BIT 14 (1974), 326–337

THE SIZE OF LALR (1) PARSERS

PAUL PURDOM

Abstract.

Simple statistical formulas are given for the size of an LALR(1) parser. For practical grammars the number of states of an LALR(1) parser is linear with grammar size.

1. Introduction.

DeRemer [1] has developed methods for building $LR(k)$ type parsers which usually have a much smaller number of states than a parser built by Knuth's [2] original algorithm. The running time for any $LR(k)$ type parser is proportional to the length of the input to be parsed. Therefore, when comparing a parser produced by DeRemer's method with any other linear time parser, one important consideration is the amount of space required to store each parser.

The following set of grammars (similar to a set found by Reynolds[3]) shows that in the worst case $LR(k)$ methods can require unacceptable amounts of space:

$$\begin{aligned}
&R \to A_i u_i && (1 \leqq i \leqq n)\\
&A_i \to c_j A_i v_j && (1 \leqq i \neq j \leqq n)\\
&A_i \to c_i B_i w_i \mid d_i x_i && (1 \leqq i \leqq n)\\
&B_i \to c_j B_i y_{ij} \mid d_i z_i && (1 \leqq i, \quad j \leqq n)\,,
\end{aligned}$$

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. Since each production contains a unique symbol, it is trivial to parse sentences for any of these grammars in linear time using a copy of the grammar indexed by the last symbol in each production. This takes $O(n^2 \log n)$ bits for the n^{th} grammar. This grammar can also be parsed with an $LR(0)$ parser.

Such a parser has $O(n2^n)$ states and $O(n^2 2^n)$ transitions. It would require $O(n^2 2^n \log n)$ bits to store the tables for the parser in the way indicated by DeRemer [1].

Although the worst case space for tables built by DeRemer's method

Received Jan. 17, 1974. Revised March 2, 1974.

is exponential in the size of the grammar, the rest of this paper will show that in practice ("in practice" means average over the grammars that people devise when they are not trying to demonstrate the worst case) DeRemer type parsers have a number of states that increase linearly with the grammar size. The number of transitions increases as the square of the grammar size. It has already been shown by DeRemer [1], Horning and LaLande [4], and Anderson, Eve, and Horning [5] that $SLR(1)$ parsers are of reasonable size for several large grammars. Furthermore Kemp [6] recently published formulas for an upper limit on the number of states in an $LR(0)$ parser which can be used to show some unusual conditions that a set of grammars must satisfy for there to be exponential growth in the size of the parsing tables.

The statistical results of this paper were obtained by considering 84 published $LALR(1)$ 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 [6] the formulas of this paper do not include the final state. Thus he always obtains one more state.

Statistical results are given for two stages in the construction of the parsers. The first batch of results concerns the size of the finite automaton for recognizing possible handles. Since this automation is finite, it has a unique minimum state version. Building this handle-finding automaton is the first step of building a simple $LR(k)$ parser, whether the methods of DeRemer [1] or those of Aho and Ullman [7] are used. The possible exponential space requirement arises from building this automaton. The remaining steps in building the parser can have only a limited effect on the size of the parser. Thus knowing the size of the handle-finding automaton is fundamental to understanding the size of simple $LR(k)$ parsers.

The second batch of results concern the size of complete optimized parsers. $LALR(1)$ parsers are built and stored in the way suggested by DeRemer [1]. The only significant difference is that these parsers have the lookahead information stored with the transitions rather than with the states. Each state has a transition under each symbol that can come next.

The following optimizations are done to the parser (most of which are discussed in more detail in DeRemer [1]): 1) Chains of pop states where no more than one pop state in the chain calls for semantic action are replaced by single pop states (one for each place the chain can be entered). The same is done for chains with one state that pops -1 (i.e. puts its state number on the stack). 2) States which pop 0 and do not

call for semantic action are eliminated. 3) Transitions to lookback states from lookback and from pushread states are directed to the place the lookback state would go with the implied state-number on the top of the stack. 4) A modification for the reduction algorithm for finite state machines is performed to combine equivalent states. 5) States where all transitions are lookahead transitions and which always go to the same next state are eliminated if they are read states and converted to pop -1 states if they are pushread states. Optimizations 1 to 5 are repeated until no further improvement can result. 6) Lookahead transitions to read states are directed to where the corresponding transition of that read state goes (and converted to a non-lookahead transition if the corresponding transition was not a lookahead transition). 7) Transitions from different states which involve the same symbol, next state, and lookahead flag are stored in the same place so long as the transitions from each state can occupy consecutive locations. To summarize, the final parser has had all the obvious improvements made to reduce the size of the parsing tables and to increase their running speed.

2. Notation.

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 non-terminal. In the tree for any non-terminal A, each node on the k^{th} level can be reached by a unique string of symbols of length $k (k \geqq 0)$ 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 $\sharp_i$ (which is not used elsewhere in the grammar) to the right end of the i^{th} production for each production in the grammar.

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 $L_A{}^+ \cup A$. A symbol A is left recursive if $A \in L_A{}^+$.

The following symbols are used in the statistical study of parser size:

G = size of grammar, sum of the lengths (in symbols) of the right side of productions plus the number of productions.

L = the number of non-root nodes in the grammar forest for the augmented grammar. This is equal to G minus the length of the initial segment of some previous production with the same left side.

R = the number of immediately left recursive nonterminals. A nonterminal, A, is immediately left recursive if $A \in L_A$.

V = the number of symbols (terminal plus nonterminal in the grammar).

S = the number of states in the parser (not counting the final state).

T = the number of transitions in the parser.

$$Q = \sum_{\substack{i = \text{node of the augmented grammar tree (omitting all roots except the root of the tree for the start symbol).}}} \bigcup_{\substack{A \text{ labels an arc out of node } i}} |L_A{}^*|$$

where $|L_A|$ denotes the number of elements in set L_A. One can calculate Q in $O(L^2)$ steps if an appropriate algorithm is used to calculate $L_A{}^*$ (see [8]). F is used as a subscript on S and T when the final optimized parser (rather than the initial handle-finder) is referred to. For the augmented grammar

$$\begin{aligned} S &\to E \dashv \#_1 & T &\to F \uparrow T \#_4 \\ E &\to E + T \#_2 & T &\to F \#_5 \\ E &\to T \#_3 & F &\to i \#_6 \\ & & F &\to (E) \#_7 \end{aligned}$$

one has $G = 21$, $L = 20$, $R = 1$, $V = 10$, $S = 13$, $T = 30$ (compare S and T with fig. 4.1 in DeRemer [1]), and $Q = 29$.

3. Statistical Result.

Table 1 summarizes the statistical results. Least square fits to equations of the form $y_i = a + bx_i + e_i$ were done using formulas given in Fisz [9]. Table 1 gives the quantity used for y, the quantity used for x, the

expected value for a and its standard error, the expected value for b and its standard error, and the expected error of a point.

Some pairs of fits are very similar. For example table 1 gives both a fit for S vs. G and for S/G vs. G. As might be expected (since the errors are small and $a \approx 0$ for the first fit) the value of b in the first fit is nearly equal to the value of a for the second fit. The formulas used for estimating, however, assume that the value of the error e_i is independent of the value of x. In most cases when S or T is used for y, the average error increases with increasing x. On the other hand when S/x or T/x is fitted to x, the average error increases with very small values of x. Since least square fitting is robust (i.e. it gives good answers even when the data deviate from the assumptions of the method), each member of a pair of fits gives a good indication of how to predict y from x. The fits to S/x or T/x, of course, emphasize the small values of x, whereas the fits to S or T emphasize the large values.

Some of the more important fits from table 1 are plotted in figures 1 through 4. Each figure shows the three lines: $y=a+bx$, $y=(a+\Delta a)+(b+\Delta b)x$, and $y=(a-\Delta a)+(b-\Delta b)x$ where Δa and Δb are the expected errors in a and b. In some cases the error is so small that the three lines may appear to be one line. The center of each vertical line gives a y value. The length of each vertical line is twice the expected error ($\sqrt{\langle e_i^2 \rangle}$

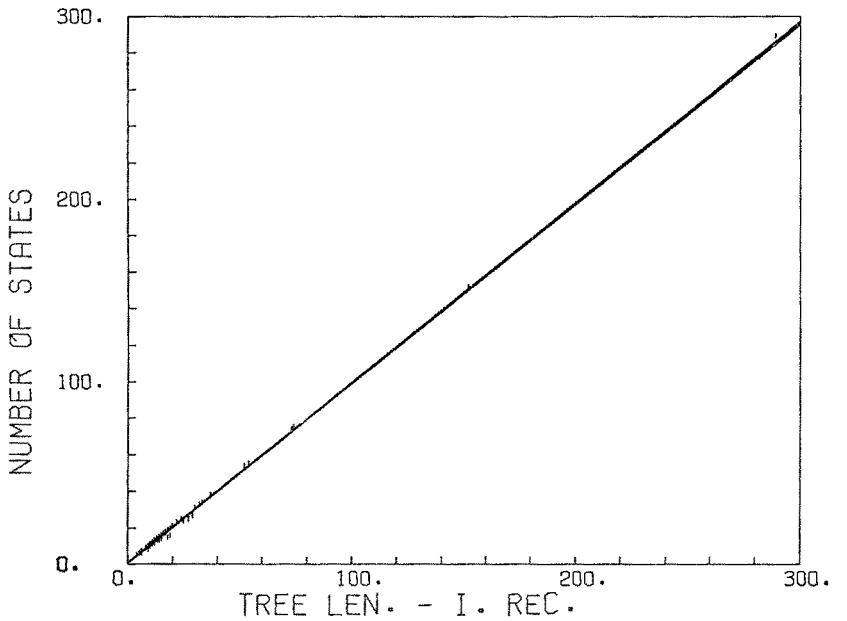

Fig. 1. Number of states for the handle finder.

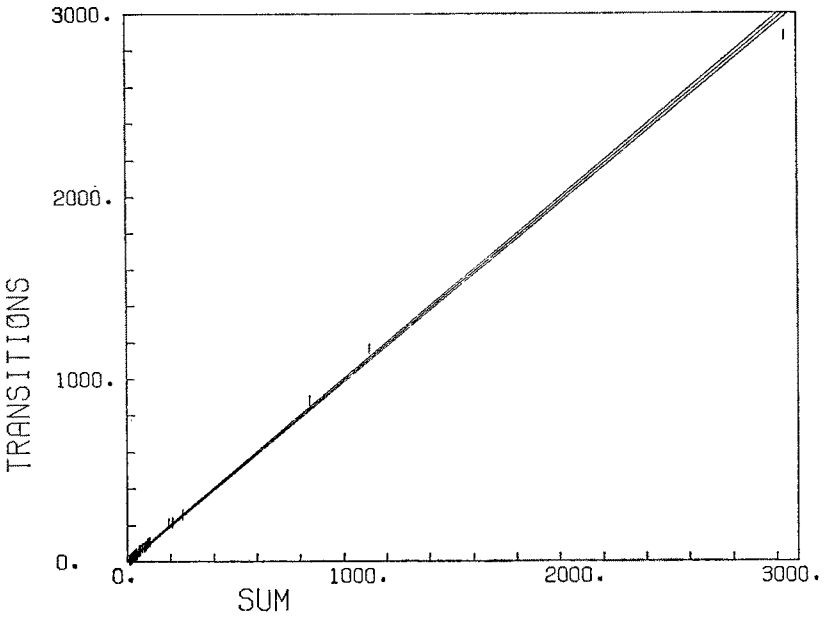

Fig. 2. Number of transitions for the handle finder.

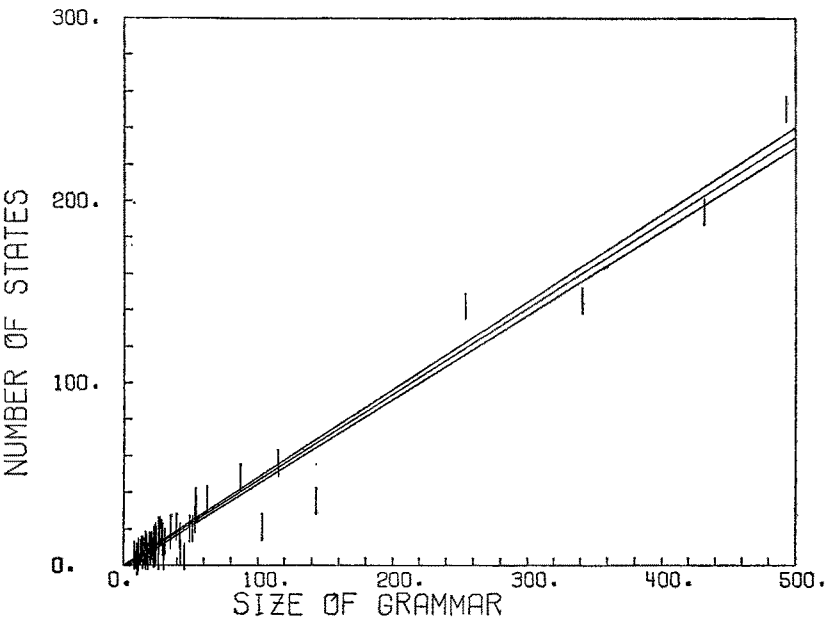

Fig. 3. Number of states for the optimized parser.

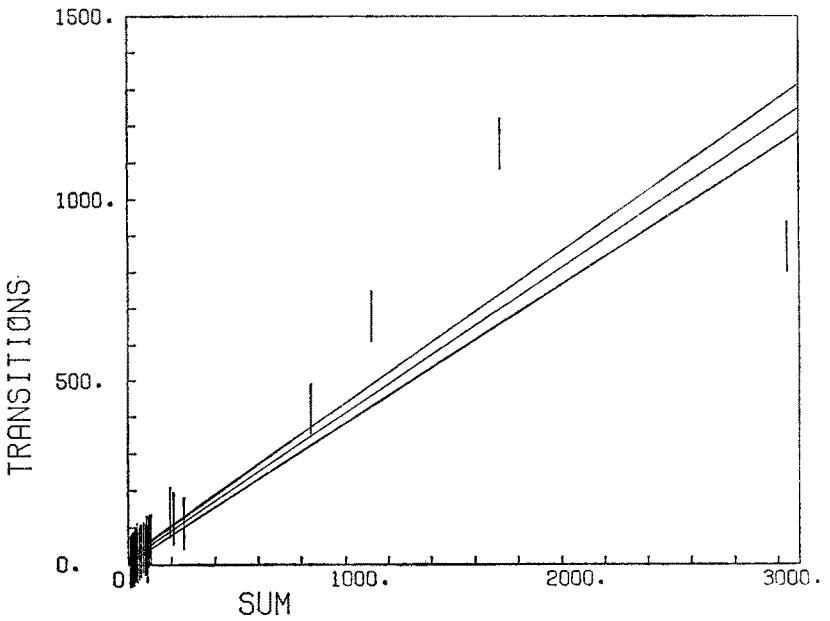

Fig. 4. Number of transitions for the optimized parser.

from table 1). For small values of x there are often several points for a single x value, so vertical lines may overlap.

The number of states in the handle finding automaton can be predicted to within a few states from the grammar size or from $L-R$ to within about 1 state. The number of states in the final parser can be predicted to within about 10 states from the grammar size and $L-R$ does not result in a much better prediction.

A rough estimate of the number of transitions per state can be made from the vocabulary size (error about 30%). A much better estimate of the number of trasitions can be obtained from Q, but Q takes much longer to compute than any of the other quantities used for predicting.

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 requires about $S_F(2+\log_2 S_F+\log_2 T_F)+2T_F \log_2 S_F$ bits. This reduces to about $G+1.2G \log .5G+0.01G^2 \log .5G$ bits.

For weak precedence, if the precedence matrices are stored with no compacting, the required storage is $G(\log_2 V)+V(2V-V_N)$ bits, where V_N is the number of nonterminal symbols.

This reduces to about $G \log_2 0.4G+0.24G^2$ bits. Thus for G in the range usually considered for parsers (a few hundred) the $LR(k)$ method will

take less storage. If the f and g functions of Floyd [11] are used for $\rangle$ and if $\langle$ is ignored, then a weak precedence parser required about $S(\log_2 V) + (2V - V_N)\log_2 V$ bits, which is about $1.5G \log .4G$ bits. Thus for G in the range of a few hundred, this version of weak precedence will 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 analysis.

A major problem for large $LR(k)$ type parsers 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].

4. Factors affecting the size of the parser.

The formulas for the size of the handle finding automaton come from the following observation: there is an almost one to one correspondence between the states of the nondeterministic finite automaton and the states of the deterministic automaton that DeRemer [1] builds from it. The states of the deterministic machine correspond to sets of states in the nondeterministic machine. Usually one can pick a state (of the nondeterministic machine) from (nearly) each state set such that each state is picked about (no more than) once.

A state of the nondeterministic machine that arises from the root of a tree in an augmented grammar forest (except the tree for the start symbol) is called a nonvital state. Any other state is called a vital state. The number of vital states is $L+1$.

The following definitions indicate the type of state sets that can arise when building the deterministic automaton.

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.

Using these definitions to set up a correspondence between vital states and state sets and also considering the effect of recursive symbols (they cause other replacement or complex state sets) one can conclude that for an automaton with no complex state sets, the number of states, S, obeys the relation $S \leqq L+1-R'$ where R' is the number of recursive symbols.

A detail consideration of how the machine is built indicates that rather unusual conditions must arise in the grammar for the machine to have a complex state set. Furthermore producing one complex state makes it easier to produce other complex states only in a small part of the machine (that part which processes the same productions that the first complex state was processing). Of course, in pathological cases complex states arise all over everywhere and exponential growth results. Similar considerations indicate that unusual conditions must arise for S to be smaller than $L+1-R'$.

The number of state sets of various types which were observed is given in Table 2.

Similar considerations indicate that the number of transitions should be about Q.

The notation of this section can be used to prove that the deterministic automaton is automatically reduced if it is built by a good algorithm (one that builds only acceptable state sets). The proof depends on the fact that each production of the augmented grammar ends with a unique symbol ($\#_i$).

The ideas of this section are covered in greater detail elsewhere [13].

Appendix: The Grammars.

The grammars for this study consist of all the published, reduced, $LALR(1)$ grammars from a set of grammars selected to test the author's parser generating program except for the Algol 60 grammar [14] (which is too large for the current version of the parser building program) and the EXP set of grammars [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).

When necessary a production of the form $S_0 \rightarrow S$ or $S_0 \rightarrow S\$$ was added to each grammar so that the start symbol would not appear on the left and so that no lookahead would be needed past the end of input.

The sources of the grammars were DeRemer [1], 6 grammars; Early [12], 12 grammars; Floyd [11], 2 grammars; Floyd [15], 4 grammars; Griffith and Patrick [16], 11 grammars; Hopcroft and Ullman [17], 31 grammars; Korenjak [18], 1 grammar; Knuth [2], 10 grammars; McKeeman, Horning and Wortman [19], 44 grammars; Pager [20], 1 grammar; Williams [21], 6 grammars; Wirth and Weber [22], 2 grammars; and Wise [23], 1 grammar.

The grammars of Wirth and Weber [22] 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 $P_i \to \varepsilon$. The ...'s in the grammar of McKeeman, Horning, and Wortman [19] were ignored.

Some of the characteristics of the grammar set are given in Table 3. There were 4 grammars with 30 to 70 symbols and 4 grammars with more than 70 symbols.

y	x	a	b	$\sqrt{\langle e_i^2 \rangle}$
S	G	0.02 ± 0.45	0.5949 ± 0.0048	3.6
S/G	G	0.6235 ± 0.0084	-0.000147 ± 0.000091	0.069
S	$L-R$	0.83 ± 0.16	0.9845 ± 0.0028	1.3
$S/(L-R+1)$	$L-R+1$	0.977 ± 0.0062	0.00002 ± 0.00011	0.051
S_F	G	-0.48 ± 0.88	0.4707 ± 0.0095	7.1
S_F/G	G	0.479 ± 0.018	-0.00010 ± 0.00020	0.15
S_F	$L-R$	0.18 ± 0.84	0.778 ± 0.015	6.9
$S_F/(L-R+1)$	$L-R+1$	0.747 ± 0.027	0.00008 ± 0.00048	0.22
T/S	V	1.677 ± 0.065	0.0426 ± 0.0020	0.52
T/S	G	1.778 ± 0.080	0.01430 ± 0.00087	0.65
T	Q	3.4 ± 2.9	0.9867 ± 0.0069	26.
T/Q	Q	1.0182 ± 0.0069	-0.000002 ± 0.000017	0.061
T_F/S_F	V	1.401 ± 0.097	0.0224 ± 0.0029	0.77
T_F/S_F	G	1.434 ± 0.095	0.0080 ± 0.0010	0.77
T_F	Q	11.8 ± 7.8	0.398 ± 0.019	69.
T_F/Q	Q	0.572 ± 0.018	-0.000051 ± 0.000043	0.16

Table 1. *Least squares fits of the form* $y_i = a + bx_i + e_i$. *Expected standard errors are given for* a *and* b. *The last column gives the expected standard error for each point in the fit.*

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	LALR(1)		Max. (Grammar)
	$V \leqq 15$	$V > 15$	
Number of grammars	65	19	
Number of terminal symbols	299	437	77(Euler(4))
Number of nonterminal symbols	254	466	128(Euler(4))
Number immediately left recursive symbols	37	89	21(Algol(18))
Number of productions	442	838	170(Algol(18))
Length of productions	815	1645	323(Algol(18))
Tree length	761	1556	310(Algol(18))

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

REFERENCES

1. F. L. DeRemer, *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), 453–460.
2. D. E. Knuth, *On the Translation of Languages from Left to Right*, Info. and Control 8(1965) 607–639.
3. John Reynolds, See Early J. C., *An Efficient Context-Free Parsing Algorithm*, thesis, Carnegie-Mellon U., 1968, pp. 128–129.
4. J. J. Horning and W. R. LaLande, *Empirical Comparison of LR(k) and Precedence Parsers*, Computer Systems Research Group, U. of Toronto, 1971.
5. T. Anderson, J. Eve and J. J. Horning, *Efficient LR(1) Parsers*, Computing Laboratory, 1969.
6. R. Kemp, *An Estimation of the Set of States of the Minimal LR(0)-Acceptor*, Automata, Languages and Programming, M. Nivat, editor, North Holland, 1973.
7. A. V. Aho and J. D. Ullman, *A Technique for Speeding Up LR(k) Parsers*, Proceedings of Fourth Annual ACM Symposium on Theory of Computing. 1972, pp. 251–263.
8. P. W. Purdom, *A Transitive Closure Algorithm*, BIT 10 (1970), 74–94.

9. M. Fisz, *Probability Theory and Mathematical Statistics*, John Wiley and Sons, 1963, pp. 535–538.
10. J. D. Ichbiak and S. P. Morse, *A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars*, CACM 13 (1970), 501–508.
11. R. W. Floyd, *Syntactic Analysis and Operator Precedence*, JACM 10 (1963), 316–333.
12. J. C. Early, *An Efficient Context-Free Parsing Algorithm*, thesis, Carnegie-Mellon U., 1968.
13. P. W. Purdom, *The Size of LR(0) Machines*, Tech. Report 2, Indiana University.
14. P. Naur and M. Woodger (Ed.), *Revised report on the algorithmic language ALGOL 60*, CACM 6 (1963), 1–20.
15. R. W. Floyd, *Bounded Context Syntactic Analysis*, CACM 7 (1964), 62–67.
16. T. V. Griffith and S. R. Petrick, *On the Relative Efficiencies of Context-Free Grammar Recognizers*, CACM, 8 (1965), 289–300.
17. J. E. Hopcroft and J. D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, 1969.
18. A. J. Korenjak, *Deterministic Language Processing*, thesis, Princeton U., 1967.
19. W. M. McKeeman, J. J. Horning and D. B. Wortman, *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, 1970.
20. David Pager, *A solution to An Open Problem by Knuth*, Info. and Control (1970), 462–473.
21. J. H. Williams, *Bounded Context Parsable Grammars*, U. of Wisconsin, Computer Science Report, No. 58, 1969.
22. N. Wirth and H. Weber, *EULER: A Generalization of ALGOL, and its Formal Definition*, CACM 9 (1966), 13–23, 89–99.
23. D. S. Wise, *An Improvement to Domelki's Algorithm*, U. of Wisconsin, Computer Science Report, No. 94, 1970.

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
BLOOMINGTON, INDIANA 47401
USA