Indiana University

COMPUTER SCIENCE Department

TECHNICAL REPORT NO. 2

THE SIZE OF LR(0) MACHINES

PAUL WALTON PURDOM JR.

TECHNICAL REPORT NO. 2
THE SIZE OF LR(0) MACHINES

PAUL WALTON PURDOM JR.

THE SIZE OF LR(0) MACHINES

ABSTRACT: Simple formulas are given for estimating the size (number of states and transitions) of an LR(0) parser. The results for number of states also apply to the SLR(1) and LALR(1) parsers of DeRemer. The formulas, which result from studying the LR(0) machines for 134 published grammars, have an accuracy of about 10 percent. The results indicate that for normal applications the number of states increases linearly with the size of the grammar. For example $D_S/S = 0.620 - 0.0001S \pm 0.072$ where D_S is the number of states and S is the size of the grammar (the total length of the right sides plus the number of productions). Since there exist grammars for which the number of states of an LR(0) parser increases exponentially with the size of the grammar, the question of why the parser for most practical grammars are so small is also considered. The results are used to compare the space for LR(0) parsers and weak precedence parsers.

INTRODUCTION

DeRemer(1) has developed methods for building fast LR(k) type parsers which usually have a much smaller number of states than those built by Knuth's(2) original algorithms. When comparing a parser produced by DeRemer's method with one produced by another method, one important consideration is the amount of space required to store each parser.

Consider the following set of grammars (similar to one found by Reynolds(3)):

$$R \rightarrow A_i u_i \qquad (1 \leq i \leq n)$$

$$A_i \rightarrow c_j A_i v_j \qquad (1 \leq i \neq j \leq n)$$

$$A_i \rightarrow c_i B_i w_i \mid d_i x_i \qquad (1 \leq i \leq n)$$

$$B_i \rightarrow c_j B_i y_{ij} \mid d_i z_i \qquad (1 \leq i, j \leq n),$$

where R is the starting symbol, $\{ A_i, B_i \}$ are nonterminal symbols, and $\{ c_i, d_i, u_i, v_i, w_i, x_i, y_{ij}, z_i \}$ are terminal symbols. These grammars can be parsed by the simple precedence method of Weber and Wirth(4). Simple precedence method requires an amount of storage equal to that required to store the grammar plus about N^2 bits where N is the number of symbols in the grammar. Thus the storage required to parse the n^{th} grammar is $O(n^4)$ bits if a simple precedence parser is used. There is a trivial fast parsing method for this set of grammars which can be stored in $O(n^2 \log n)$ bits (the space for storing a copy of the grammar). On the other hand an LR(0) parser for this grammar has $O(n2^n)$ states and $O(n^2 2^n)$ transitions. Therefore, the amount of storage required is $O(n^2 2^n \log n)$ bits if the parser is stored in the way indicated by DeRemer(1).

Although the above example shows that LR(k) parsers can be very large, empirical studies by DeRemer(1), by Horning and LaFonde (5), and by Anderson, Eve, and Horning (6) show that SLR(1) parsers are usually of reasonable size. Furthermore Kemp(7) recently found a formula for the exact size of an LR(0) parser. Although his formula is extremely complex for the general case, it shows that certain grammars have an LR(0) parser with no more states than 2 plus the length of the right sides of the productions.

The algorithms of DeRemer build the finite control for a push-down automaton from any grammar. If the grammar is in the appropriate class for the algorithm then the control is deterministic. In this paper the resulting machine will often be called a parser even if it is not deterministic. Since the main potential application of LR(k) methods is for parsing programming languages, the main interest is in the SLR(1) and LALR(1) methods given by DeRemer (1). Since the control built by the LR(0) algorithm has nearly the same number of states as that built by the SLR(1) and the LALR(1) algorithms, the formula for the size of LR(0) parsers can be used to estimate the size of SLR(1) and LALR(1) parsers.

The results of this paper were obtained by first building parsers to note the general trend in their size as a function of various characteristics of their grammar, then by doing a theoretical examination of the algorithms to identify which simple features of the grammar could best be used to predict the size of the parser, and finally by doing a statistical fit to the data obtained by building parsers. This resulted in very simple formulas for predicting the number of states with an accuracy of about 10 percent. This can be

compared with Kemp's (7) result, which is exact, but which in general takes a considerable effort to compute.

The statistical results were obtained by considering 134 published grammars, including 84 LALR(1) grammars. Additional details about these grammars are given in the appendix.

It is assumed that the grammars are reduced and that the starting symbol is not used on the right side of any production.

Unlike Kemp(7) the formulas of this paper do not include the final state. Thus he always obtains one more state.

The statistical results are summarized in table 1 where various quantities y have been fit to quantitities x using a least squares fit to $y_i = a + bx_i + e_i$ where e_i is the i^{th} error. The standard error is $(\sum_i e_i^2 / (n-2))^{1/2}$ where n is the number of data points. The meaning of the various symbols in the x and y columns are given in the following sections.

2. THE GRAMMAR FOREST

To study the size of LR(0) parsers it is useful to represent the grammar as a forest where there is a tree for each nonterminal. In the tree for any nonterminal A, each node on the k^{th} level can be reached by a unique string of symbols of length k ($k \geq 0$) which is the first k symbols of the right side of one or more productions that has A as left side symbol. Each node on level k of the tree for A has a transition under each symbol x such that the string used to reach the node followed by x is the same as the first k+1 symbols of some production that has A as the left symbol. The number of arcs in the tree for A can easily be obtained by counting the length in symbols of each production that has A on the left side, but omitting those symbols that make up an initial string that is the same as an initial string of some production of A that has already been counted.

The augmented grammar is formed by adding symbol $\#_i$ (which is not used elsewhere in the grammar) to the right end of the i^{th} production for each production in the grammar. This grammar is used in the construction of the narser.

3. NONDETERMINISTIC MACHINE

As indicated by DeRemer(1) the nondeterministic machine can be built by building the forest for the augmented grammar, combining all the leaves into a single node, and for each node that has a transition under a nonterminal symbol add a transition under the empty symbol to the root of the tree for that nonterminal (omitting arcs connecting a node to itself). The root of the tree for the starting symbol becomes the initial state and the node formed from the leaves becomes the final state.

The number of states, N_S, and transitions, N_T, are given by

$$N_S = N_N + T$$
$$N_T = P + T + T_N - R_I$$

where N_N is the number of nonterminals, T is the number of arcs in the grammar tree, P is the number of productions, T_N is the number of nonterminals arcs in the tree, and R_I is the number of immediately left recursive nonterminals.

Since many properties of the parser can be estimated from the grammar size, S, which is the number of productions plus the sum of the length of right sides of all productions, table 1 also contains fits to N_S and N_T in terms of S.

For what follows it is useful to divide the status of the nondeterminisitc machine into two classes.

Definition: A state that arises from the root of a tree (except the tree for the start symbol) is called a nonvital state. Any other state is called a vital state.

The nonvital states are reached only by ε -transitions. The vital states are never reached by an ε-transition.

4. DETERMINISTIC MACHINE

The deterministic machine, which recognizes characteristic strings, can be built by a standard algorithm which associates each set of states in the nondeterministic that can reach from the initial state with a state of the deterministic machine. Following DeRemer(1), however, the state corresponding to the empty set and all transitions to it are omitted. When referring to both the nondeterministic machine and to the deterministic machine, the states of the deterministic machine will often be called state sets to emphasize their relation to sets of states in the nondeterministic machine.

Theorem: The machine producted by the above algorithm operating on a nondeterministic machine produced by the preceding algoritms is a reduced machine.

Proof: The vital states in a state set determine which nonvital states are in the state set. Also there is only one final state because of the unique symbols, $\#_1$, which cause transitions to the final state. Consider two distinct state sets of the deterministic machine neither of which is the final state. Then there is a vital state in one set that is not in the other. Consider the sequence that will cause the nondeterministic machine to go from that state to the final state without following any ϵ-transitions. The same sequence will cause the deterministic machine to go from the state set containing that state to the final state set. No other state of the nondeterministic machine will go to the final state under the sequence being considered even if it follows some ϵ-transitions. This follows because 1) the machine has a tree structure except for the ϵ-transitions and the final state, 2) the last symbol of the sequence (which is of the form $\#_1$)

is used on only one transition in the machine, 3) the ε -transitions to to the root of trees, and 4) a vital state can not be reached from any other state by an ε -transition. Point 2) implies a unique next to last state on the path. Points 1) through 3) imply that any path under the sequence must go through the vital state under discussion and must get there without using any symbols of the sequence. Point 4) implies that at least one symbol must be read to get to that vital state from any other state.

Since only one of the two state sets under discussion contains the state from the nondeterministic machine which can reach the final state with the selected sequence, the other state set can not reach the final state set under this sequence. Since all pairs of states can be distinguished the machine is reduced.

As indicated by DeRemer(1) the machine built by this section needs some modification to produce a practical LR(0), SLR(1), or LALR(1) parser. We will not consider these algorithms except to remark that for those cases where the statistical formulas in table 1 give accurate answers using DeRemer's method will increase the number of states by about N_N (the number of nonterminal symbols) and the number of transitions by the amount required for lookahead. If the methods of Aho and Ullman(8) are used, only the transitions are increased (by the amount required for lookahead). The results are directly comparable to those of Kemp(7) except that he counts one more state.

5. FACTORS AFFECTING THE SIZE OF THE PARSER

The experimental data indicated that the number of state sets is about equal to the number of vital states. Considering what can happen with two vital states i and j, since i and j must both be in the final machine, one can have the following types of state sets: 1) {i,j} , 2) {i}{j} , 3) {i}{i,j} , 4) {j}{i,j} and 5) {i}{j}{i,j} , where all states except i and j have been omitted from the sets. The first results in one state set from i and j. The last results in three. The remaining cases result in two.

Generalizing these ideas to more than 2 states, the following definitions for types of state sets are useful:

simple: contains only one vital state.

replacement: contains two or more vital states, including at least one that does not appear in any other state set.

simple replacement: a replacement set that contains no more than one vital state which appears in another state set.

other replacement: a replacement set that is not a simple replacement set.

chain: a nonreplacement state set that contains at least one vital state that appears in no state set except those containing all the states of the chain state set.

complex: any other state set.

If the deterministic machine has only simple states then it contains T + 1 states (the number of vital states). If it contains n complex states, then it contains no more than n + T + 1 states. Table 2 shows that most of the states in a machine are usually simple states and that simple replacement states make up a large portion of the remainder. The machines have very few complex states.

Define L_A to be the set of all symbols which are the first symbol (in the augmented grammar) of the right side of some production with

A as the left side. When A is a terminal symbol L_A is empty. Then L_A^+ is the set of all symbols that are the first symbol of a string derivable from A in one or more steps and L_A^* is $L^+ \cup A$. A symbol A is left recursive if $A \in L_A^+$.

The state set for the initial state of the deterministic machine is simple. The transition from one state set under some symbol A will go to a simple state set if and only if the original state set contains exactly one state with a transition under the symbol A. A transition under some symbol A will go to a complex state set if and only if the original state set contains two states with transitions under A. there exists a state set (corresponding to a state in the reduced deterministic machine) with the first of these two states but not the second and there exists also a state set with the second state but not the first.

To understand the relation between properties of the grammar and the types of state sets, it is useful to rephrase these results in a way that draws particular attention to the vital states in a state set and where the sets of the type L_A^+ are used to determine the transition from the nonvital states in the state set. This approach permits one to understand why a machine has mostly simple states and very few complex states.

The initial state set of the machine is simple. Also the state sets that are reached by transitions obeying the conditions in the rest of this paragraph are simple. For any symbol A, if A causes a transition from one of the vital states in the state set, if A does not cause a transition from any other vital states in the state set, and A is not an element of L_B^+ for any symbol B (perhaps equal to A)

which causes a transition from a vital state in the state set, then the state set reached by a transition under A is simple. If A is a symbol that does not cause a transition for any vital state of the state set, but $A \varepsilon L_B^+$ for some symbol B that does, then A causes a transition to a simple state set unless there exists C, D, and E where C causes a transition for some vital state of the state set, $D \neq E$, $D \varepsilon L_B^+$, $E \varepsilon L_C^+$, $A \varepsilon L_D$, and $A \varepsilon L_E$. This includes all the ways a simple state can be reached. It indicates why one might expect a lot of simple states but also indicates that most machines will have some nonsimple states.

The following are the conditions that will cause a transition under a symbol A to go to a complex state. If A causes a transition both for some vital state i and for some other vital state j of a state set, and the machine also has a state set with i but not j, and one with j but not i, then the state set reached by a transition under A is complex. If A causes a transition for some vital state i of a state set, $A \varepsilon L_B^+$ with $B \neq A$ where B causes a transtion for some other vital state j of the state set, the machine also has a state set with i but not j, and the machine has some other state with a transition under B, then the state set reached is complex. If A does not cause a transtion for any vital state, but $A \varepsilon L_B^+$ where B does, then if C causes a transition for some vital state and there exists D and E with $A \varepsilon L_D$, $A \varepsilon L_E$, $D \varepsilon L_B^+$, and $E \varepsilon L_C^+$ then the state set reached by A is complex providing there exists a state set with a transition under D but not E and one with a transition under E but not D. These are all the conditions that cause transitions to complex state sets. Only the last

condition can cause a transition from a simple state to a complex state. This is the condition that Reynolds(3) made extensive use of in his grammar set. As indicated by the above conditions, it takes real perversity to get many complex states into a grammar.

Transitions from a simple state set under a left recursive symbol always leads to a replacement state set. If the symbol is immediately left recursive, the state set will be simple replacement. These are the most important, but not the only ways that a replacement state set can be reached.

For each recursive symbol there is at least one vital state (exactly one for immediately recursive symbols) that never occurs except with other vital states, and those other vital states never occur without the state(s) associated with the recursive symbol. Thus for machines with no complex state sets, the number of states, D_S, obeys the relation $D_S \leq T + 1 - R$ where R is the number of recursive symbols.

One may completely analyze a machine with the above methods by assuming the deterministic machine consists entirely of simple state sets. For each state set the vital state and its transitions can be determined immediately from the grammar forest. Usually the analysis will indicate that some vital states are combined into nonsimple state sets, and part of the analysis must be repeated for these state sets. One continues until no new state sets arise. For most grammars the analysis will converge rapidly.

If a machine had all simple states, the number of transitions, D_T, would be equal to Q where

$$Q = \sum_{i=(\text{vital states})} \left| \bigcup_{\substack{A \text{ causes} \\ \text{a transition} \\ \text{from state } i}} L_A^* \right|$$

where $|L|$ indicates the number of elements in set L. If all nonsimple state sets contain only vital states that do not appear in other state sets, D_T will be less than Q. Most other cases will cause D_T to be greater than Q. One can calculate Q in $O(TN)$ steps if an appropriate algorithm is used to calculate L_A^* (See 9).

6. EXPERIMENTAL RESULTS

Table 1 gives several experimental results for the size of the deterministic machine. The number of states can be estimated with an error of 12% from just the grammar size. Using $T + 1 - R_I$ permits estimating the number of states for LALR(1) grammars with an error of only 5% (R_I was used rather than R because it is much easier to calculate).

The simple estimates of the number of transitions from the grammar size or vocabulary size give only a rough estimate of the average number of transitions per state. The data does not even give a clear indication of what function of S or N should be used. An examination of D_T/D_S vs. N for the LALR(1) data showed that D_T/D_S increased rapidly for $4 \leq N \leq 15$, and much more slowly for $N > 15$. There were only 19 LALR(1) grammars with more than 15 symbols. Fitting each region separately gave

$$D_T/D_S = 0.998 + 0.113N \pm 0.286 \quad (4 \leq N \leq 15)$$

$$D_T/D_S = 2.183 + 0.038N \pm 0.805 \quad (16 \leq N).$$

The two results are equal for N just under 16. The other fits were examined on each set of LALR(1) grammars, but only D_T/D_S vs. S gave significantly different fits for the two regions. The non-LALR(1) grammars all had 17 or less symbols so separate fits were not made for the two regions.

The estimate of D_T from Q was much better than those from the N and S, but the error was still bigger than that of the estimate for the number of states. Evidently there is a lot of variation in the average size of L_A^* for various grammars.

Using the data in tables 1 and 3 it is possible to do a rough comparison between the space for LR(k) parser and precedence parser (such as weak precedence (10)). To store an LALR(1) parser in the way indicated by DeRemer (ignoring the space for lookahead transitions and the space saved by overlapping transition tables) requires about $(D_S + N_N)\ (\log_2 D_T) + 2D_T\ (\log_2 D_S)$ bits. Using $D_T \simeq (1.7 + 0.04N)\ D_S$, $D_S \simeq T$, $N_N \simeq 0.5N$, and $T \simeq 1.4N$, this reduces to about $1.9N \log_2(2.4N + 0.06N^2) + (4.8N + 0.12N^2)\log_2(1.4N)$ bits. If the lookahead and overlap effects are included, my measurements indicate that the size of the parser is reduced by about a factor of 2.

For weak precedence, if the precedence matrices are stored with no compacting, the required storage is $S(\log_2 N) + N(2N - N_N)$ bits. Using $N_N \simeq 0.5N$ and $S \simeq 1.1T \simeq 1.6N$, this reduces to about $1.6N \log_2 N + 1.5N^2$ bits. Thus for N in the range usually considered for parsers (a few hundred) the LR(k) method should take less storage. If the f and g functions of Floyd(1) are used for > and if < is ignored, then a weak precedence parser required about $S(\log_2 N) + (2N - N_N)\log_2 N$ bits, which is about 3.1NlogN bits. Thus for N in the range of a few hundred, this version of weak precedence should require somewhat less storage, assuming the technique will work on the required grammar. In many cases the size of the grammar must be increased to use weak precedence and this effect has not been considered in the above analyses.

These results suggest that for large LR(k) type parsers a major problem is to find a more efficient way to store the transition table. One method to do LR(k) parsing with a small table is to use Early's(12) algorithm, but more work is needed to see if it can be adapted to produce parsers which run as fast as those produced by DeRemer's method(1).

APPENDIX: THE GRAMMAR

The grammars for this study consist of all the published, reduced grammars from a set of grammars selected to test the author's parser generating program except for the Algol 60 grammar(13) (which is too large for the current version of the parser building program) and the EXP(2) grammar(3) (which is an example from a set of grammars for which the results of this paper do not hold). They were selected to include both a large number of programming language grammars (to test the parser builder for its expected use) and a large number of grammars from articles on grammar theory (to find bugs that would show up only under unusual conditions). The author feels that this same set of grammars should be useful for predicting the size of parsers and for indicating when the prediction formulas are not very accurate (by causing large error terms).

From each source every reduced grammar that was presented in the format of a context free grammar was selected (except EXP(2) in (3)). If the start symbol was used on the left, a new start symbol and production of the form

$$S_0 \rightarrow S$$

was added where S was the old start symbol and S_0 was not used in the original grammar. If the resulting parser required lookahead past the final state, the initial production

$$S_0 \rightarrow S \ \$$$

was added instead (where \$ was not used in the original grammar). When duplicate grammars were found, only the first published one was retained.

The sources of the grammars were DeRemer(1), 6 grammars; Early(12), 12 grammars; Floyd(11), 2 grammars; Floyd(14), 4 grammars: Griffith and Petrick(15), 11 grammars; Hopcroft and Ullman(16), 31 grammars; Korenjak(17), 1 grammar; Knuth(2), 10 grammars; McKeeman, Horning and Wortman(18), 44 grammars; Pager(19), 1 grammar; Williams(20), 6 grammars; Wirth and Weber(4), 2 grammars; and Wise(21), 1 grammar.

The grammars of Wirth and Wber(4) were modified by adding to the end of each production that had non null semantics a nonterminal of the form P_i and a production $P_i \rightarrow \varepsilon$. The . . .'s in the grammar of McKeeman, Horning, and Wortman(18) were ignored.

Some of the characteristics of the grammar set are given in table 3. The non-LALR(1) grammars included only one grammar with more than 15 symbols (it had 17). There were 4 grammars with 30 to 70 symbols and 4 grammars with more than 70 symbols. The lack of numerous large grammars was probably the main draw back to the data used for this study. The few large grammars did, however, have a major effect on the slope of the fits.

Summary of Results					
x	y	a	b	standard error	case
S	N_S/x	0.810 0.816	-0.00009 -0.00002	±0.087 ±0.080	ALL LALR(1)
S	N_T/x	1.224 1.224	0.0004 0.0004	±0.103 ±0.093	ALL LALR(1)
S	D_S/x	0.620 0.624	-0.0001 -0.0001	±0.073 ±0.068	ALL LALR(1)
$T+1-R_I$	D_S/x	0.983 0.977	0.00008 0.00002	±0.072 ±0.050	ALL LALR(1)
S	D_T/D_S	1.779 1.778	0.014 0.014	±0.648 ±0.647	ALL LALR(1)
N	D_T/D_S	1.723 1.677	0.042 0.043	±0.581 ±0.516	ALL LALR(1)
$N^{1/2}$	D_T/D_S	0.363 0.192	0.582 0.611	±0.587 ±0.485	ALL LALR(1)
Q	D_T/x	1.055 1.014	-0.00007 -0.00006	±0.173 ±0.114	ALL LALR(1)
N	T/x	1.524 1.376	0.002 0.003	±0.687 ±0.362	ALL LALR(1)

Table 1. Least squares fit of $y_1 = a + bx_1 + e_1$ where e_1 is an error term to 134 grammars and to 84 LALR(1) grammars.

Types of States	All Grammars	LALR(1)
Simple	2835	2190
Simple Replacement	447	359
Other Replacement	7	5
Chain	42	7
Complex	10	7
TOTAL	3341	2568

Table 2. The number of states of various types from the machine built with the test grammars.

Properties of the Grammars	All	LALR(1)		Max.(Grammar)
		N≤15	N>15	
Number of grammars	134	65	19	
Number of terminal symbols	922	299	437	77(Euler(4))
Number of nonterminal symbols	949	254	466	128(Euler(4))
Number immediately left recursive symbols	153	37	89	21(Algol(17))
Number of productions	1693	442	838	170(Algol(17))
Length of productions	3332	815	1645	323(Algol(17))
Tree length	3049	761	1556	310(Algol(17))

Table 3. The number of various items used in the grammars and the maximum value for any grammar.

1. DeRemer, F.L. Practical Translators for LR(k) Languages, Project MAC TR-65 MIT(1969). also U.S. Clearinghouse AD 699501. Much of this information is also in DeRemer,F.L. Simple LR(k) Grammars. CACM 14(1971) p. 453-460.

2. Knuth, D.E. On the Translation of Languages from Left to Right. Info. and Control 8(1965),p. 607-639.

3. Reynolds, John, See Early, J.C. An Efficient Context-Free Parsing Algorithm, thesis, Carnegie-Mellon U., 1968, p. 128-129.

4. Wirth, N. and Weber, H. EULER: A Generalization of ALGOL, and its Formal Definition. CACM 9(1966), p. 13-23, 89-99.

5. Horning, J.J. and Lalonde, W.R. Empirical Comparison of LR(k) and Precedence Parsers. Computer Systems Research Group, U. Of Toronto, 1971.

6. Anderson, T. Eve, J., and Horning, J.J. Efficient LR(1) Parsers. Computing Laboratory, 1969.

7. Kemp, R. Grosse Von LR(0)-Akzeptoren. Théorie des automates des langages et de la programmation (résumés des communications) 1972.

8. Aho, A.V. and Ullman, J.D. A Technique for Speeding Up LR(k) Parsers. Proceedings of Fourth Annual ACM Symposium on Theory of Computing. 1972, p. 251-263.

9. Purdom, P.W. A Transitive Closure Algorithm. BIT 10 (1970), p. 74-94.

10. Ichbiak, J.D. and Morse, S.P. A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars. CACM 13(1970), p. 501-508.

11. Floyd, R.W. Syntactic Analysis and Operator Precedence. JACM 10 (1963), p. 316-333.

12. Early, J.C. An Efficient Context-Free Parsing Algorithm, thesis, Carnegie-Mellon U., 1968.

13. Naur, P. and Woodger, M. (Eds.) Revised report on the algorithmic language ALGOL 60. CACM 6, (1963), 1-20.

14. Floyd, R.W. Bounded Context Syntactic Analysis. CACM 7, (1964), p. 62-67.

15. Griffith, T.V. and Petrick, S.R. On the Relative Efficiencies of Context-Free Grammar Recognizers. CACM, 8(1965), p. 289-300.

16. Hopcroft, J.E. and Ullman, J.D. Formal Languages and their Relation to Automata, Addison-Wesley, Reading, 1969.

17. Korenjak, A.J. Deterministic Language Processing, thesis, Princeton U., 1967.

18. McKeeman, W.M., Horning, J.J., and Wortman, D.B. A Compiler Generator, Prentice-Hall, Englewood Cliffs, 1970.

19. Pager, David. A Solution to An Open Problem by Knuth. Info. and Control, (1970), p. 462-473.

20. Williams, J.H. Bounded Context Parsable Grammars, U. of Wisconsin, Computer Science Report, No. 58, 1969.

21. Wise, D. S. An Improvement to Domelki's Algorithm, U. of Wisconsin, Computer Science Report, No. 94, 1970.

Cover design by Indiana University Publications