_2 \ldots|_n z_n$

In the above rules, $x \mid y$ is the primary category and the other left-hand-side category is the secondary category. Also, we refer to the leftmost nonterminal

*This work was partially supported by NSF grant IRI-8909810. We are very grateful to Aravind Joshi, Michael Niv, Mark Steedman and Kent Wittenburg for helpful discussions.

of a category as the *target* of the category. We assume that categories are parenthesis-free. The results presented here, however, generalize to the case of fully parenthesized categories. The version of CCG used in [7, 5] allows for the possibility that the use of these combinatory rules can be restricted. Such restrictions limit the possible categories that can instantiate the variables. We do not consider this possibility here, though the results we present can be extended to handle these restrictions.

Derivations in a CCG involve the use of the combinatory rules in R. Let $\Longrightarrow$ be defined as follows, where Υ_1 and Υ_2 are strings of categories and terminals and c, c_1, c_2 are categories.

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

The string language generated is defined as $L(G) = \{ w \mid S \stackrel{*}{\Longrightarrow} w \mid w \in V_T^* \}$.

1.2 Context-Free Paths

In Section 2 we describe a recognition algorithm that involves extending the CKY algorithm for CFG. The differences between the CKY algorithm and the one presented here result from the fact that the derivation tree sets of CCG have more complicated path sets than the (regular) path sets of CFG tree sets. Consider the set of CCG derivation trees of the form shown in Figure 1 for the language $\{ ww \mid w \in \{a, b\}^* \}$.

Due to the nature of the combinatory rules, categories behave rather like stacks since their arguments are manipulated in a last-in-first-out fashion. This has the effect that the paths can exhibit nested dependencies as shown in Figure 1. Informally, we say that CCG tree sets have context-free paths. Note that the tree sets of CFG have regular paths and cannot produce such tree sets.

2 Recognition of CCG

The recognition algorithm uses a 4 dimensional array L for the input $a_1 \ldots a_n$. In entries of the array L we cannot store complete categories since exponentially many categories can derive the substring

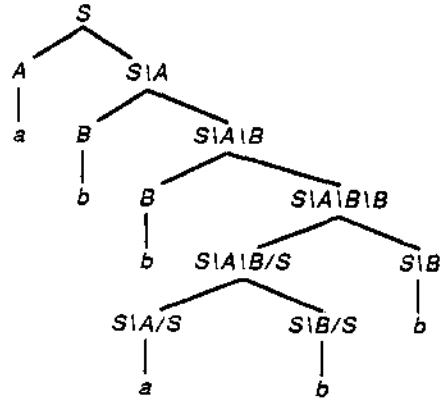

Figure 1: Trees with context-free paths

$a_i \ldots a_j$[1] it is necessary to store categories carefully It is possible, however, to share parts of categories between different entries in L. This follows from the fac that the use of a combinatory rule depends only on (1) the target category of the primary category of th rule; (2) the first argument (suffix of length 1) of th primary category of the rule;(3) the entire (bounded secondary category. Therefore, we need only find thi (bounded) information in each array entry in orde to determine whether a rule can be used. Entries o the form $((A, \alpha), T)$ are stored in $L[i, j][p, q]$. This en codes all categories whose target is A, suffix α, an that derive the $a_i \ldots a_j$. The tail T and the indices p and q are used to locate the remaining part of these categories. Before describing precisely the information that is stored in L we give some definitions.

If $\alpha \in (\{\backslash, /\} V_N)^n$ then $|\alpha| = n$. Given a CCG, $G = (V_T, V_N, S, f, R)$ let k_1 be the largest n such that R contains a rule whose secondary category is $y|_1 z_1|_2 \ldots |_n z_n$ and let k_2 be the maximum of k_1 and all n where there is some $c \in f(a)$ such that $c = A\alpha$ and $|\alpha| = n$.

In considering how categories that are derived in the course of a derivation should be stored we have two cases.

1. Categories that are either introduced by lexical

[1] This is possible since the length of the category can be linear with respect to $j - i$. Since previous approaches to CCG parsing store entire categories they can take exponential time.

items appearing in the input string or whose length is less that k_1 and could therefore be secondary categories of a rule. Thus all categories whose length is bound by k_2 are encoded in their entirety within a single array entry.

2. All other categories are encoded with a sharing mechanism in which we store up to k_1 arguments locally together with an indication of where the remaining arguments can be found.

Next, we give a proposition that characterizes when an entry is included in the array by the algorithm. An entry $(A,\alpha),T) \in L[i,j][p,q]$ where $A \in V_N$ and $\alpha \in (\{\backslash,/\}V_N)^*$ when one of the following holds.

If $T = \gamma$ then $\gamma \in \{\backslash,/\}V_N$, $1 \leq |\alpha| \leq k_1$, and for some $\alpha' \in (\{\backslash,/\}V_N)^*$ the following hold

(1) $A\alpha'\alpha \stackrel{*}{\Longrightarrow} a_i \ldots a_{p-1} A\alpha'\gamma a_{q+1} \ldots a_j$.

(2) $A\alpha'\gamma \stackrel{*}{\Longrightarrow} a_p \ldots a_q$.

(3) Informally, the category $A\alpha'\gamma$ in (1) above is "derived" from $A\alpha'\alpha$ such that there is no intervening point in the derivation before reaching $A\alpha\gamma$ at which the all of the suffix α of $A\alpha'\alpha$ has been "popped".

Alternatively, if $T = -$ then $0 \leq |\alpha| < k_1 + k_2$, $(p,q) = (0,0)$ and $A\alpha \stackrel{*}{\Longrightarrow} a_i \ldots a_j$. Note that we have $|\alpha| < k_1 + k_2$ rather than $|\alpha| \leq k_2$ (as might have been expected from the discussion above). This is the case because a category whose length is strictly less than k_2, can, as a result of function composition, result in a category of length $< k_1 + k_2$. Given the way that we have designed the algorithm below, the latter category is stored in this (non-sharing) form.

2.1 Algorithm

If $c \in f(a_i)$ for some category c, such that $c = A\alpha$, then include the tuple $((A,\alpha),-)$ in $L[i,i][0,0]$.

For some i and j, $1 \leq i < j \leq n$ consider each rule $x/y \quad y|_1 z_1 \ldots |_m z_m \rightarrow x|_1 z_1 \ldots |_m z_m$[2].
For some k, $i \leq k < j$, we look for some $((B,\beta),-) \in L[k+1,j][0,0]$, where $|\beta| = m$, (corresponding to the secondary category of the rule) and we look for $((A,\alpha/B),T) \in L[i,k][p,q]$ for some α, T, p and q (corresponding to the primary category of the rule). From these entries in L we know that for some α' $A\alpha'\alpha/B \stackrel{*}{\Longrightarrow} a_i \ldots a_k$ and $B\beta \stackrel{*}{\Longrightarrow} a_{k+1} \ldots a_j$. Thus, by the combinatory rule given above we have $A\alpha'\alpha\beta \stackrel{*}{\Longrightarrow} a_i \ldots a_j$ and we should store and encoding of the category $A\alpha'\alpha\beta$ in $L[i,j]$. This encoding depends on α', α, β, and T.

- $T = -$

If $|\alpha\beta| < k_1 + k_2$ then **(case 1a)** add $((A,\alpha\beta),-)$ to $L[i,j][0,0]$. Otherwise, **(case 1b)** add $((A,\beta),/B)$ to $L[i,j][i,k]$.

- $T \neq -$ and $m > 1$

The new category is longer than the one found in $L[i,k][p,q]$. If $\alpha \neq \epsilon$ then **(case 2a)** add $((A,\beta),/B)$ to $L[i,j][i,k]$, otherwise **(case 2b)** add $((A,\beta),T)$ to $L[i,j][p,q]$.

- $T \neq -$ and $m = 1$ **(case 3)**

The new category has the same length as the one found in $L[i,k][p,q]$. Add $((A,\alpha\beta),T)$ to $L[i,j][p,q]$.

- $T = \gamma \neq -$ and $m = 0$

The new category has the a length one less than the one found in $L[i,k][p,q]$. If $\alpha \neq \epsilon$ then **(case 4a)** add $((A,\alpha),T)$ to $L[i,j][p,q]$. Otherwise, **(case 4b)** since $\alpha = \epsilon$ we have to look for part of the category that is not stored locally in $L[i,k][p,q]$. This may be found by looking in each entry $L[p,q][r,s]$ for each $((A,\beta'\gamma),T')$. We know that either $T' = -$ or $\beta' \neq \epsilon$ and add $((A,\beta'),T')$ to $L[i,j][r,s]$. Note that for some α'', $A\alpha''\beta'\gamma \stackrel{*}{\Longrightarrow} a_p \ldots a_q$, $A\alpha''\beta'/B \stackrel{*}{\Longrightarrow} a_i \ldots a_k$, and thus by the combinatory rule above $A\alpha''\beta' \stackrel{*}{\Longrightarrow} a_i \ldots a_j$.

As in the case of CKY algorithm we should have loop statements that allow i,j to range from 1 through n such that the length of the spanned substring starts from 1 ($i = j$) and increases to n ($i = 1$ and $j = n$). When we consider placing entries in $L[i,j]$ (i.e., to detect whether a category derives $a_i \ldots a_j$) we have to consider whether there are two subconstituents (to simplify the discussion let us consider only forward combinations) which span the substrings $a_i \ldots a_k$ and $a_{k+1} \ldots a_j$. Therefore we need to consider all values for k between i through $j-1$ and consider the entries in $L[i,k][p,q]$ and $L[k+1,j][0,0]$ where $i \leq p \leq q < k$ or $p = q = 0$.

The above algorithm can be shown to run in time $O(n^7)$ where n is the length of the input. In case 4b. we have to consider all possible values for r,s between p and q. The complexity of this case dominates the complexity of the algorithm since the other cases do involve fewer variables (i.e., r and s are not involved). Case 4b takes time $O((q-p)^2)$ and with the loops for i,j,k,p,q ranging from 1 through n the time complex-

[2] Backward composition and application are treated in the same way as this rule, except that all occurrences below of i and k are swapped with occurrences of $k+1$ and j, respectively.

ity of the algorithm is $O(n^7)$.

However, this algorithm can be improved to obtain a time complexity of $O(n^6)$ by using the same method employed in [9]. This improvement is achieved by moving part of case 4b outside of the k loop, since looking for $((A, \beta'/\gamma'), T')$ in $L[p, q][r, s]$ need not be done within the k loop. The details of the improved method may be found in [9] where parsing of Linear Indexed Grammar (LIG) was considered. Note that $O(n^6)$ (which we achieve with the improved method) is the best known result for parsing Tree Adjoining Grammars, which generates the same class of languages generated by CCG and LIG.

3 Recovering All Parses

At this stage, rather than enumerating all the parses, we will encode these parses by means of a shared forest structure. The encoding of the set of all parses must be concise enough so that even an exponential number of parses can be represented by a polynomial sized shared forest. Note that this is not achieved by any previously presented shared forest presentation for CCG [8].

3.1 Representing the Shared Forest

Recently, there has been considerable interest in the use of shared forests to represent ambiguous parses in natural language processing [1, 8]. Following Billot and Lang [1], we use grammars as a representation scheme for shared forests. In our case, the grammars we produce may also be viewed as acyclic and-or graphs which is the more standard representation used for shared forests.

The grammatical formalism we use for the representation of shared forest is Linear Indexed Grammar (LIG)[3]. Like Indexed Grammars (IG), in a LIG stacks containing indices are associated with nonterminals, with the top of the stack being used to determine the set of productions that can be applied. Briefly, we define LIG as follows.

If α is a sequence of indices and γ is an index, we use the notation $A[\alpha\gamma]$ to represent the case where a stack is associated with a nonterminal A having γ on top with the remaining stack being the α. We use the following forms of productions.

$A[\cdot\alpha] \rightarrow A_1[\alpha_1] \ldots A_{i_1}[\alpha_{i-1}]\, A_i[\cdot\beta]\, A_{i+1}[\alpha_{i+1}] \ldots A_n[\alpha_n]$

$A[\alpha] \rightarrow a$

The first form of production is interpreted as: if a nonterminal A is associated with some stack with the sequence α on top (denoted $[\cdot\alpha]$), it can be rewritten such that the i^{th} child inherits this stack with β replacing α. The remaining children inherit the bounded stacks given in the production.
The second form of production indicates that if a nonterminal A has a stack containing a sequence α then it can be rewritten to a terminal symbol a.

The language generated by a LIG is the set of strings derived from the start symbol with an empty stack.

3.2 Building the Shared Forest

We start building the shared forest after the recognizer has completed the array L and decided that a given input $a_1 \ldots a_n$ is well-formed. In recovering the parses, having established that some α is in an element of L, we search other elements of L to find two categories that combine to give α. Since categories behave like stacks the use of CFG for the representation of the set of parse trees is not suitable. For our purposes the LIG formalism is appropriate since it involves stacks and production describing how a stack can be decomposed based on only its top and bottom elements.

We refer to the LIG representing the shared forest as G_{sf}. The set of indices used in G_{sf} have the form (A, α, i, j). The terminals used in G_{sf} are names for the combinatory rule or the lexical assignment used (thus derived terminal strings encode derivations in G). For example, the terminal F_m indicates the use of the forward composition rule $x/y \quad y|_1 z_1|_2 \ldots |_m z_m$ and $\langle c, a\rangle$ indicates the lexical assignment, c to the symbol a. We use one nonterminal, P.

An input $a_1 \ldots a_n$ is accepted if it is the case that $((S, \epsilon), -) \in L[1, n][0, 0]$. We start by marking this entry. By marking an entry $((A, \alpha), T) \in L[i, j][p, q]$ we are predicting that there is some derivation tree, rooted with the category S and spanning the input $a_1 \ldots a_n$, in which a category represented by this entry will participate. Therefore at some point we will have to consider this entry and build a shared forest to represent all derivations from this category.

Since we start from $((S, \epsilon), -) \in L[1, n][0, 0]$ and proceed to build a (representation of) derivation trees in a top down fashion we will have loop statements that vary the substring spanned $(a_i \ldots a_j)$ from the

[3] It has been shown in [10, 3] that LIG and CCG generate the same class of languages.

largest possible (i.e., $i = 1$ and $j = n$) to the smallest (i.e., $i = j$). Within these loop statements the algorithm (with some particular values for i and j) will consider marked entries, say $((A, \alpha), T) \in L[i, j][p, q]$ (where $i \leq p \leq q < j$ or $p = q = 0$), and will build representations of all derivations from the category (specified by the marked entry) such that the input spanned is $a_i \ldots a_j$. Since $((A, \alpha), T)$ is a representation of possibly more than one category, several cases arise depending on α and T. All these cases try to uncover the reasons why the recognizer placed this entry in $L[i, j][p, q]$. Hence the cases considered here are inverses of the cases considered in the recognition phase (and noted in the algorithm given below).

Mark $((S, \epsilon), -)$ in $L[1, n][0, 0]$.

By varying i from 1 to n, j from n to i and for all appropriate values of p and q if there is a marked entry, say $((A, \alpha), T) \in L[i, j][p, q]$ then do the following.

• *Type 1 Production* **(inverse of 1a, 3, and 4a)**
If for some k such that $i \leq k \leq j$, some α, β such that $\alpha' = \alpha\beta$, and $B \in V_N$ we have $((A, \alpha/B), T) \in L[i, k][p, q]$ and $((B, \beta), -) \in L[k + 1, j][0, 0]$ then let p be the production

$$P[\cdot\cdot(A, \alpha', i, j)] \rightarrow F_m\, P[\cdot\cdot(A, \alpha/B, i, k)]\, P[(B, \beta, k + 1, j)]$$

where $m = |\beta|$. If p is not already present in G_{sf} then add p and mark $((A, \alpha/B), T) \in L[i, k][p, q]$ as well as $((B, \beta), -) \in L[k + 1, j][0, 0]$.

• *Type 2 Production* **(inverse of 1b and 2a)**
If for some k such that $i \leq k \leq j$, and α, B, T', r, s, k we have $((A, \alpha/B), T') \in L[i, k][r, s]$ where $(p, q) = (i, k)$, $((B, \alpha'), -) \in L[k + 1, j][0, 0]$, $T = /B$, and the lengths of α and α' meet the requirements on the corresponding strings in case 1b and 2a of the recognition algorithm then then let p be the production

$$P[\cdot\cdot(A, \alpha/B, i, k)(A, \alpha', i, j)] \rightarrow F_m\, P[\cdot\cdot(A, \alpha/B, i, k)]\, P[(B, \alpha', k + 1, j)]$$

where $m = |\alpha'|$. If p is not already present in G_{sf} then add p and mark $((A, \alpha/B), T') \in L[i, k][r, s]$ and $((B, \alpha'), -) \in L[k + 1, j][0, 0]$.

• *Type 3 Production* **(inverse of 2b)**
If for some k such that $i \leq k \leq j$, and some B it is the case that $((A, /B), T) \in L[i, k][p, q]$ and $((B, \alpha'), -) \in L[k + 1, j][0, 0]$ where $|\alpha'| > 1$ then then let p be the production

$$P[\cdot\cdot(A, \alpha', i, j)] \rightarrow F_m\, P[\cdot\cdot(A, /B, i, k)]\, P[(B, \alpha', k + 1, j)]$$

where $m = |\alpha'|$. If p is not already present in G_{sf} then add p and mark $((A, /B), T) \in L[i, k][p, q]$ and $((B, \alpha'), -) \in L[k + 1, j][0, 0]$.

• *Type 4 Production* **(inverse of 4b)**
If for some k such that $i \leq k \leq j$, and some B, γ', r, s, k, we find $((A, /B,), \gamma') \in L[i, k][r, s]$, $((A, \alpha'\gamma'), T) \in L[r, s][p, q]$, and $((B, \epsilon), -) \in L[k + 1, j][0, 0]$ then then let p be the production

$$P[\cdot\cdot(A, \alpha', i, j)] \rightarrow F_0\, P[\cdot\cdot(A, \alpha'\gamma', r, s)(A, /B, i, k)]\, P[(B, \epsilon, k + 1, j)]$$

If p is not already present in G_{sf} then add p and mark $((A, /B), \gamma') \in L[i, k][r, s]$ and $((B, \epsilon), -) \in L[k + 1, j][0, 0]$.

• *Type 5 Production*
If $j = i$, then it must be the case that $T = -$ and there is a lexical assignment assigning the category $A\alpha'$ to the input symbol given by a_i. Therefore, if it has not already been included, output the production

$$P[(A, \alpha', i, i)] \rightarrow \langle A\alpha, a_i \rangle$$

The number of terminals and nonterminals in the grammar is bounded by a constant. The number of indices and the number of productions in G_{sf} are $O(n^5)$. Hence the shared forest representation we build is polynomial with respect to the length of the input, n, despite the fact that the number of derivations trees could be exponential.

We will now informally argue that G_{sf} can be built in time $O(n^7)$. Suppose an entry $((A, \alpha'), T)$ is in $L[i, j][p, q]$ indicating that for some β the category $A\beta\alpha'$ dominates the substring $a_i \ldots a_j$. The method outlined above will build a shared forest structure to represent all such derivations. In particular, we will start by considering a production whose left hand side is given by $P[\cdot\cdot(A, \alpha', i, j)]$. It is clear that an introduction of production of type 4 dominates the time complexity since this case involves three other variables (over input positions), i.e., r, s, k; whereas the introduction of other types of production involve only one new variable k. Since we have to consider all possible values for r, s, k within the range i through j, this step will take $O((j - i)^3)$ time. With the outer loops for i, j, p, and q allowing these indices to range from 1 through n, the time taken by the algorithm is $O(n^7)$.

Since the algorithm given here for building the shared forest simply finds the inverses of moves made in the recognition phase we could have modified the recognition algorithm so as to output appropriate G_{sf} productions during the process of recognition without altering the asymptotic complexity of the recognizer. However this will cause the introduction of useless productions, i.e., those that describe subderivations which do not partake in any derivation from the category S spanning the entire input string $a_1 \ldots a_n$.

4 Spurious Ambiguity

We say that a given CCG, G, exhibits spurious ambiguity if there are two distinct derivation trees for a string w that assign the same function argument structure. Two well-known sources of such ambiguity in CCG result from type raising and the associativity of composition. Much attention has been given to the latter form of spurious ambiguity and this is the one that we will focus on in this paper.

To illustrate the problem, consider the following string of categories.

$$A_1/A_2 \quad A_2/A_3 \quad \ldots \quad A_{n-1}/A_n$$

Any pair of adjacent categories can be combined using a composition rule. The number of such derivations is given by the Catalan series and is therefore exponential in n. We return a single representative of the class of equivalent derivation trees (arbitrarily chosen to be the right branching tree in the later discussion).

4.1 Dealing with Spurious Ambiguity

We have discussed how the shared forest representation, G_{sf}, is built from the contents of array L. The recognition algorithm does not consider whether some of the derivations built are *spuriously* equivalent and this is reflected in G_{sf}. We show how productions of G_{sf} can be marked to eliminate spuriously ambiguous derivations. Let us call this new grammar G_{ns}. As stated earlier, we are only interested in detecting spuriously equivalent derivations arising from the associativity of composition. Consider the example involving spurious ambiguity shown in Figure 2. This example illustrates the general form of spurious ambiguity (due to associativity of composition) in the derivation of a string made up of contiguous substrings $a_{i_1} \ldots a_{j_1}$, $a_{i_2} \ldots a_{j_2}$, and $a_{i_3} \ldots a_{j_3}$ resulting in a category $A_1\alpha_1\alpha_2\alpha_3$. For the sake of simplicity we assume that each combination indicated is a forward combination and hence $i_2 = j_1 + 1$ and $i_3 = j_2 + 1$.

Each of the 4 combinations that occur in the above figure arises due to the use of a combinatory rule, and hence will be specified in G_{sf} by a production. For example, it is possible for combination 1 to be represented by the following type 1 production.

$$P[\cdot\cdot(A_1, \alpha'\alpha_2/A_3, i_1, j_2)] \rightarrow F_m\, P[\cdot\cdot(A_1, \alpha'/A_2, i_1, j_1)]\, P[(A_2, \alpha_2, i_2, j_2)]$$

where $i_2 = j_1 + 1$, α' is a suffix of α_1 of length less than k_1, and $m = |\alpha_2|$. Since $A_1\alpha_1/A_3$ and $A_3\alpha_3$ are used as secondary categories, their lengths are bounded by $k_1 + 1$. Hence these categories will appear in their entirety in their representations in the G_{sf} productions. The four combinations[4] will hence be represented in G_{sf} by the productions:

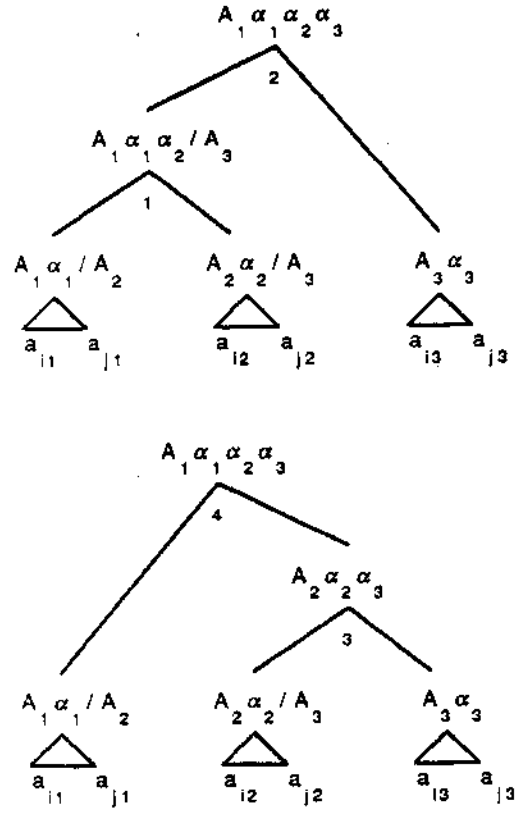

Figure 2: Example of spurious ambiguity

Combination 1: $P[\cdot\cdot(A_1, \alpha'\alpha_2/A_3, i_1, j_2)] \rightarrow F_{n_1}\, P[\cdot\cdot(A_1, \alpha'/A_2, i_1, j_1)]\, P[(A_2, \alpha_2/A_3, j_1 + 1, j_2)]$

Combination 2: $P[\cdot\cdot(A_1, \alpha'\alpha_2\alpha_3, i_1, j_3)] \rightarrow F_{n_2}\, P[\cdot\cdot(A_1, \alpha'\alpha_2/A_2, i_1, j_2)]\, P[(A_3, \alpha_3, j_2 + 1, j_3)]$

Combination 3: $P[\cdot\cdot(A_2, \alpha_2\alpha_3, j_1 + 1, j_3)] \rightarrow F_{n_2}\, P[\cdot\cdot(A_2, \alpha_2/A_3, j_1 + 1, j_2)]\, P[(A_3, \alpha_3, j_2 + 1, j_3)]$

Combination 4: $P[\cdot\cdot(A_1, \alpha'\alpha_2\alpha_3, i_1, j_3)] \rightarrow F_{n_3}\, P[\cdot\cdot(A_1, \alpha'/A_2, i_1, j_1)]\, P[(A_2, \alpha_2\alpha_3, j_1 + 1, j_3)]$

where $n_1 = |\alpha_2/A_3|$, $n_2 = |\alpha_3|$, and $n_3 = |\alpha_2\alpha_3|$.

[4] We consider the case where each combination is represented by a Type 1 production.

These productions give us sufficient information to detect spurious ambiguity locally, i.e., the local left and right branching derivations. Suppose we choose to retain the right branching derivations only. We are no longer interested in combination 2. Therefore we mark the production corresponding to this combination.

This production is not discarded at this stage because although it is marked it might still be useful in detecting more spurious ambiguity. Notice in Figure 3

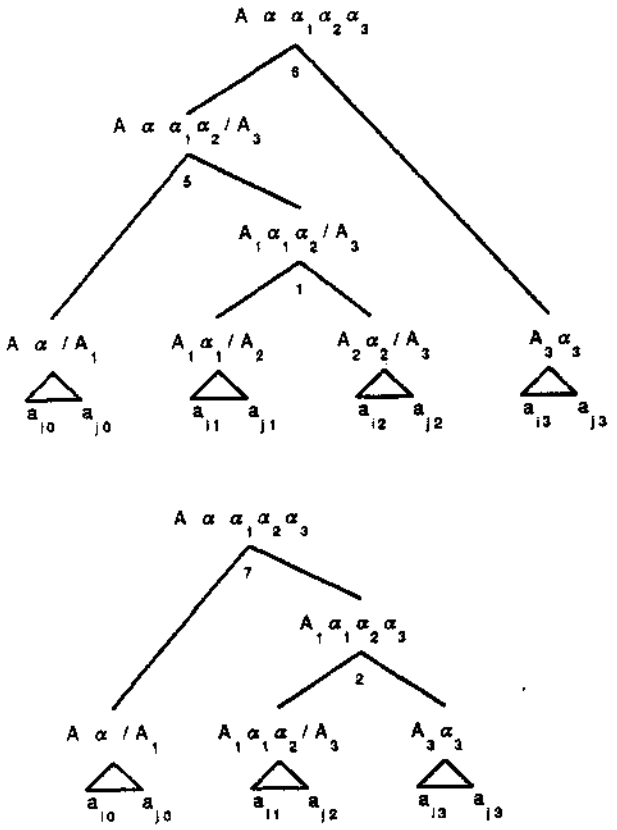

Figure 3: Reconsidering a marked production

that the subtree obtained from considering combination 5 and combination 1 is right branching whereas the entire derivation is not. Since we are looking for the presence of spurious ambiguity locally (i.e., by considering two step derivations) in order to mark this derivation we can only compare it with the derivation where combination 7 combines $A\alpha/A_1$ with $A_1\alpha_1\alpha_2\alpha_3$ (the result of combination 2)[5]. Notice we would have already marked the production corresponding to combination 2. If this production had been discarded then the required comparison could not have been made and the production due to combination 6 can not have been marked. At the end of the marking process all marked productions can be discarded[6].

In the procedure to build the grammar G_{ns} we start with the productions for lexical assignments (type 5). By varying i_1 from n to 1, j_3 from $i+2$ to n, i_2 from j_3 to i_1+1, and i_3 from i_2+1 to j_3 we look for a group of four productions (as discussed above) that *locally* indicates the the presence of spurious ambiguity. Productions involved in derivations that are not right branching are marked.

It can be shown that this local marking of spurious derivations will eliminate all and only the spuriously ambiguous derivations. That is, enumerating all derivations using unmarked productions, will give all and only genuine derivations. If there are two derivations that are spuriously ambiguous (due to the associativity of composition) then in these derivations there must be at least one occurrence of subderivations of the nature depicted in Figure 3. This will result in the marking of appropriate productions and hence the spurious ambiguity will be detected. By induction it is also possible to show that only the spuriously ambiguous derivations will be detected by the marking process outlined above.

5 Conclusions

Several parsing strategies for CCG have been given recently (e.g., [4, 11, 2, 8]). These approaches have concentrated on coping with ambiguity in CCG derivations. Unfortunately these parsers can take exponential time. They do not take into account the fact that categories spanning a substring of the input could be of a length that is linearly proportional to the length of the input spanned and hence exponential in number. We adopt a new strategy that runs in polynomial time. We take advantage of the fact that regardless of the length of the category only a bounded amount of information (at the beginning and end of the cate-

[5] Although this category is also the result of combination 4, the tree with combinations 5 and 6 can not be compared with the tree having the combinations 7 and 4.

[6] Steedman [8] has noted that although all multiple derivations arising due to the so-called spurious ambiguity yield the same "semantics" they need not be considered useless.

gory) is used in determining when a combinatory rule can apply.

We have also given an algorithm that builds a shared forest encoding the set of all derivations for a given input. Previous work on the use of shared forest structures [1] has focussed on those appropriate for context-free grammars (whose derivation trees have regular path sets). Due to the nature of the CCG derivation process and the degree of ambiguity possible this form of shared forest structures is not appropriate for CCG. We have proposed a shared forest representation that is useful for CCG and other formalisms (such as Tree Adjoining Grammars) used in computational linguistics that share the property of producing trees with context free paths.

Finally, we show the shared forest can be marked so that during the process of enumerating all parses we do not list two derivations that are *spuriously* ambiguous. In order to be able to eliminate spurious ambiguity problem in polynomial time, we examine two step derivations to locally identify when they are equivalent rather than looking at the entire derivation trees. This method was first considered by [2] where this strategy was applied in the recognition phase.

The present algorithm removes spurious ambiguity in a separate phase after recognition has been completed. This is a reasonable approach when a CKY-style recognition algorithm is being used (since the degree of ambiguity has no effect on recognition time). However, if a predictive (e.g., Earley-style) parser were employed then it would be advantageous to detect spurious ambiguity during the recognition phase. In a predictive parser the performance on an ambiguous input may be inferior to that on an unambiguous one. Due to the spurious ambiguity problem in CCG, even without *genuine* ambiguity, the parser's performance be poor if spurious ambiguity was not detected during recognition. CKY-style parsers are closely related to predictive parsers such as Earley's. Therefore, we believe that the techniques presented here, i.e., (1) the sharing of stacks used in recognition and in the shared forest representation and (2) the local identification of spurious ambiguity (first proposed by [2]) can be adapted for use in more practical predictive algorithms.

References

[1] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In 27^{th} *meeting Assoc. Comput. Ling.*, 1989.

[2] M. Hepple and G. Morrill. Parsing and derivational equivalence. In *European Assoc. Comput. Ling.*, 1989.

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

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

[6] M. Steedman. Parsing spoken language using combinatory grammars. In *International Workshop of Parsing Technologies*, Pittsburgh, PA, 1989.

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

[8] M. Tomita. Graph-structured stack and natural language parsing. In 26^{th} *meeting Assoc. Comput. Ling.*, 1988.

[9] K. Vijay-Shanker and D. J. Weir. The recognition of Combinatory Categorial Grammars, Linear Indexed Grammars, and Tree Adjoining Grammars. In *International Workshop of Parsing Technologies*, Pittsburgh, PA, 1989.

[10] 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.

[11] K. B. Wittenburg. Predictive combinators: a method for efficient processing of combinatory categorial grammar. In 25^{th} *meeting Assoc. Comput. Ling.*, 1987.