HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-98-02

Extended Context-Free Grammars and Normal Form Algorithms*

Jürgen Albert† Dora Giammarresi‡ Derick Wood§

May 27, 1998

Abstract

We investigate the complexity of a variety of normal-form transformations for extended context-free grammars, where by extended we mean that the set of right-hand sides for each nonterminal in such a grammar is a regular set. The study is motivated by the implementation project GraMa which will provide a C++ toolkit for the symbolic manipulation of context-free objects just as Grail does for regular objects.

Our results generalize known bounds for context-free grammars but do so in nontrivial ways. Specifically, we introduce a new representation scheme for extended context-free grammars (the symbol-threaded expression forest), a new normal form for these grammars (dot normal form) and new regular expression algorithms.

1 Introduction

in the 1960's, extended context-free grammars were introduced, using Backus–Naur form, as a useful abbreviatory notation that made context-free grammars easier to write. More recently, the Standardized General Markup Language (SGML) [13] used a similar abbreviatory notation to define extended context-free grammars for documents. Currently, XML [5], which is a simplified version of SGML, is being promoted as the markup language for the web, instead of HTML (a specific grammar or DTD specified using SGML). These developments led to the investigation of how notions applicable to context-free grammars could be carried over to extended context-free grammars. There does not appear to have been any consolidated effort to study extended context-free grammars in their own right. We begin

*This research was supported under a grant from the Research Grants Council of Hong Kong SAR. It was carried out while the first and second authors were visiting HKUST.

†Lerhstuhl für informatik II, Universität Würzburg, Am Hubland, D-97074 Würzburg, Germany.

‡Dipartimento di Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, via Torino, 155 - 30173 Venezia Mestre, Italy.

§Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR.

such an investigation with the most basic problems for extended context-free grammars: reduction and normal-form transformations. There has been some related work that is more directly motivated by SGML issues; see the proof of decidability of structural equivalence for extended context-free grammars [3] and the demonstration that SGML exceptions do not add expressive power to extended context-free grammars [14].

We are currently designing a manipulation system toolkit GraMa for extended context-free grammars, pushdown machines, and context-free expressions. It is an extension of Grail [17, 16], a similar toolkit for regular expressions and finite-state machines. As a result, we need to choose appropriate representations of grammars and machines that admit efficient transformation algorithms (as well as other algorithms of interest).

Earlier results on context-free grammars were obtained by Harrison and Yehudai [9, 10, 22] and by Hunt *et al.* [12] among others. Harrrison's chapter [9] on normal form transformations provides an excellent survey of early results. Cohen and Gotlieb [4] suggested a specific representation for context-free grammars and demonstrated how it aided the programming of various operations on them.

We first define extended context-free grammars using the notion of production schemas that are based on regular expressions, although we briefly discuss, in the final section of the paper, how the results carry over when we base the schemas on finite-state machines. As finite-state machines and regular expressions are both first-class objects in Grail, they can be used interchangeably and we expect that they will be in GraMa.

We then describe algorithms for the fundamental normal-form transformations in Section 3. Before doing so, we propose a representation for extended context-free grammars as regular expression forests with symbol threads. We then discuss some algorithmic problems for regular expressions before tackling the various normal forms. When we discuss unit-production removal we will modify this representation somewhat to ensure that we avoid blow-up in the size of the target grammar.

2 Notation and terminology

We treat extended context-free grammars as context-free grammars in which the right-hand sides of productions are regular expressions. Let V be an alphabet. Then, we define a regular expression over V and its language in the usual way [1, 21] with the Kleene plus as an additional operator. The symbol λ denotes the empty string.

An **extended context-free grammar** G is specified by a tuple (N, Σ, P, S), where N and Σ are disjoint finite alphabets of **nonterminal symbols** and **terminal symbols,** respectively, P is a finite set of **production schemas,** and the nonterminal S is the **sentence symbol.** Each production schema has the form $A \rightarrow E_A$, where A is a nonterminal and E_A is a regular expression over $V = N \cup \Sigma$. When $\beta = \beta_1 A \beta_2 \in V^*$, $A \rightarrow E_A \in P$, and $\alpha \in L(E)$, the string $\beta_1 \alpha \beta_2$ can be derived from the string β and we denote this fact by writing $\beta \Rightarrow \beta_1 \alpha \beta_2$. The **language L(G) of an extended context-free grammar G** is the set of terminal strings derivable from the sentence symbol of G. Formally, $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$, where $\Rightarrow^+$ denotes the transitive closure of the derivability relation.

Even though a production schema may correspond to an infinite number of ordinary context-free productions, it is known that extended and standard context-free grammars

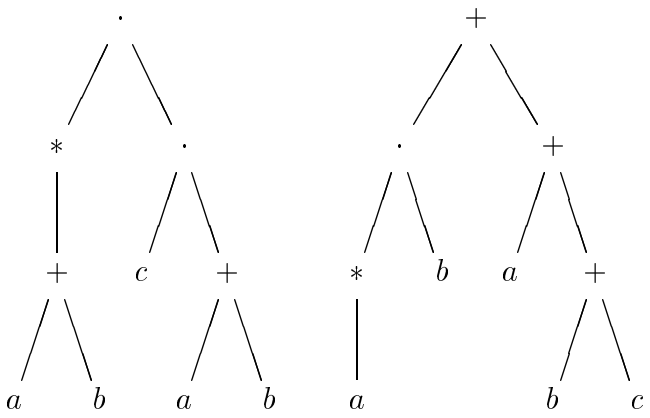

Figure 1: An expression forest for the regular expressions $(a+b)^*\cdot c\cdot(a+b)$ and $a^*b+(a+(b+c))$

describe exactly the same languages; for example, see the texts of Salomaa [20] and of Wood [21].

As is standard in the literature, we denote the size of a regular expression E by $|E|$ and define it as the number of symbols and metasymbols in E. We denote the size of a set A by $\#A$. To measure the complexity of any grammatical transformation we need to define the size of a grammar. There are two traditional measures of the size of a context-free grammar that we extend to extended context-free grammars as follows. Given an extended context-free grammar $G = (N, \Sigma, P, S)$, we define the **size of G** to be

$$\sum_{A \in N} (1 + |E_A|)$$

and we denote it by $|G|$. We define the **norm of G** to be

$$|G| \log \#(N \cup \Sigma)$$

and we denote it by $\| G \|$. Clearly, the norm is a more realistic measure of a grammar's size as it takes into account the size of the encoding of the symbols of the grammar but we use only the size measure here.

3 Normal-form transformations

We need the notion of an expression forest that corresponds to a set of regular expressions. In Fig. 3, we give an example of forest for two regular expressions.

As an extended context-free grammar has a number of production schemas that are regular expressions, we represent such grammars as an expression forest, where each tree in the forest corresponds to one production schema and each tree is named by its corresponding nonterminal. (The naming avoids the tree repetition problem.) We now add threads such that the thread for symbol X connects all appearances of the symbol X in the expression trees.

3.1 Reachability and usefulness

A symbol X is **reachable** if it appears in some string derived from the sentence symbol; that is, if there is a derivation $S \Rightarrow^+ \alpha X \beta$ where α and β are possibly empty strings over $\Sigma \cup N$.

As in standard context-free grammars, reachable symbols can be easily identified by means of a digraph traversal. More precisely, we construct a digraph whose vertices are symbols in $N \cup \Sigma$ and there is an edge from A to B if and only if B labels an external node of the expression tree named A. (We assume that the regular expressions do not contain the empty-set symbol.) Then, a depth-first traversal of this digraph starting from S gives all reachable symbols of the grammar. The time taken by the traversal and digraph construction is linear in the size of the grammar.

A nonterminal symbol A is **useful** if there is a derivation $A \Rightarrow^+ \alpha$, where α is a terminal string. The set of useful symbols can be computed recursively as follows. We first find all symbols B such that $L(E_B)$ contains a string of terminal symbols (possibly the empty string). All these B are useful symbols. Then, a symbol A is useful if $L(E_A)$ contains a string of terminals and the currently detected reachable symbols.

We can exploit the representation of the grammar by an expression forest to ensure that this algorithm runs in time linear in $|G|$. The idea is to make the test on $L(E_A)$ a pruning operation in the corresponding expression tree. More precisely, each time we test an $L(E_A)$, we delete part of its expression tree; once we "delete" the root, we will have discovered that A is useful. Since the algorithm actually deletes portions of the trees, we apply it to a copy of the expression forest. We first preprocess the forest to prune all subtrees that have "$*$" as their root label. More precisely, we perform a preorder traversal of each tree and whenever we find a $*$ we replace it by λ and delete its subtree. Then, we perform a second pruning in a bottom-up fashion starting from leaves that contain the empty string, terminal symbols or currently detected useful symbols. This operation is defined recursively as follows. Let x be a node of an expression tree and let $p(x)$ be its parent. If $p(x)$ is a "$+$", then delete the subtree rooted at $p(x)$ and recursively apply pruning at $p(p(x))$; if $p(x)$ is a "$\cdot$", then replace it with its other child (the sibling of x) and recursively apply pruning at this node. The overall time for pruning is linear in $|G|$ since we can use the threads to find all appearances of the symbol we want to prune in the expression forest. Moreover each node of the expression forest is traversed at most twice.

A grammar G is **reduced** if all its symbols are both reachable and useful. As in standard context-free grammars, to reduce a grammar first identify all useful symbols and then select (together with the corresponding schemas) those that are also reachable.

3.2 Null-free form

Given an extended context-free grammar $G = (N, \Sigma, P, S)$, we can determine the nullable nonterminals (the ones that derive the empty string) using a similar algorithm to the one we used for usefulness, see Section 3.1. This algorithm takes $O(|G|)$ time. Given this information we can make the grammar null free in two steps. First, we replace all appearances of each nullable symbol A with the regular expression $(A + \lambda)$. This step takes time proportional to

the total numbers of appearances of nullable symbols in G—we use the symbol threads for fast access to them.

In the second step we transform each production schema $A \to E_A$, where A is nullable, into a null-free production schema $A \to E'_A$, where $\lambda \notin L(E'_A)$. This step takes time at most $O(2^{|G|})$. The exponential behavior is attained in a pathological worst case when we have nested dotted subexpressions in which each operand of the dot can produce the empty string. We need to replace such a pair $E \cdot F$ with $E_- + F_- + (E_- \cdot F_-)$, where E_- means the transformed version of E that does not produce the empty string. Note that we double the length of the subexpression (at least). As the doubling can occur on the subexpressions of E and F and of their subexpressions, we obtain the exponential worst-case bound. (Note that this is the same case that occurs when the grammar is context-free and every symbol is nullable.)

Observe that if we represented the regular expressions with finite-state machines, we have at most quadratic blow up.

We can, however, apply a similar sleight-of-hand technique to the one of Yehudai [10, 22]. For standard context-free grammars, he reduces the lengths of the right-hand sides to at most two, so that the problem with catenation has only a constant blow up.

We can replace each dotted subtree of an expression tree, that has a dotted ancestor, with a new nonterminal and add a new production schema to the grammar. We can repeat this local modification until no dotted subtree has a dotted subtree within it. Thus, the new grammar has size that is of the same order as the original size. For example, given the production schema

$$A \to ((a + b) \cdot (b + \lambda) \cdot (c + d)) \cdot e;$$

we can replace it with new production schemas:

$$A \to (B \cdot (c + d)) \cdot e$$

and

$$B \to (a + b) \cdot (b + \lambda).$$

Repeating the transformation for A, we obtain

$$A \to C \cdot e$$

and

$$C \to B \cdot (c + d).$$

We say that the resulting grammar is in **dot normal form.** We can readily argue that the size of the resulting grammar is at most twice the size of the original size and it has increased the number of nonterminals by at most the size of the original grammar.

Basing the preceding null-removal construction on grammars in dot normal form, ensures that the algorithm runs in time $O(|G|)$ time and produces a new grammar that has size $O(|G|)$.

3.3 Unit-free form

A **unit production** is a production of the form $A \to B$. Transforming an extended context-free grammar into unit-free form requires three steps. First, we identify all unit-productions. Second, we transform the regular expressions (their representations as expression trees) such that their languages do not include unit nonterminal strings. Lastly, we modify the production schemas such that if $A \Rightarrow^+ B$ (B is unit-reachable from A), then $A \to E'_A + E_B$ where B is not in $L(E'_A)$.

Given an extended context-free grammar G, for each nonterminal symbol A, we can determine its unit productions in time $O(|E_A|)$ by traversing E_A in preorder. Whenever we meet a *- or a $+$-node we continue the traversal in the corresponding subtrees. If we meet a $\cdot$-node, then it depends on whether the languages corresponding to the two subtrees contain the empty string or not: if one of them does, then we continue the traversal in the other subtree (if both of them contain the empty string, then we continue the traversal in both subtrees). When eventually we reach a node labeled with a nonterminal B, then that occurrence of B corresponds to a unit production for A. We define a new thread, which we refer to as the "unit thread" that connects all occurrences of nonterminals that correspond to unit productions in the expression forest. The unit threads are used in the second step. Moreover, while discovering unit productions, we define, for each nonterminal A, the set of nonterminals that are unit-reachable from A. These sets are needed to modify the production schemas. The overall running time for this step is $O(|G|)$.

In the second step, we modify the expression trees at the nodes where we found unit productions. We traverse the expression trees from their frontiers to their roots and, in particular, we follow the paths that start from the nodes labeled with nonterminals that correspond to unit productions (we access them by following the unit threads). We go up the paths as long we meet only $+$-nodes.

Consider what happens when we reach a $\cdot$-node and its subtrees both yield the empty string. Let L and R be its subtrees and let $B = \{B_1, \ldots, B_h\}$ and $C = \{C_1, \ldots, C_k\}$ be the occurrences of nonterminals corresponding to unit productions that are in L and R, respectively, that we have not yet "eliminated". We replace the $\cdot$-subtree with a new tree that combines with $+$- and $\cdot$-nodes the following copies of the original subtrees: L, R, L without B, R without C, L without λ and R without λ. Using the notation, $[L], [R], [L - B], [R - C], [L - \lambda]$ and $[R - \lambda]$ to denote these subtrees, we construct the expression:

$$[L - B] + [R - C] + ([L] \cdot [R - \lambda]) + ([L - \lambda] \cdot [R]).$$

In this way we eliminate all unit nonterminal strings $B \cup C$ from the language of the regular expression. When only one of the subtrees yields the empty string, the transformation is simpler but similar.

When we meet a *-node, we replace its subtree in a similar way. Let T be a subtree with a *-root node and let $B = \{B_1, \ldots, B_h\}$ be the occurrences of nonterminals corresponding to unit productions that we have not yet eliminated. We replace that *-node with a subtree that combines with $+$- and Kleene $^+$-nodes the following subtrees: λ, T and T without B to give the regular expression

$$\lambda + [T - B] + [T]^{++}.$$

The time complexity of this second step is the same as that of the transformation of regular expressions into null-free expressions (see Section 3.2)

Finally, we need to modify the production schemas such that, for each nonterminal A, if $B_1, \ldots, B_h$ are all symbols unit reachable from A, then we define the new production schema as

$$A \rightarrow E_A + E_{B_1} + \ldots + E_{B_h}.$$

Rather than duplicating the corresponding trees in the expression forest (it would give a quadratic blow up in the size of the resulting grammar in terms of $|G|$) we define the new expression tree with +-nodes and pointers to the trees named A and $B_1, \ldots, B_h$, respectively. This takes time $O(\#N^2)$ and only $O(\#N^2)$ extra space is added to the size of the grammar G.

3.4 Greibach form

The original normal form result for context-free grammars was established by Sheila Greibach [7] in the 1960's; it was a key result in the use of the multiple-path syntactic analyzer developed at Harvard University at that time. An extended context-free grammar is in *Greibach normal form* if its productions have only the following form:

$$A \rightarrow a\alpha,$$

where a is a terminal symbol and α is a possibly empty string of nonterminal symbols. The transformation of an extended context-free grammar into Greibach normal form requires two giant steps: left-recursion removal and back left substitution. Recall that a grammar is left recursive if there is a nonterminal A such that $A \Rightarrow^+ A\alpha$ in the grammar, for some string α. We consider the second step first.

We assume that the given extended context-free grammar $G = (N, \Sigma, P, S)$ is reduced, factored, null free, and unit free. In addition, for the second step we also assume that the grammar is not left recursive. Since the grammar is not left recursive there is a partial order on the nonterminals, **left reachability,** that is defined by $A \leq B$ if there is a leftmost derivation $A \Rightarrow^* B\alpha$. As usual, we can consider the nonterminals to be enumerated as $A_1, \ldots, A_n$ such that whenever $A_i \leq A_j$, then $i \leq j$. Observe that A_n is already in Greibach normal form since it has only productions of the form $A_n \rightarrow a$, where $a \in \Sigma$. We now convert the nonterminals one at a time from A_{n-1} down to A_1. The conversion is conceptually simple, yet computational expensive. When converting A_i, we replace all nonterminals that can appear in the first positions in the strings in $L(E_{A_i})$ with their regular expressions. Thus, $A_i \rightarrow E'_A$ is now in Greibach normal form. To be able to carry out this substitution efficiently we first convert each regular expression E_i into **first normal form;** that is,

$$E'_i = A_{i+1} \cdot E_{i,i+1} + \cdots + A_n \cdot E_{i,n}$$

$$+\delta^i_{n+1} \cdot a_{n+1} + \cdots + \delta^i_{n+m} \cdot a_{n+m},$$

where $\Sigma = \{a_{n+1}, \ldots, a_{n+m}$, and $\delta^i_{n+k} = \{\lambda\}$, if $a_{n+k} \in L(E_i)$ and $\delta^i_{n+k} = \emptyset$, if $a_{n+k} \notin L(E_i)$. It is crucial that $L(E'_i) = L(E_i)$. Then, all we need do is replace the first A_k of each term with E_k.

If the grammar is left recursive, we first need to remove or change left recursion. We use a technique introduced by Greibach [8], investigated in detail by Hotz and his his coworkers [11, 18, 19] and rediscovered by others [6]. It involves producing, for each nonterminal, a distinct subgrammar of G that is left linear and, hence, can be converted into an equivalent right linear grammar. This conversion changes left recursion into right recursion and does not introduce any new left recursion. For more details, see Wood's text [21]. The important property of the left-linear subgrammars is that every sentential leftmost derivation sequence in G can be mimicked by a sequence of leftmost derivation sequences, each of which is a sentential leftmost derivation sequence in one of the left-linear subgrammars. Once we convert the left-linear grammars into right-linear grammars this property is weakened in that we mimic the original derivation sequence with a sequence of sentential rightmost derivation sequences in the right-linear grammars. The new grammar that is equivalent to G is the collection of the distinct right-linear grammars, one for each nonterminal in G.

As the modified grammar is no longer left recursive, we can now apply left back substitution to obtain a final grammar in Greibach normal form.

How well does this algorithm perform? The back left substitution causes, in the worst case, a blow up of $\#N|G|$ in the size of the Greibach normal form grammar. Left recursion removal causes a blow up of $\#N|G|$ at worst. Lastly, converting the production schemas into first normal form causes a blow up of $2\#N|G|$ (this bound is not obvious). As all three operations take time proportional to the sizes of their output grammars, essentially, Greibach normal form takes $O(\#N^5|G|^2)$ time, in the worst case. The reason for the $\#N^5$ term is that we first remove left recursion which not only increases the size of the grammar but also squares the number of nonterminals from $\#N$ to $\#N^2$! The number of nonterminals is crucial in the size bound for the grammar obtained by first normal form conversion and left back substitution.

We can however, reduce the worst-case time and space by using indirection as we did for unit-production removal. Rather than performing the back left substitution for a specific nonterminal, we use a reference to its schema. This technique gives a blowup of only $|G| + \#N^2$ at most and, thus, it reduces the complete conversion time to be $O(\#N^3|G|)$ in the worst case.

We can also apply the technique that Koch and Blum [15] suggested; namely, leave unit-production removal until after we have obtained Greibach-like normal form. Moreover, transforming an extended context-free grammar into dot normal form appears to be a very useful technique to avoid undesirable blow up in grammar size. We are currently investigating this and other approaches.

4 Final remarks

The results that we have presented are truly a generalization of similar results for context-free grammars. The time and space bounds are similar when relativized to the grammar sizes. The novelty of the algorithms is four-fold. First, we have introduced the regular expression forest with symbol threads as an efficient data representation for context-free grammars and extended context-free grammars. We believe that this representation is new. The only previously documented representations are those of Cohen and Gotlieb [4] and of Barnes [2]

and they are much more simplistic. Second, we have demonstrated how indirection using referencing can save time and space in the null-production removal and left back substitution algorithms. Although the use of the technique is novel in this context, it is well known technique in other applications. It is an application of lazy evaluation or evaluation on demand. Third, we have introduced dot normal form for extended context-free grammars that plays a role similar to Chomsky normal form for standard context-free grammars. Fourth, we have generalized the left-linear grammar to right-linear conversion to the extended case.

We are currently investigating whether we can get Greibach normal form more efficiently and whether we can improve the performance of unit-production removal.

Lastly, we would like to mention some other applications of the regular expression forest with symbol threads. First, we can reduce usefulness determination to nullability determination. Given an extended context-free grammar $G = (N, \Sigma, P, S)$, we can replace every appearance of every terminal symbol with the null string to give $G' = (N, \emptyset, P', S)$. Now, a nonterminal A in G is useful if and only if it is nullable in G'. Second, we can use the same algorithm to determine the length of the shortest terminal strings generated by each nonterminal symbol. The idea is that we replace each appearance of a terminal symbol with the integer 1 and each appearance of the null string with 0. We then repeatedly replace: each node labeled "+" that has two integer children with the minimum of the two integers; each node labeled "$\cdot$" that has two integer children with the sum of the two integers; and each node labeled "*" with 0. The root value is the required length.

We can use the same "generic" algorithm to compute the smallest terminal alphabet for the terminal strings derived from a nonterminal and so on.

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[2] K.R. Barnes. Exploratory steps towards a grammatical manipulation package (GRAMPA). Master's thesis, McMaster University, Hamilton, Ontario, Canada, 1972.

[3] H.A. Cameron and D. Wood. Structural equivalence of extended context-free and extended e0l grammars. In preparation, 1998.

[4] D.J. Cohen and C.C. Gotlieb. A list structure form of grammars for syntactic analysis. *Computing Surveys*, 2:65–82, 1970.

[5] D. Connolly. W3C web page on XML. http://www.w3.org/XML/, 1997.

[6] A. Ehrenfeucht and G. Rozenberg. An easy proof of Greibach normal form. *Information and Control*, 63:190–199, 1984.

[7] S.A. Greibach. A new normal form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12:42–52, 1965.

[8] S.A. Greibach. A simple proof of the standard-form theorem for context-free grammars. Technical report, Harvard University, Cambridge, MA, 1967.

[9] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.

[10] M.A. Harrison and A. Yehudai. Eliminating null rules in linear time. *Computer Journal*, 24:156–161, 1981.

[11] G. Hotz. Normal-form transformations of context-free grammars. *Acta Cybernetica*, 4:65–84, 1978.

[12] H.B. Hunt III, D.J. Rosenkrantz, and T.G. Szymanski. On the equivalence, containment and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences*, 12:222–268, 1976.

[13] ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.

[14] P. Kilpeläinen and D. Wood. SGML and exceptions. In C. Nicholas and D. Wood, editors, *Proceedings of the Third International Workshop on Principles of Document Processing (PODP 96)*, pages 39–49, Heidelberg, 1997. Springer-Verlag. Lecture Notes in Computer Science 1293.

[15] R. Koch and N. Blum. Greibach normal form transformation revisited. In Reischuk and Morvan, editors, *STACS'97 Proceedings*, pages 47–54, New York, NY, 1997. Springer-Verlag. Lecture Notes in Computer Science 1200.

[16] D. R. Raymond and D. Wood. Grail: Engineering automata in C++, version 2.5. Technical Report HKUST-CS96-24, Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong, 1996.

[17] D.R. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[18] R.J. Ross. *Grammar Transformations Based on Regular Decompositions of Context-Free Derivations.* PhD thesis, Department of Computer Science, Washington State University, Pullman, WA, USA, 1978.

[19] R.J. Ross, G. Hotz, and D.B. Benson. A general Greibach normal form transformation. Technical Report CS-78-048, Department of Computer Science, Washington State University, Pullman, WA, USA, 1978.

[20] A. Salomaa. *Formal Languages.* Academic Press, New York, NY, 1973.

[21] D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, second edition, 1998. In preparation.

[22] A. Yehudai. *On the Complexity of Grammar and Language Problems.* PhD thesis, University of California, Berkeley, CA, 1977.