Removing Left Recursion from Context-Free Grammars[*]

Robert C. Moore
Microsoft Research
One Microsoft Way
Redmond, Washington 98052
bobmoore@microsoft.com

Abstract

A long-standing issue regarding algorithms that manipulate context-free grammars (CFGs) in a "top-down" left-to-right fashion is that left recursion can lead to nontermination. An algorithm is known that transforms any CFG into an equivalent non-left-recursive CFG, but the resulting grammars are often too large for practical use. We present a new method for removing left recursion from CFGs that is both theoretically superior to the standard algorithm, and produces very compact non-left-recursive CFGs in practice.

1 Introduction

A long-standing issue regarding algorithms that manipulate context-free grammars (CFGs) in a "top-down" left-to-right fashion is that left recursion can lead to nontermination. This is most familiar in the case of top-down recursive-descent parsing (Aho et al., 1986, pp. 181–182). A more recent motivation is that off-the-shelf speech recognition systems are now available (e.g., from Nuance Communications and Microsoft) that accept CFGs as language models for constraining recognition; but as these recognizers process CFGs top-down, they also require that the CFGs used be non-left-recursive.

The source of the problem can be seen by considering a directly left-recursive grammar production such as $A \rightarrow A\alpha$. Suppose we are trying to parse, or recognize using a speech recognizer, an A at a given position in the input. If we apply this production top-down and left-to-right, our first subgoal will be to parse or recognize an A at the same input position. This immediately puts us into an infinite recursion. The same thing will happen with an indirectly left-recursive grammar, via a chain of subgoals that will lead us from the goal of parsing or recognizing an A at a given position to a descendant subgoal of parsing or recognizing an A at that position.

In theory, the restriction to non-left-recursive CFGs puts no additional constraints on the languages that can be described, because any CFG can in principle be transformed into an equivalent non-left-recursive CFG. However, the standard algorithm for carrying out this transformation (Aho et al., 1986, pp. 176–178) (Hopcroft and Ullman, 1979, p. 96)—attributed to M. C. Paull by Hopcroft and Ullman (1979, p. 106)—can produce transformed grammars that are orders of magnitude larger than the original grammars. In this paper we develop a number of improvements to Paull's algorithm, which help somewhat but do not completely solve the problem. We then go on to develop an alternative approach based on the left-corner grammar transform, which makes it possible to remove left recursion with no significant increase in size for several grammars for which Paull's original algorithm is impractical.

2 Notation and Terminology

Grammar nonterminals will be designated by "low order" upper-case letters (A, B, etc.); and terminals will be designated by lower-case letters. We will use "high order" upper-case letters (X, Y, Z) to denote single symbols that could be either terminals or nonterminals, and Greek letters to denote (possibly empty) sequences of terminals and/or nonterminals. Any production of the form $A \rightarrow \alpha$ will be said to be an *A-production*, and α will be said to be an *expansion* of A.

We will say that a symbol X is a *direct left corner* of a nonterminal A, if there is an A-production with X as the left-most symbol on the right-hand side. We define the *left-corner* relation to be the reflexive transitive closure of the direct-left-corner relation, and we define the *proper-left-corner* relation to be the transitive closure of the direct-left-corner relation. A nonterminal is *left recursive* if it is a proper left corner of itself; a nonterminal is *directly left recursive* if it is a direct left corner of itself; and a nonterminal is *indirectly left recursive* if it is left recursive, but not directly left recursive.

3 Test Grammars

We will test the algorithms considered here on three large, independently-motivated, natural-language grammars. The CT grammar[1] was compiled into a CFG from a task-specific unification grammar written for CommandTalk (Moore et al., 1997), a spoken-language interface to a military simulation system. The ATIS grammar

[*] Revised version of paper appearing in *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics, ANLP-NAACL 2000*. Revised 30 January 2001.

[1] Courtesy of John Dowding, SRI International

	Toy Grammar	CT Grammar	ATIS Grammar	PT Grammar
Grammar size	88	55,830	16,872	67,904
Terminals	40	1,032	357	47
Nonterminals	16	3,946	192	38
Productions	55	24,456	4,592	15,039
LR nonterminals	4	535	9	33
Productions for LR nonterminals	27	2,211	1,109	14,993

Table 1: Grammars used for evaluation.

was extracted from an internally generated treebank of the DARPA ATIS3 training sentences (Dahl et al., 1994). The PT grammar[2] was extracted from the Penn Treebank (Marcus et al., 1993). To these grammars we add a small "toy" grammar, simply because some of the algorithms cannot be run to completion on any of the "real" grammars within reasonable time and space bounds.

Some statistics on the test grammars are contained in Table 1. The criterion we use to judge effectiveness of the algorithms under test is the size of the resulting grammar, measured in terms of the total number of terminal and nonterminal symbols needed to express the productions of the grammar. We use a slightly nonstandard metric, counting the symbols as if, for each nonterminal, there were a single production of the form $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$. This reflects the size of files and data structures typically used to store grammars for top-down processing more accurately than counting a separate occurrence of the left-hand side for each distinct right-hand side.

It should be noted that the CT grammar has a very special property: none of the 535 left recursive nonterminals is indirectly left recursive. The grammar was designed to have this property specifically because Paull's algorithm does not handle indirect left recursion well.

It should also be noted that none of these grammars contains empty productions or cycles, which can cause problems for algorithms for removing left recursion. It is relatively easy to trasform an arbitrary CFG into an equivalent grammar which does not contain any of the probelmatical cases. In its initial form the PT grammar contained cycles, but these were removed at a cost of increasing the size of the grammar by 78 productions and 89 total symbols. No empty productions or cycles existed anywhere else in the original grammars.

4 Paull's Algorithm

Paull's algorithm for eliminating left recursion from CFGs attacks the problem by an iterative procedure for transforming indirect left recursion into direct left recursion, with a subprocedure for eliminating direct left recursion. This algorithm is perhaps more familiar to some as the first phase of the textbook algorithm for transforming CFGs to Greibach normal form (Greibach, 1965).[3] The subprocedure to eliminate direct left recursion performs the following transformation (Hopcroft and Ullman, 1979, p. 96):

> Let
>
> $$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_r$$
>
> be the set of all directly left recursive A-productions, and let
>
> $$A \rightarrow \beta_1 \mid \ldots \mid \beta_s$$
>
> be the remaining A-productions. Replace all these productions with
>
> $$A \rightarrow \beta_1 \mid \beta_1 A' \mid \ldots \mid \beta_s \mid \beta_s A',$$
>
> and
>
> $$A' \rightarrow \alpha_1 \mid \alpha_1 A' \mid \ldots \mid \alpha_s \mid \alpha_s A',$$
>
> where A' is a new nonterminal not used elsewhere in the grammar.

This transformation is embedded in the full algorithm (Aho et al., 1986, p. 177), displayed in Figure 1.

The idea of the algorithm is to eliminate left recursion by transforming the grammar so that all the direct left corners of each nonterminal strictly follow that nonterminal in a fixed total ordering, in which case, no nonterminal can be left recursive. This is accomplished by iteratively replacing direct left corners that precede a given nonterminal with all their expansions in terms of other nonterminals that are greater in the ordering, until the nonterminal has only itself and greater nonterminals as direct left corners. Any direct left recursion for that nonterminal is then eliminated by the first transformation discussed.

The difficulty with this approach is that the iterated substitutions can lead to an exponential increase in the size of the grammar. Consider the grammar consisting of the productions $A_1 \rightarrow 0 \mid 1$, plus $A_{i+1} \rightarrow A_i0 \mid A_i1$ for $1 \leq i < n$. It is easy to see that Paull's algorithm will transform the grammar so that it consists of all possible A_i-productions with a binary sequence of length i

[2] Courtesy of Eugene Charniak, Brown University

[3] This has led some readers to attribute the algorithm to Greibach, but Greibach's original method was quite different and much more complicated.

```
Assign an ordering $A_1, \ldots, A_n$ to the nonterminals of the grammar.

for $i := 1$ to $n$ do begin
    for $j := 1$ to $i - 1$ do begin
        for each production of the form $A_i \to A_j\alpha$ do begin
            remove $A_i \to A_j\alpha$ from the grammar
            for each production of the form $A_j \to \beta$ do begin
                add $A_i \to \beta\alpha$ to the grammar
            end
        end
    end
    transform the $A_i$-productions to eliminate direct left recursion
end
```

Figure 1: Paull's algorithm.

Grammar Description	Grammar Size
original toy grammar	88
PA, "best" ordering	156
PA, lexicographical ordering	970
PA, "worst" ordering	5696

Table 2: Effect of nonterminal ordering on Paull's algorithm.

on the right-hand side, for $1 \leq i \leq n$, which is exponentially larger than the original grammar. Notice that the efficiency of Paull's algorithm crucially depends on the ordering of the nonterminals. If the ordering is reversed in the grammar of this example, Paull's algorithm will make no changes, since the grammar will already satisfy the condition that all the direct left corners of each nonterminal strictly follow that nonterminal in the revised ordering. The textbook discussions of Paull's algorithm, however, are silent on this issue.

In the inner loop of Paull's algorithm, for nonterminals A_i and A_j, such that $i > j$ and A_j is a direct left corner of A_i, we replace all occurrences of A_j as a direct left corner of A_i with all possible expansions of A_j. This only contributes to elimination of left recursion from the grammar if A_i is a left-recursive nonterminal, and A_j lies on a path that makes A_i left recursive; that is, if A_i is a left corner of A_j (in addition to A_j being a left corner of A_i). We could eliminate replacements that are useless in removing left recursion if we could order the nonterminals of the grammar so that, if $i > j$ and A_j is a direct left corner of A_i, then A_i is also a left corner of A_j. We can achieve this by ordering the nonterminals in decreasing order of the number of distinct left corners they have. Since the left-corner relation is transitive, if C is a direct left corner of B, every left corner of C is also a left corner of B. In addition, since we defined the left-corner relation to be reflexive, B is a left corner of itself. Hence, if C is a direct left corner of B, it must follow B in decreasing order of number of distinct left corners, unless B is a left corner of C.

Table 2 shows the effect on Paull's algorithm of ordering the nonterminals according to decreasing number of distinct left corners, with respect to the toy grammar.[4] In the table, "best" means an ordering consistent with this constraint. Note that if a grammar has indirect left recursion, there will be multiple orderings consistent with our constraint, since indirect left recursion creates cycles in the the left-corner relation, so every nonterminal in one of these cycles will have the same set of left corners. Our "best" ordering is simply an arbitrarily chosen ordering respecting the constraint; we are unaware of any method for finding a unique best ordering, other than trying all the orderings respecting the constraint.

As a neutral comparison, we also ran the algorithm with the nonterminals ordered lexicographically. Finally, to test how bad the algorithm could be with a really poor choice of nonterminal ordering, we defined a "worst" ordering to be one with *increasing* numbers of distinct left corners. It should be noted that with either the lexicographical or worst ordering, on all of our three large grammars Paull's algorithm exceeded a cut-off of 5,000,000 grammar symbols, which we chose as being well beyond what might be considered a tolerable increase in the size of the grammar.

[4]As mentioned previously, grammar sizes are given in terms of total terminal and nonterminal symbols needed to express the grammar.

	CT Grammar	ATIS Grammar	PT Grammar
original grammar	55,830	16,872	67,904
PA	62,499	$> 5,000,000$	$> 5,000,000$
LF	54,991	11,582	37,811
LF+PA	59,797	2,004,473	$> 5,000,000$
LF+NLRG+PA	57,924	72,035	$> 5,000,000$

Table 3: Grammar size comparisons with Paull's algorithm variants

Let PA refer to Paull's algorithm with the nonterminals ordered according to decreasing number of distinct left corners. The second line of Table 3 shows the results of running PA on our three large grammars. The CT grammar increases only modestly in size, because as previously noted, it has no indirect left recursion. Thus the combinatorial phase of Paull's algorithm is never invoked, and the increase is solely due to the transformation applied to directly left-recursive productions. With the ATIS grammar and PT grammar, which do not have this special property, Paull's algorithm exceeded our cut-off, even with our best ordering of nonterminals.

Some additional optimizations of Paull's aglorithm are possible. One way to reduce the number of substitutions made by the inner loop of the algorithm is to "left factor" the grammar (Aho et al., 1986, pp. 178–179). The left-factoring transformation (LF) applies the following grammar rewrite schema repeatedly, until it is no longer applicable:

> **LF**: For each nonterminal A, let α be the longest nonempty sequence such that there is more than one grammar production of the form $A \rightarrow \alpha\beta$. Replace the set of all productions
>
> $$A \rightarrow \alpha\beta_1,\ \ldots,\ A \rightarrow \alpha\beta_n$$
>
> with the productions
>
> $$A \rightarrow \alpha A',\ A' \rightarrow \beta_1,\ \ldots,\ A' \rightarrow \beta_n,$$
>
> where A' is a new nonterminal symbol.

With left factoring, for each nonterminal A there will be only one A-production for each direct left corner of A, which will in general reduce the number of substitutions performed by the algorithm.

The effect of left factoring by itself is shown in the third line of Table 3. Left factoring actually reduces the size of all three grammars, which may be unintuitive, since left factoring necessarily increases the number of productions in the grammar. However, the transformed productions are shorter, and the grammar size as measured by total number of symbols can be smaller because common left factors are represented only once.

The result of applying PA to the left-factored grammars is shown in the fourth line of Table 3 (LF+PA). This produces a modest decrease in the size of the non-left-recursive form of the CT grammar, and brings the non-left-recursive form of the ATIS grammar under the cut-off size, but the non-left-recursive form of the PT grammar still exceeds the cut-off.

The final optimization we have developed for Paull's algorithm is to transform the grammar to combine all the non-left-recursive possibilities for each left-recursive nonterminal under a new nonterminal symbol. This transformation, which we might call "non-left-recursion grouping" (NLRG), can be defined as follows:

> **NLRG**: For each left-recursive nonterminal A, let $\alpha_1, \ldots, \alpha_n$ be all the expansions of A that do not have a left recursive nonterminal as the left most symbol. If $n > 1$, replace the set of productions
>
> $$A \rightarrow \alpha_1,\ \ldots,\ A \rightarrow \alpha_n$$
>
> with the productions
>
> $$A \rightarrow A',\ A' \rightarrow \alpha_1,\ \ldots,\ A' \rightarrow \alpha_n,$$
>
> where A' is a new nonterminal symbol.

Since all the new nonterminals introduced by this transformation will be non-left-recursive, Paull's algorithm with our best ordering will never substitute the expansions of any of these new nonterminals into the productions for any other nonterminal, which in general reduces the number of substitutions the algorithm makes. We did not empirically measure the effect on grammar size of applying the NLRG transformation by itself, but it is easy to see that it increases the grammar size by exactly two symbols for each left-recursive nonterminal to which it is applied. Thus an addition of twice the number of left-recursive nonterminals will be an upper bound on the increase in the size of the grammar, but since not every left-recursive nonterminal necessarily has more than one non-left-recursive expansion, the increase may be less than this.

The fifth line of Table 3 (LF+NLRG+PA) shows the result of applying LF, followed by NLRG, followed by PA. This produces another modest decrease in the size of the non-left-recursive form of the CT grammar and reduces the size of the non-left-recursive form of the ATIS grammar by a factor of 27.8, compared to LF+PA. The non-left-recursive form of the PT grammar remains larger than the cut-off size of 5,000,000 symbols, however.

	CT Grammar	ATIS Grammar	PT Grammar
original grammar	55,830	16,872	67,904
LF	54,991	11,582	37,811
LF+NLRG+PA	57,924	72,035	$> 5,000,000$
LC	762,576	287,649	1,567,162
LC$_{LR}$	60,556	40,660	1,498,112
LF+LC$_{LR}$	58,893	13,641	67,167
LF+NLRG+LC$_{LR}$	57,380	12,243	50,277

Table 4: Grammar size comparisons for LC transform variants

5 Left-Recursion Elimination Based on the Left-Corner Transform

An alternate approach to eliminating left-recursion is based on the left-corner (LC) grammar transform of Rosenkrantz and Lewis (1970) as presented and modified by Johnson (1998). Johnson's second form of the LC transform can be expressed as follows, with expressions of the form A–a, A–X, and A–B being new nonterminals in the transformed grammar:

1. If a terminal symbol a is a proper left corner of A in the original grammar, add $A \rightarrow aA$–a to the transformed grammar.
2. If B is a proper left corner of A and $B \rightarrow X\beta$ is a production of the original grammar, add A–$X \rightarrow \beta A$–B to the transformed grammar.
3. If X is a proper left corner of A and $A \rightarrow X\beta$ is a production of the original grammar, add A–$X \rightarrow \beta$ to the transformed grammar.

In Rosenkrantz and Lewis's original LC transform, schema 2 applied whenever B is a left corner of A, including all cases where $B = A$. In Johnson's version schema 2 applies when $B = A$ only if A is a proper left corner of itself. Johnson then introduces schema 3 handle the residual cases, without introducing instances of nonterminals of the form A–A that need to be allowed to derive the empty string.

The original purpose of the LC transform is to allow simulation of left-corner parsing by top-down parsing, but it also eliminates left recursion from any noncyclic CFG.[5] Furthermore, in the worst case, the total number of symbols in the transformed grammar cannot exceed a fixed multiple of the square of the number of symbols in the original grammar, in contrast to Paull's algorithm, which exponentiates the size of the grammar in the worst case.

Thus, we can use Johnson's version of the LC transform directly to eliminate left-recursion. Before applying this idea, however, we have one general improvement to make in the transform. Johnson notes that in his version of the LC transform, a new nonterminal of the form A–X is useless unless X is a proper left corner of A. We further note that a new nonterminal of the form A–X, as well as the original nonterminal A, is useless in the transformed grammar, unless A is either the top nonterminal of the grammar or appears on the right-hand side of an original grammar production in other than the left-most position. This can be shown by induction on the length of top-down derivations using the productions of the transformed grammar. Therefore, we will call the original nonterminals meeting this condition "retained nonterminals" and restrict the LC transform so that productions involving nonterminals of the form A–X are created only if A is a retained nonterminal.

Let LC refer to Johnson's version of the LC transform restricted to retained nonterminals. In Table 4 the first three lines repeat the previously shown sizes for our three original grammars, their left-factored form, and their non-left-recursive form using our best variant of Paull's algorithm (LF+NLRG+PA). The fourth line shows the results of applying LC to the three original grammars. Note that this produces a non-left-recursive form of the PT grammar smaller than the cut-off size, but the non-left-recursive forms of the CT and ATIS grammars are considerably larger than the most compact versions created with Paull's algorithm.

We can improve on this result by noting that, since we are interested in the LC transform only as a means of eliminating left-recursion, we can greatly reduce the size of the transformed grammars by applying the transform only to left-recursive nonterminals. More precisely, we can retain in the transformed grammar all the productions expanding non-left-recursive nonterminals of the original grammar, and for the purposes of the LC transform, we can treat non-left-recursive nonterminals as if they were terminals. Furthermore, we broaden the definition of a retained nonterminal to include all nonterminals that appear on the right-hand side of a production whose left-hand side is a non-left-recursive nonterminal. This is necessary, since such nonterminals can act as the top nonterminal of a subgrammar to which the left-corner transform is applied. With this broader definition of retained nonterminal, our revised left-corner transform is as follows:

1. If a terminal symbol or non-left-recursive nonterminal X is a proper left corner of a retained left-

[5] In the case of a cyclic CFG, the schema 2 fails to guarantee a non-left-recursive transformed grammar.

recursive nonterminal A in the original grammar, add $A \rightarrow XA\text{–}X$ to the transformed grammar.

2. If B is a left-recursive proper left corner of a retained left-recursive nonterminal A and $B \rightarrow X\beta$ is a production of the original grammar, add $A\text{–}X \rightarrow \beta A\text{–}B$ to the transformed grammar.
3. If X is a proper left corner of a retained left-recursive nonterminal A and $A \rightarrow X\beta$ is a production of the original grammar, add $A\text{–}X \rightarrow \beta$ to the transformed grammar.
4. If A is a non-left-recursive nonterminal and $A \rightarrow \beta$ is a production of the original grammar, add $A \rightarrow \beta$ to the transformed grammar.

Let LC$_{LR}$ refer to the LC transform restricted by these modifications so as to apply only to left-recursive nonterminals. The fifth line of Table 4 shows the results of applying LC$_{LR}$ to the three original grammars. LC$_{LR}$ greatly reduces the size of the non-left-recursive forms of the CT and ATIS grammars, but the size of the non-left-recursive form of the PT grammar is only slightly reduced. This is not surprising if we note from Table 1 that almost all the productions of the PT grammar are productions for left-recursive nonterminals. However, we can apply the additional transformations that we used with Paull's algorithm, to reduce the number of productions for left-recursive nonterminals before applying our modified LC transform. The effects of left factoring the grammar before applying LC$_{LR}$ (LF+LC$_{LR}$), and additionally combining non-left-recursive productions for left-recursive nonterminals between left factoring and applying LC$_{LR}$ (LF+NLRG+LC$_{LR}$), are shown in the sixth and seventh lines of Table 4.

With all optimizations applied, the non-left-recursive forms of the ATIS and PT grammars are smaller than the originals (although not smaller than the left-factored forms of these grammars), and the non-left-recursive form of the CT grammar is only slightly larger than the original. In all cases, LF+NLRG+LC$_{LR}$ produces more compact grammars than LF+NLRG+PA, the best variant of Paull's algorithm—slightly more compact in the case of the CT grammar, more compact by a factor of 5.9 in the case of the ATIS grammar, and more compact by at least two orders of magnitude in the case of the PT grammar.

6 Conclusions

We have shown that, in its textbook form, the standard algorithm for eliminating left recursion from CFGs is impractical for three diverse, independently-motivated, natural-language grammars. We apply a number of optimizations to the algorithm—most notably a novel strategy for ordering the nonterminals of the grammar—but one of the three grammars remains essentially intractable. We then explore an alternative approach based on the LC grammar transform. With several optimizations of this approach, we are able to obtain quite compact non-left-recursive forms of all three grammars. Given the diverse nature of these grammars, we conclude that our techniques based on the LC transform are likely to be applicable to a wide range of CFGs used for natural-language processing.

References

A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts.

D. A. Dahl et al. 1994. Expanding the scope of the ATIS task: the ATIS-3 corpus. In *Proceedings of the Spoken Language Technology Workshop*, pages 3–8, Plainsboro, New Jersey. Advanced Research Projects Agency.

S. A. Greibach. 1965. A new normal-form theorem for context-free phrase structure grammars. *Journal of the Association for Computing Machinery*, 12(1):42–52, January.

J. E. Hopcroft and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts.

M. Johnson. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *Proceedings, COLING-ACL '98*, pages 619–623, Montreal, Quebec, Canada. Association for Computational Linguistics.

M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, June.

R. Moore, J. Dowding, H. Bratt, J. M. Gawron, Y. Gorfu, and A. Cheyer. 1997. Commandtalk: A spoken-language interface for battlefield simulations. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 1–7, Washington, DC. Association for Computational Linguistics.

S. J. Rosenkrantz and P. M. Lewis. 1970. Deterministic left corner parser. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, pages 139–152.