AMBIGUITY IN CONTEXT-FREE GRAMMARS

Bruce S. N. Cheung
The University of Hong Kong

Robert C. Uzgalis
University of Auckland

Abstract

This paper presents a solution to the CFG Ambiguity Problem which has come about in a recent automatic language acquisition research. The ambiguity problem of a grammar has not been addressed in prior language acquisition research. A theorem has told us to classify the CFG ambiguity problem as undecidable. Many researchers hence simply ignore the ambiguity problem or generate only unambiguous grammar. However, grammatical ambiguity lurks behind every language acquisition scheme. An ambiguity is undesirable because different derivation trees represent different meanings. This paper has investigated the CFG ambiguity problem. There are two significant results: First, the problem is solvable in an application in which the string length is bounded, that is most applications. Second, the ambiguity search algorithm developed can solve the PCP problem with bounded string length.

Introduction

If a system allows context-free grammars as user input, then validity checking should be done on the grammar to make sure a user is not getting himself in trouble. Many practical natural language parsers or compilers of programming languages are content to find a single interpretation for a single sentence. Ambiguity in the grammar is problematic because different derivation trees represent different meanings. Although the ambiguity problem is known to be undecidable, this paper is devoted to make as much of the problem as decidable as possible. There are few algorithms for detecting ambiguity:

(1) Eickel and Paul investigated the parsing and ambiguity problem for an ε-free context-sensitive language [1]. Their algorithm started with a given grammar and an input string. It first converted the given grammar into the Simple Chomsky Normal Form (SCNF). It then found all possible strings that directly derived the string, and this step was applied iteratively to every string obtained in previous iteration. By iterating a number of times (this number was an input parameter), ambiguity was revealed by identical strings. There were four weak points to this algorithm. First, the algorithm only told whether a given string was ambiguous. It was unable to tell whether a grammar was an ambiguous language description. Second, they defined something called critical productions in SCNF and used them as a starting point for the algorithm. Their definition of ambiguity was hence only a subset of ambiguity(derivation trees with crossing branches). Third, the algorithm required the 'number of iterations' to be pre-set. Fourth, the algorithm used complete search and this is combinatorically explosive.

(2) The development of compiler generators resulted in Yacc [5]. Yacc was a LALR(1) parser generator developed in conjunction with the programming language "C" within UNIX operating system group at Bell Laboratories, Murrey Hill. Yacc detects ambiguous LALR(1) grammars [6,7]. Nevertheless, Yacc accepts a BNF grammar. BNF is equivalent to CFG, and LALR(1) is a restricted subset of CFG. Yacc indicates 'ambiguity' when it detects reduce-shift conflicts or reduce-reduce conflicts. Unfortunately, these conflicts may be present in unambiguous BNF grammars that are not LALR(1). Therefore, the algorithm is not sound in deciding CFG ambiguity.

In this paper, I construct an algorithm which systematically searchs a CFG for ambiguity. The algorithm may fail in some situations. However, there are two significant results: First, the CFG ambiguity problem problem is solvable in an application in which the string length is bounded. Second, the search algorithm developed can solve the Post's Correspondence Problem, the PCP problem, with bounded string length.

Ambiguity Preserving Equivalence

Definition 1. (*Ambiguity Preserving Equivalence*) A grammar $G = (V, T, P, S)$ is ambiguity preserving equivalent to a grammar $G' = (V', T, P', S)$ *iff it satisfies the following two conditions:*

(i) Language Equivalence Condition: $L(G) = L(G')$.

(ii) Ambiguity Preserving Condition: For any $w \in T^*$, G is ambiguous for w iff G' is ambiguous for w. We write $G \propto w$ iff $G' \propto w$.

[8] describes an algorithm that can transform any reduced ε-free context-free grammar with no unit productions into GNF. The input CFG and resulting GNF of the algorithm are proven to be ambiguity preserving equivalence. [8] also describes an algorithm that can map any leftmost derivation in a GNF normalized grammar back into the corresponding leftmost derivation in the original grammar. These two algorithms together make it suffice to consider the ambiguity problem in GNF form. Solving the GNF ambiguity problem implies solving the CFG-ambiguity problemand vice-versa.

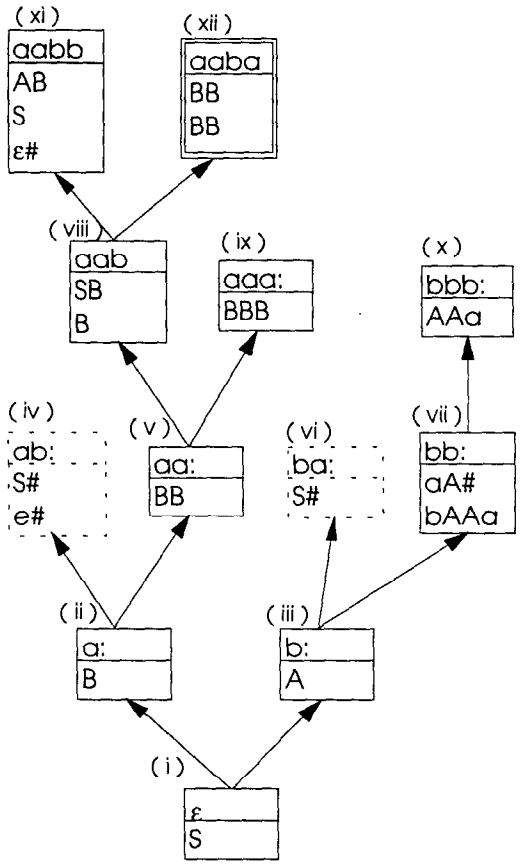
© 1995 ACM 0-89791-658-1 95 0002 3.50

Basic Idea: Searching for Ambiguity

This section presents an example of search for an ambiguity. The search is basically a breath-first search with pruning. The search procedure first generates strings by applying each GNF grammar rule once. It then iteratively checks for ambiguity, removes all trivial strings obtained, and generates further strings by applying GNF grammar rules again. Let's take the following context-free GNF grammar which generates equal numbers of a's and b's as an exampler:

G = ({S,A,B}, {a,b}, P, S) with
P = {S→aB, S→bA, A→a, A→aS, A→baA, A→bAAa,
B→b, B→bS, B→aBB}

The following gives the corresponding search tree:



The search for ambiguity starts with a the root node. In general, a node is labeled by the prefix of terminal string that has been derived. The ε appearing in the root node means no prefix has developed yet. The search starts with the initial symbol S which is called the pattern of that node. Every node has a unique label and may have one or more patterns (ps) for searching. Labels and patterns are shown in the diagram as the upper and lower parts of a node respectively. Every node is identified by its label and ps. Therefore, a node can be represented by <label , ps>. Here, the initial node is <ε, {S}>. Besides a search tree, the algorithm maintains two sets namely the expanded pattern set (eps) and the expanded compatible-primitives set (ecs). The former set stores those patterns (with prefix and suffix of terminal strings truncated) that have been expanded. The latter set keeps track of sets of compatible patterns that have been expanded. They will be described in detail later. Initially, they are empty.

The first node visited is <ε , { S }> has ps {S}. Since the pattern S has no suffix of terminal strings and S is not in the eps, the procedure expands it and enters it into eps. In general, expanding a pattern means expanding the leftmost nonterminal in that pattern. According to the productions S→aB and S→bA, expanding S gives nodes (ii) and (iii). The node <a, {B}> has ps = {B}. B has no suffix of terminal string and is not in eps. The node is expanded and B is enterned into eps. According to the productions S→b, B→bS and B→aBB, the nodes (iv) and (v) are created. The node <b, {A}> has ps {A}. Similiary, A has no suffix of terminal string and is not in eps. The productions A→baA and A→bbAAa result in prefix of terminal strings bba, and bbb. However, the prefix bb is a label of any node. The terminals after bb occur as prefix in the correspondence patterns in the ps. The node is expanded and gives nodes (vi) and (vii) and A is entered into eps.

The search continues with the node <aa, {BB}>. The pattern BB has no suffix of terminal string and is not in eps. The node is expanded and BB is entered into eps. According to the production B→b, B→bS and B→aBB, expanding the leftmost nonterminal gives new nodes (viii) and (ix) respectively. The procedure then visits the node <ab, {S, ε}>. First, S cannot derive an empty string. Second, S already exists in eps. Therefore, both patterns are terminated, and this node is closed. Similarly, the node <ba, {S}> is also closed.

Next is the node <bb, {aA, bAAa}>. First, aA and bAAa are incompatible since they have a different prefix. Second, A already exists in the eps. Therefore, the pattern aA has terminated. Third, AAa has the terminal suffix a and AA is not in the eps. The pattern bAAa results in the node (x). In the figure, the dotted box is used to denote closed node and the symbol '#' denotes terminated pattern.

The procedure then visits the node <aab, {SB, B}>. First, the patterns SB and B are called compatible primitives since they may result in two identical strings. Second, the set {SB, B} are not yet present in ecs. Both the patterns SB and B are expanded even though B is already in eps. The set {SB, B} enters ecs. The expanding results in two new nodes (xi) and (xii). The ps of the latter node contains two BB's, thus the grammar is ambiguous. By tracing the paths obtained in the search tree, the string aabaBB has the following two distinct leftmost derivations.

$$S \Rightarrow_{lm} aB \Rightarrow_{lm} aaBB \Rightarrow_{lm} aabSB \Rightarrow_{lm} aabaBB$$
$$S \Rightarrow_{lm} aB \Rightarrow_{lm} aaBB \Rightarrow_{lm} aabB \Rightarrow_{lm} aabaBB$$

The next section formally describes the ambiguity search algorithm.

The Ambiguity Search Procedure

Definition 2 (*Compatible Primitives*) Two patterns x and y are said to be incompatible iff they either conflict in their prefixes, or their suffixes, or both. Two patterns' prefixes are said to be in conflict iff:

(1) $\chi = \alpha A \beta_1$ and $y = \alpha B \beta_2$ where $\alpha \in T^*$, A and $B \in V$, β_1 and $\beta_2 \in (V \cup T)^*$, and $First(A) \cap First(B)$; or

(2) $\chi = \alpha b_1 \beta_1$ and $y = \alpha b_2 \beta_2$ where $\alpha \in T^*$, b_1 and $b_2 \in T$, β_1 and $\beta_2 \in (V \cup T)^*$, and $b_1 \neq b_2$; or

(3) $\chi = \alpha a \beta_1$ and $y = \alpha A \beta_2$ where $\alpha \in T^*$, $a \in T$, $A \in T$, β_1 and $\beta_2 \in (V \cup T)^*$, and $a \notin First(A)$.

Two patterns' suffixes are said to be in conflict iff:

(1) $\chi = \alpha_1 b_1 \beta$ and $y = \alpha_2 b_2 \beta$ where $\beta \in T^*$, b_1 and $b_2 \in T$, α_1 and $\alpha_2 \in (V \cup T)^*$, and $b_1 \neq b_2$.

Two patterns that are not incompatible are said to be compatible. Two compatible patterns are said to be compatible primitives if they do not have any common prefixes and suffixes except the null-string. Given any two compatible patterns $\{p_1 , p_2\}$, truncating all their common prefixes and suffixes will result in two compatible primitives $\{p_1^*, p_2^*\}$.

Definition 3 (*Terminated Pattern*) A pattern p in the ps of a search tree node is said to be terminated iff

(1) The pattern $p = \varepsilon$; or

(2) The corresponding nonterminal bounded substring p', which is the substring starting from its first non-terminal to the last non-terminal, is present in eps and either:

(a) p is the only pattern in this ps; or

(b) p is incompatible with all other patterns in this ps; or

(c) p is compatible with patterns $p_1, p_2, \ldots , p_n$ in this ps, and every corresponding compatible primitive pairs $\{p^* , p_r^*\}$ is a member of the ecs.

Definition 4 (*Closed Node*) A search tree node is said to be closed iff every pattern in the ps of that node is terminated.

The following procedure takes a GNF as input, searchs for an ambiguous string and then stops. It calls the procedure prune to prune closed nodes and terminated patterns that will never result in ambiguity; and calls the procedure expand to expand a node.

```
Procedure search ( grammar : list of production);
 var
   tree: search_tree;
 begin
    tree := <ε, {S}>;
    while not ((ambiguity found) or (no unvisit leaf)) do
    begin
      Choose an unvisit leaf with shortest label; prune(leaf) ;
      if not closed leaf then
      begin
        expand ( leaf, grammar );
        for each node affected by expand do
            if node.ps has same patterns then ambiguity found
      end
    end
 end
```

According to definition 4, if all patterns associated with a node are terminated, then the node is closed. The procedure prune determines what patterns associating with a tree node are terminated and whether the node is closed. It prunes the terminated patterns and closed nodes to reduce the searching space. The following procedure prune is simply a translation of definition 4.

```
procedure prune ( node: search_tree );
   begin
   for each pattern in node.ps do
      if pattern is terminated then
         mark pattern terminated;
   if every pattern in node.ps is terminated then
      mark node closed
  end
```

The condition 'pattern is terminated' in prune is simply a translation of definition 3. The following procedure expands a node. Assume a pattern is a string, it makes use the strings functions like: head which returns the first symbol of a string, tail which returns a new string by eliminating the first symbol from a string, which returns the concatenation of two strings, and str which converts a character into a single character string.

```
procedure expand (node: search_tree; grammar: list of
                  production);
   var
      c: element_type ;
      x, y, z: pattern;
  begin
  for each non-terminated pattern x in node.ps do
     begin
     c := head ( x );
     y := tail ( x );
     if c is a terminal then
        enter y into the ps of the node labeled (node.label ⊕
        str(c)) {create new node if not exist}
     else
       begin
         for each rule with the form c➔aα do
             enter (α ⊕ y) into the ps of the node labeled
             (node.label ⊕ str(a)) {create new one if not exist};
         enter into eps the nonterminal_bounded_substring x';
         {the substring starting from its first non-terminal to
          the last non-terminal}
         for each compatible pattern z_i in node do
           enter compatible primitive pairs {x*,z_i*} into ecs
       end
     end
  end
```

The described ambiguity search algorithm was proved to be sound and complete in [8]. Together with the Ambiguity Preserving Equivalence property of the GNF normalization algorithm developed in [8], all CFG with finite length ambiguous strings are provable by the ambiguity search algorithm.

Undecidability of CFG Ambiguity Problem

The CFG ambiguity problem is undecidable. Every algorithm which tries to solve that problem must run forever for some situation. Therefore, it is crucial to restrict unbounded loops to uninteresting cases.

It is trivial that every finite context-free grammar has only finite length compatible primitives. Regular grammars also have only finite length compatible primitives because left-regular grammar and right-regular grammar are one-one correspondence and for every left-regular grammar all patterns in a ps of a search tree node are shorter than the length of its longest production. Therefore, there are only finite number of distinct compatible primitives and unbounded loops are impossible.

Next linear grammars are considered. Although a linear grammar is simple, the linear grammar ambiguity problem is unsolvable. This can be shown by using the PCP-Normal Form grammar as a counter example. A PCP-Normal Form grammar is the grammar derived from a Post-Correspondence Problem (thereafter called PCP) using the algorithm described in [4]. The PCP is formulated as follows:

Given an alphabet X, where |X| is greater than 2, and two arbitrary l -tuples (u1, u2, . . . , ul) and (v1, v2, . . . , vl) of nonempty strings over X, decide whether or not there exists a nonempty set of subscripts { i1, i2, . . . , ik} where the i's are in the range of 1 to l.; $u_{i1}\, u_{i2} \ldots u_{ik} = v_{i1}\, v_{i2} \ldots v_{ik}$. The PCP-Normal Form grammar has the following forms in its productions:

(i) $S \rightarrow S_u$ and $S \rightarrow S_v$;
(ii) $S \rightarrow u_{iw}\, S_u a_{iw}$ and $S \rightarrow u_{iw} a_{iw}$, where $a_{iw} \notin X$; and
(ii) $S \rightarrow v_{iw}\, S_v a_{iw}$ and $S \rightarrow v_{iw} a_{iw}$.

The PCP can be reduced to the ambiguity problem in the form of this restricted linear grammar. In order to cause unbounded loops in the search algorithm, there must be unlimited compatible primitive pairs because all duplicated instances of a compatible primitive pairs will be pruned. The pruning will result in a finite tree if only a finite number of compatible primitive pairs exist. This implies that the length of some compatible primitives must be unbounded. The unsolvability of the PCP ambiguity problem derives from unbounded compatible primitives. What causes unbounded compatible primitives in a PCP-Normal Form grammar? For every PCP-Normal Form grammar, there is only one nonterminal appearing in the right-hand side of a production (either a single S_u or a single S_v). In the compatible primitive pairs of a PCP-Normal Form grammar, at least one primitive has a nonterminal as the first (last) symbol since the common prefix (suffix) has been truncated. This is also true for any linear grammar. The terminal prefix in a compatible primitive is bounded and must not be longer than the longest u_{iw} or v_{iw} (or in case of a linear grammar, the longest terminal prefix appearing in the productions). Therefore, unbounded compatible primitives must come from the unbounded suffix a_{iw}'s. Since there are only finite number of bounded terminal prefix pairs (and at least one of them is empty) in those compatible primitives during searching, there must be unlimited number of terminal suffix pairs (and at least one of them is empty) associated with certain terminal prefix pairs. That is, after a number of node expands, the same terminal prefix pairs are obtained in some compatible primitives but the terminal suffix pairs varied (the length is unbounded). It is not possible to predict exactly what terminal prefix pairs will result in unbounded suffix pairs. At best, it is only possible to show that for any PCP Normal Form Grammar the number of compatible primitives per search tree level will never increase after some time.

The a_i's in a PCP-Normal Form grammar keep track of the pairs being used. If the terminal suffix pairs of a compatible primitive pairs are not both empty, the branching factor will be two because the in the non-empty terminal suffix tells which two rules to apply. Moreover, one of them will result in a string without a nonterminal. Therefore, the branching factor will effectively only be one. If both terminal suffixes of a compatible primitive are empty and one primitive has the leading symbol b, the branching factor will be the number of rules with b as a leading symbol. The prefix pairs of the compatible primitive pairs are bounded and there are less than $(2|X|^n)$ of them, where |X| is the number of symbols in the alphabet and n is the maximum string length of the PCP pairs. Moreover, only a small proportion of the $(2|X|^n)$ primitive PCP pairs will be encountered because most are unreachable. After all reachable compatible primitives with empty terminal suffixes have been visited, the total number of compatible primitives per search tree level will never increase because the branching factor of the remaining compatible primitives will only be one.

In a non-linear context-free grammar, there may be more than one nonterminal appearing in the right-hand side of productions. The compatible primitive pairs may be unbounded even though both the prefix and suffix are bounded. It is more difficult to have bounded compatible primitives during the searching. The states that may lead to unbounded loops are even more difficult to determine. Fortunately, in practice, there is usually heuristic information. For example, when the maximum length of the sentences to be generated/parsed by the grammar is known, the number of levels of the tree to be expanded can be pre-set and this prevents unbounded compatible primitives.

Pruning and Efficiency

In an application, it is crucial to make sure that the search algorithm can efficiently find the ambiguous strings. In [8], a worst case study (assume there is not any heuristic rule) shows that the upper bounded for solving the bounded GNF ambiguity problem using the GNF ambiguity search algorithm is $O((n+r)!/(n(r!)))$, given that n is the maximum string length and r is the number of GNF rules.

In the ambiguity search procedure, the following techniques improve the efficiency of the implementation:

(i) The procedure prune eliminates terminated patterns that have already been expanded and will never result in ambiguity. The patterns being pruned are those that have been expanded and are singleton in a node; or those that have been expanded and are incompatible with other patterns in the same node; or those that have been expanded and are merely compatible with patterns which have been expanded together before.

(ii) By computing the transitive closure of all the nonterminals, the compatibility of two patterns are be checked efficiently. The terminals that a nonterminal can father are computed respectively. If one pattern contains terminals that cannot be derived by the second pattern, then the two nonterminals must be incompatible. This step is found useful in pruning flat grammars with a large number of terminals.

(iii) Since frequently the procedure checks whether a certain pattern occurs in the set eps or ecs. Fast hashing is employed to speed up search the eps and ecs.

(iv) Halting conditions are explicitly input as a parameter into the search procedure to prevent unbounded loops. For example, the maximum level number to be expanded.

The following table shows the results of finding all ambiguous patterns of some grammars. The search tree was expanded up to level 10. The second and third columns of the table give the grammar rules, the numbers of patterns being expanded with and without pruning respectively.

A→A b A→B A B→B A A→a B→b	110	220
S→a B S→b A A→a A→a S A→b a A A→b b A A A B→b B→b S B→a B B	167	1975
S→i S e S S→i S S→a	66	435
E→E E E→E E E→ (E) E→identifier	717	1748
S→S 1 S→S2 S→S 2 S 1→a b c S 1 a 1 S 1→a b c a 1 S 1→d e S 1 a 2 S 1→d e a 2 S 1→f S 1 a 3 S 1→f a 3 S 2→a S 2 a 1 S 2→a a 1 S 2→b c d S 2 a 2 S 2→b c d a 2 S 2→e f S 2 a 3 S 2→e f a 3	83	3658

Results and Directions

This paper has investigated the ambiguity problem of context-free grammar. While it is true that detecting the ambiguity of a CFG grammar is undecidable, what is more important is why it is undecidable. The fundamental idea behind those proofs is that the strings described by a grammar can be of arbitrary length. But the problem is decidable in any application in which the maximum string length is bounded. The search algorithm created here solves all application cases with a known maximum string length.

The pruning procedure adopted reduces the size of search space significantly. Further investigation may find more powerful heuristics to speed searching.

A by-product of the ambiguity search algorithm is an algorithm to 'solve' the PCP problem. The last two examples shown in table 6.1 show that there are a big differences between the number of patterns expanded with pruning and the number of patterns expanded without pruning. Moreover, it is found that the number of compatible primitives per search tree level will not increase after all reachable compatible primitives with empty terminal suffixes have been visited; there must be less than $(2|X|^n)$ of them, where $|X|$ is the number of symbols in the alphabet and n is the maximum string length of the PCP pairs.

References:

[1] J. Eickel and M. Paul, "The Parsing and Ambiguity Problem for Chomsky-Languages." In T.B. Steel, Jr. (Ed.), Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, Amsterdam, London, 1966.

[2] S. Gorn, "Explicit Definitions and Linguistic Dominoes." In J. Hart and S. Takasu (Eds.), Systems and Computer Science, University of Toronto Press, Toronto, Canada, 1965.

[3] S.A. Greibach, "A New Normal Form Theorem for Context-free Phrase Structure Grammars," J. ACM 12 : 1, 42 52, 1965.

[4] J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Mass., 1979.

[5] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler." In B.W. Kernighan and M.D. Mellroy, UNIX Programmer's Manual, Bell Laboratories, 1978. Seven Edition.

[6] A.T. Schreiner and H.G. Friedman, Jr., Introduction to Compiler Construction with UNIX, Prentice-Hall, Inc.,Englewood Cliffs, 1985.

[7] A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1979.

[8] B.S.N. Cheung, A Theory of Automatic Language Acquisition, Ph.D. Thesis, University of Hong Kong, 1994.