TRANSITION DIAGRAM SYSTEMS AND NORMAL FORM ALGORITHMS

DORA GIAMMARRESI
Dipartimento di Matematica Applicata e Informatica,
Università Ca' Foscari di Venezia,
via Torino 155, 30173 Venezia Mestre, Italy
E-mail: dora@dsi.unive.it

DERICK WOOD
Department of Computer Science,
Hong Kong University of Science & Technology,
Clear Water Bay, Kowloon, Hong Kong SAR
E-mail: dwood@cs.ust.hk

We investigate the complexity of a variety of normal-form transformations for transition diagram systems, which are a parsing view of extended context-free grammars. A transition diagram system is a finite collection of finite-state machines each of which is labeled with a unique nonterminal symbol. 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. We provide both new algorithms and efficient implementations of them.

1 Introduction

In the 1960's, extended context-free grammars (as Backus–Naur form) were introduced as a useful abbreviatory notation that made context-free grammars easier to write. More recently, the Standardized General Markup Language (SGML) [18] used a similar abbreviatory notation to define extended context-free grammars for documents. Currently, XML [4], which is a simplified version of SGML, is being promoted as the markup language for the W3, instead of HTML (a specific grammar or DTD specified using SGML). The original development of extended context-free grammars was mirrored by the design of parsers based on them that used what Conway [5] calls transition diagram systems. These systems are simply finite collections of finite-state machines for the right-hand side languages of extended context-free grammars. The difference is, however, that they naturally suggest a parsing approach based on recursive finite-state machines. This idea was used by Glennie [11], by Conway [5], by Kaminger [19], as well as by Lomet [20,21]. More recently, the designers of the document editor Rita [6,8,7] used this model to more rapidly prototype the editor's syntax analyzer to specific document grammars. They also used it to direct the user's interaction with their structured documents by dynamically generating menus based on the current structure of a document and by generating possible insertion structures in the current context.

We are currently designing a manipulation toolkit GraMa for extended context-free grammars, pushdown machines, and context-free expressions. It is an extension of Grail [23,22], a similar toolkit for regular expressions and finite-state machines. As a result, we need to design representations of grammars and machines that admit efficient normal-form transformation algorithms (as well as other algorithms of interest). Specifically, as GraMa will use object-oriented inheritance, we have to provide methods for normal-form transformations that will be efficient whatever representation is used for the regular right-hand-side languages in extended context-free grammars. Understanding the complexity of these transformations for transition diagram systems will help us to design an efficient GraMa toolkit. Surprisingly, there does not appear to have been any consolidated effort to study transition diagram systems in their own right apart from the formalization work of Lomet [21]. In a companion paper we investigate the use of regular expressions for the right-hand-side languages [2].

Earlier results for normal-form transformation of context-free grammars were obtained by Harrison and Yehudai [14,15,27] and by Hunt *et al.* [17] among others. Harrison's discussion [14] of normal-form transformations provides an excellent survey of early results. Cohen and Gotlieb [3] suggested a specific representation for context-free grammars and demonstrated how it aided the programming of various operations on them. Their representation is, however, simplistic and cannot be easily (if at all) generalized for extended context-free grammars.

There has been, however, no previous attempt to provide efficient normal-form algorithms for *extended* context-free grammars. One reason is that work on extended context-free grammars has ignored how the regular right-hand sides are represented. Therefore, although the conceptual algorithms that we present are either standard or appear in the literature for context-free grammars, their efficient implementations for transition-diagram systems are not. The novelty of our approach is the use of the specific representation of grammars to obtain new, efficient algorithms. Moreover, the approach to deriving Greibach normal form, although not new, is certainly nonstandard—no texts apart from those of Floyd and Beigel [10], and Wood [26] come close to describing it.

We first define, in Section 2, transition-diagram systems using the notion of production schemas that are based on finite-state machines. We next informally describe algorithms for the fundamental normal-form transformations in Section 3.

2 Notation and terminology

We treat extended context-free grammars as context-free grammars in which the right-hand sides of productions are finite-state machines. Let V be an alphabet and let λ denote the null string. Then, we define a finite-state machine over V and its language in the usual way [1,26]. We consider machines that are in general nondeterministic and have null-input moves. Moreover, we assume that there is a unique start state and a unique final state and that those states are only-exiting and only-entering, respectively (without loss of generality since we can always add a new start and a new final state to the given machine). We specify a finite-state machine with a tuple $M = (Q, V, E, s, f)$, where Q is the set of states, $s, f \in Q$ are the start and the final states, respectively, and $E \subseteq Q \times V \cup \{\lambda\} \times Q$ is the transition relation. More precisely we will refer to the machine as the corresponding state-transition digraph, where Q is the set of vertices and E is the set of directed labeled edges. Moreover this digraph has a source and a sink vertex that corresponds to the start and final states, respectively. We will often need to determine whether, in a given machine, there is a path from a state p to a state q that corresponds to the string $w \in V^*$. We will denote such a path by $[p, w, q]$. Furthermore, we will often refer to an edge (p, X, q) as an X-edge or as an X-transition.

A **transition-diagram system** 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 $A \rightarrow M_A$, where A is a nonterminal and $M_A = (Q_A, V, E_A, s_A, f_A)$ is a finite-state machine over $V = N \cup \Sigma$, for some nonterminal A. We define the derivability relation $\Rightarrow$ in the usual way as follows: Given a string $\beta \in V^*$ such that $\beta = \beta_1 A \beta_2$, for some $A \in N$, we can derive $\beta_1 \alpha \beta_2$ from β if $\alpha \in L(M_A)$. As usual, we define the transitive closure of $\Rightarrow$ to give $\Rightarrow^+$ and the reflexive, transitive closure of $\Rightarrow$ to give $\Rightarrow^*$. The **language L(G) of a transition-diagram system 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 describe exactly the same languages; for example, see the text of Wood [26].

We denote the size of a set A by $\#A$. We define the size $|M|$ of a finite-state machine $M = (Q, V, E, s, f)$ by $|M| = \sum_{(p,X,q) \in E} |pXq|$, that is, as the size of the corresponding state digraph[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 transition-diagram systems as follows. Given a transition-diagram system $G = (N, \Sigma, P, S)$, we define the **size of G** to be

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

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

Because a transition-diagram system has a number of production schemas that are finite-state machines, we represent the system as a labeled digraph and also maintain the set of all start–final pairs (s_A, f_A) of the machines. Initially, a transition-diagram system is a disconnected collection of digraphs (or finite-state machines); however, some of the transformations we discuss introduce connectivity.

Given a nonterminal A and its start–final pair (s_A, f_A), the subgraph that includes all paths from s_A to f_A gives the production schema for A. We have chosen the representation to avoid subgraph repetition as we shall demonstrate later.

We describe each of the normal-form transformations in a similar way. First, we define the transformation; second, we describe a conceptual algorithm; and, third, we present somewhat informally an efficient implementation. The last step involves processing and modifying the finite-state machines of the system. For each algorithm that we present, we assume that, for each nonterminal A in a given transition-diagram system $G = (N, \Sigma, P, S)$, its machine M_A has only reachable and useful states.

3.1 Reachability, usefulness and reduction

We provide three efficient algorithms for transition-diagram systems: to detect reachable symbols, to detect useful symbols and to reduce a system. All the

[a] More precisely, $|M| = 3\#E$.

algorithms run in time linear in the size of G in the worst case.

We begin with the definitions. A symbol X is **reachable** in a transition-diagram system $G = (N, \Sigma, P, S)$ if it appears in some string derived from the sentence symbol S; that is, if there is a derivation $S \Rightarrow^{+} \alpha X \beta$ where α and β are possibly null strings over $\Sigma \cup N$. A nonterminal symbol A is **useful** if there is a derivation $A \Rightarrow^{+} \alpha$, where α is a terminal string. A system G is **reduced** if all its symbols are both reachable and useful.

As in standard context-free grammars, reachable symbols can be easily identified by means of a digraph traversal. More precisely, we conceptually construct a digraph whose vertices are symbols in $N \cup \Sigma$ and such that there is an edge from A to B if and only if B labels an edge in the machine M_A. Then, a depth-first traversal of this digraph starting from S gives all reachable symbols of the system. The time taken by the traversal and digraph construction is linear in the size of the system.

The set of useful symbols can be computed iteratively as follows. First, all terminal symbols are useful. Second, in one pass through the system, we can find all nonterminals B such that in M_B there is some path $[s_B, w, f_B]$, where w is a string (possibly null) of terminal symbols. All such nonterminals B are classified as useful symbols. Next, an unclassified nonterminal A is useful if, in M_A, there is a path $[s_A, w, f_A]$, where w is a string of currently useful symbols. We now carry out another pass through the system until no unclassified nonterminals are demonstrated to be useful. The algorithm, as described, can be implemented to run in $O(\#N|G|)$ time.

We now explain how we can speed up the preceding algorithm such that it runs in time linear in $|G|$. In the first pass, we use what we call a **blocking traversal** of each finite-state machine M_A from its start state s_A. A blocking traversal is a standard breadth-first traversal except that whenever it reaches edges that are labeled with unclassified nonterminals, it does not follow them; it treats them as if they are not in the digraph. If (p, B, q) is such a blocking edge, then the traversal also adds q to the continuing set $\mathrm{cont}(B)$ for B. (Before beginning the traversals, we initialize all continuing sets to be empty.) As a result, the only edges that the initial blocking traversal explores are ones that are labeled with either terminals or the null string. Therefore, if the traversal ever visits the final state f_A, then we have demonstrated that A is useful.

After we finish the initialization pass, some nonterminals are classified as useful. (If none are, then $L(G)$ is empty.) If there are only empty continuing sets, then we have found all useful nonterminals and the algorithm terminates. Otherwise, we carry out a second pass. For all useful nonterminals B, we continue the blocking traversals, in all the machines, from the states in the continuing set of B since the edges that were previously blocked can now be followed. We also remove these states from their continuing sets. As before, whenever a blocking traversal reaches edges that are labeled with unclassified nonterminals, it does not follow them; it treats them as if they are not in the digraph. If (p, C, q) is such a blocking edge, then the traversal also adds q to the continuing set $\mathrm{cont}(C)$. As a result, the only edges that the blocking traversal explores are ones that are labeled with either useful symbols or the null string. We continue the blocking traversals until none of them can be continued. When one of the blocking traversals meets a final state, it demonstrates that the corresponding nonterminal is useful.

We carry of further passes, in a similar manner, until no more unclassified nonterminals are demonstrated to be useful. As we maintain the partial traversals at all times, we can ensure that we never explore an edge more than once. Thus, the algorithm runs in $O(|G|)$ time.

As for standard context-free grammars, to reduce a system we first identify all useful symbols and then select (together with the corresponding schemas) those symbols (and their schemas) that are also reachable.

To remove useless nonterminals and their associated appearances, we remove their machines, remove the useless symbols from N, remove all appearances of useless symbols in the remaining machines and, lastly, reduce all remaining machines. Since all machines are now reduced, we can remove unreachable symbols as follows. First, remove all unreachable symbols from N and Σ and then remove all machines labeled with unreachable nonterminals. Note that unreachable symbols do not appear in any of the remaining machines as they have already been reduced. The two steps take $O(|G|)$ time in the worst case.

3.2 Null-free form

Given a transition-diagram system $G = (N, \Sigma, P, S)$, we can make it null free in three steps. First, we determine the nullable nonterminals (the ones that derive the null string) using a similar algorithm to the one we used for usefulness detection, see Section 3.1. This algorithm takes $O(|G|)$ time. Second, for each nullable symbol A and for all edges (p, A, q) (in all the machines), we add new edges (p, λ, q). Therefore, the second step takes time proportional to the total numbers of appearances of nullable symbols in G and at most doubles the size of the system; that is, it takes $O(|G|)$ time in the worst case. Third, we transform each production schema $A \rightarrow M_A$, where A is nullable, into a null-free production schema $A \rightarrow M'_A$, where $\lambda \notin L(M'_A)$. This step takes time $O(|G|)$, in the worst case, and it produces a new system that has size $O(|G|)$.

We now describe the third step in more detail. Let $M = (Q, V, E, s, f)$

be a finite-state machine such that the null string is in $L(M)$. We wish to transform M such that its language does not contain the null string. The idea is to guarantee that each path from s to f contains at least one nonnull symbol. One way is to ensure that all paths from s to f start with a nonnull labeled edge. Notice that, since $s \neq f$, the null string is in $L(M)$ if and only if there is at least one null path from s to f. We consider the subgraph of M that consists of all null paths from s and we refer to its set of vertices as S^λ. Then, we ensure that all the paths in M start with a nonnull labeled edge. More formally, we define the following transformation:

Null-removal from the start: *Let S^λ be the set of states reachable from s by null paths. Delete all edges (s, λ, p) and, for each edge $(p, B, q) \in E$ with $B \neq \lambda$, $p \in S^\lambda$ and $p \neq s$, add a new edge (s, B, q) to E.*

After applying this transformation, we say the machine is in **start-null-free form.**

Alternatively, we can achieve the same result by ensuring that all paths from s to f end with a nonnull labeled edge. We define this complementary transformation as follows.

Null removal to the final: *Let F^λ be the set of states from which f is reachable by null paths. Delete all edges (q, λ, f) and, for each edge $(p, B, q) \in E$ with $B \neq \lambda$, $q \in F^\lambda$ and $q \neq f$, add a new edge (p, B, f) to E.*

After applying this transformation, we say the machine is in **final-null-free form.** We make use of these two transformations in Section 3.3.

It is straightforward to verify that applying one of these two transformations to a given machine M whose language contains the null string, will remove the null string from the language. Moreover the transformation takes time $O(|M|)$, in the worst case, and doubles, at most, the size of M.

Thus, performing the third step of null-production removal takes a total time of $O(|G|)$ with only a linear increase in the size of G, achieving linear time for all three steps with a linear increase in the size of G.

3.3 Unit-free form

A **unit production** is a production of the form $A \rightarrow B$ where A and B are nonterminals. We assume that the given transition-diagram system is null-free, in which case, to transform it into unit-free form requires three steps. First, we detect all unit productions. Second, we transform each finite-state machine M_A into a machine $\bar{M}_A$ such that $\bar{M}_A$'s language does not include any unit-nonterminal strings but does include all other strings in $L(M_A)$. Third and lastly, we modify the production schemas such that whenever $A \Rightarrow^+ B_i$, $1 \leq i \leq h$, in the original system, for some $h \geq 0$ (B_i is unit reachable from A), we replace $A \rightarrow M_A$ in the original system with $A \rightarrow M'_A$, where $L(M'_A) = L(\bar{M}_A) \cup \bigcup_{i=1}^{h} L(\bar{M}_{B_i})$.

Given a transition-diagram system G, for each nonterminal symbol A, A's unit productions correspond to nonterminal strings of length one accepted by the corresponding machine M_A. In the first step, we transform each machine $M = (Q, V, E, s, f)$ into both *start-null-free form* and *final-null-free form.* This takes time $O(|M|)$, in the worst case. It simplifies both the detection and the removal of unit-nonterminal strings accepted by M since such strings correspond to edges (s, B, f), where B is a nonterminal symbol. We refer to (s, B, f) as a unit edge. Next, for all nonterminals A, we compute the set U_A of all nonterminals that are unit reachable from A. These sets are needed to modify the finite-state machines. The running time for this step is $O(|G|)$.

Now, in the second step, we remove all edges of the form (s_A, B, f_A) from machine M_A, where B is a nonterminal symbol, to give a unit-free machine $\bar{M}_A$. Since s_A is only exiting and f_A is only entering, this removal only removes B from $L(M_A)$; it does not remove any other strings from $L(M_A)$. Thus, $L(\bar{M}_A) = L(M_A) - N$, where N is the nonterminal alphabet. Now, let $\bar{M}_A = (Q_A, V, \bar{E}_A, s_A, f_A)$ be the unit-free machines of the original system G. The time to perform the second step is $O(|G|)$.

In the third and final step, we modify the finite-state machines such that, for each nonterminal A, if $B_1, \ldots, B_h$ are all symbols unit reachable from A, then the new production schema is $A \rightarrow M'_A$, where

$$L(M'_A) = L(\bar{M}_A) \cup L(\bar{M}_{B_1}) \cup \cdots \cup L(\bar{M}_{B_h}).$$

To avoid duplicating the state digraphs which would give a quadratic blow up in the size of the resulting system in terms of $|G|$, we define the new machines $M'_A = (Q'_A, V, E'_A, s'_A, f'_A)$ as follows:

$$\begin{aligned}
Q'_A &= Q_A \cup Q_{B_1} \cup \cdots \cup Q_{B_h} \cup \{s'_A, f'_A\}, \\
E'_A &= \bar{E}_A \cup \bar{E}_{B_1} \cup \ldots \cup \bar{E}_{B_h} \\
&\quad \cup \{(s'_A, \lambda, s_{B_i}) : 1 \leq i \leq h\} \\
&\quad \cup \{(f_{B_i}, \lambda, f'_A) : 1 \leq i \leq h\} \\
&\quad \cup \{(s'_A, \lambda, s_A), (f_A, \lambda, f'_A)\}.
\end{aligned}$$

We have added a new start state and a new final state to $\bar{M}_A$ and have avoided making duplicate copies of the machines $\bar{M}_{B_i}$ by adding the null edges from the new start state to the start states of all original machines that are involved.

This modification ensures that we can enter any of these machines when determining a specific production for A. More important, by adding null edges from the final states of all involved original machines to the new final state, we guarantee that only strings that determine a path from the new start state to the new final state yield a production for A. There are at most $2\#N$ new null edges for each machine M'_A; therefore, this step takes time $O(\#N^2)$ with an increase of $O(\#N^2)$ in the size of the system G.

In total, this unit-removal algorithm takes $O(|G|)$ time and space, which improves the usual algorithm for context-free grammars since it does explicit substitutions for every unit production.

3.4 Greibach form

This normal form result for context-free grammars was established in the 1960's by Sheila Greibach [12]. It was a key result in the use of the multiple-path syntactic analyzer developed at Harvard University at that time. A transition-diagram system is in *Greibach normal form* if its productions have only the following form:

$$A \to a\alpha,$$

where a is a terminal symbol and α is a possibly empty string of nonterminal symbols. This definition is analogous to that for standard context-free grammars. The transformation of a transition-diagram system into Greibach normal form uses two steps: left-recursion removal and back left substitution. Recall that a system is **left recursive** if there is a nonterminal A such that $A \Rightarrow^+ A\alpha$ in the system, for some string α. Many texts use a recursive transformation that gives a system in Greibach normal form of size exponential in the size of the original system; see Aho and Ullman's text [1] for example. We use a different approach that is not well known yet has the advantage that is has at most quadratic blow-up in the size of the system. We consider the second step first.

We assume that the given transition-diagram system $G = (N, \Sigma, P, S)$ is reduced, factored, null free, and unit free. In addition, for the second step, we also assume that the system is not left recursive. Since the system 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 $A_1, \dots, A_n$ 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 $A_n \to a$, 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 it could be computational expensive. When converting A_i, we replace all nonterminals that can appear in the first positions in the strings in $L(M_{A_i})$ with a copy of their machines. (If we do this replacement explicitly, we get a quadratic blow-up in the size of G.) Thus, $A_i \rightarrow M'_A$ is now in Greibach normal form.

To carry out the replacement efficiently we first convert each finite machine M_i into start-null-free form, which takes linear time and space with respect to $|G|$ (see Section 3.2). Then, to avoid quadratic blow up we can do an implicit substitution of the nonterminal symbol by its machine as follows. For each machine $M = (Q, V, E, s, f)$, we clone both its start and the final states. More precisely, we rename s and f as $\bar{s}$ and $\bar{f}$, respectively, and refer to them as dummy start and dummy final states, respectively. Then, we connect the dummy states to their corresponding real states by defining new null edges $(s, \lambda, \bar{s})$ and $(\bar{f}, \lambda, f)$. Now to substitute (implicitly) an edge $(\bar{s_A}, B, q)$ with the corresponding machine M_B we add only edges $(\bar{s_A}, \lambda, \bar{s_B})$ and $(\bar{f_B}, \lambda, q)$, and remove the edge $(\bar{s_A}, B, q)$. It gives only a linear increase in the size of the system.

If the system is left recursive, we first need to remove left recursion. We use a technique introduced by Greibach [13], investigated in detail by Hotz and his his co-workers [16,24,25] and rediscovered by others [9]. It involves producing, for each nonterminal, a distinct subsystem of G that is left linear and, hence, can be converted into an equivalent right linear system. (Although this specific approach was not used by Greibach, she used a similar approach.) This conversion changes left recursion into right recursion and does not introduce any new left recursion. For more details, see Wood's text [26]. The important property of the left-linear subsystem 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 subsystems. Once we convert the left-linear systems into right-linear systems this property is weakened in that we mimic the original derivation sequence with a sequence of sentential rightmost derivation sequences in the right-linear systems. The new system that is equivalent to G is the collection of the distinct right-linear systems, one for each nonterminal in G.

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

We now analyze the complexity of the described algorithm. Left-recursion removal causes a blow up of $\#N|G|$ at worst in the size of the system and squares the number of nonterminals from $\#N$ to $\#N^2$. Then, we avoid a blow up of $\#N|G|$ in the size of the system by doing back left substitution explicitly, we use the preceding implicit technique. In this case, the only expensive (that is, nonlinear) step is the left-recursion removal; thus, the complete conversion time and space is reduced to $O(\#N|G|)$.

Acknowledgments

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

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. J. Albert, D. Giammarresi, and D. Wood. Extended context-free grammars and normal form transformations. In *Automata Implementation: Third International Workshop on Implementing Automata, WIA '98*, New York, NY, 1998. Springer-Verlag.
3. D.J. Cohen and C.C. Gotlieb. A list structure form of grammars for syntactic analysis. *Computing Surveys*, 2:65–82, 1970.
4. D. Connolly. W3C web page on XML. http://www.w3.org/XML/, 1997.
5. M.E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6:296–408, 1963.
6. D.D. Cowan and G. de V. Smit. Document editing with batch document formatting. In J.C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 140–153, Cambridge, 1986. Cambridge University Press.
7. D.D. Cowan, E.W. Mackie, G.M. Pianosi, and G. de V. Smit. Rita—An editor and user interface for manipulating structured documents. *Electronic Publishing, Origination, Dissemination, and Design*, 4(3):125–150, September 1991.
8. G. de V. Smit. *A Formatter-Independent Structured Document Preparation System.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1987.
9. A. Ehrenfeucht and G. Rozenberg. An easy proof of Greibach normal form. *Information and Control*, 63:190–199, 1984.
10. R.W. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Language Theory.* Computer Science Press, San Francisco, CA, 1994.
11. A.E. Glennie. On the syntax machine and the construction of a universal compiler. Technical Report 2, Computation Center, Carnegie Institute

of Technology, Schenley Park, Pittsburgh, PA, 1960.
12. S.A. Greibach. A new normal form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12:42–52, 1965.
13. S.A. Greibach. A simple proof of the standard-form theorem for context-free grammars. Technical report, Harvard University, Cambridge, MA, 1967.
14. M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.
15. M.A. Harrison and A. Yehudai. Eliminating null rules in linear time. *Computer Journal*, 24:156–161, 1981.
16. G. Hotz. Normal-form transformations of context-free grammars. *Acta Cybernetica*, 4:65–84, 1978.
17. 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.
18. ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.
19. P. Kaminger. Syntax definition by means of recursive graphs. Technical Report TR 25.078, IBM Laboratory Vienna, Austria, February 1968.
20. D.B. Lomet. The construction of efficient deterministic language processors. Technical Report RC 2768, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, January 1970.
21. D.B. Lomet. A formalization of transition diagram systems. *Journal of the ACM*, 20:235–257, 1973.
22. D. R. Raymond and D. Wood. Grail: Engineering automata in C++, version 2.5. Technical Report HKUST-CS96-24, Department of Computer Science, HKUST, Kowloon, Hong Kong, 1996.
23. D.R. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.
24. 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.
25. 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.
26. D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, second edition, 1998. In preparation.
27. A. Yehudai. *On the Complexity of Grammar and Language Problems.* PhD thesis, University of California, Berkeley, CA, 1977.