Programming Languages

T.A. Standish Editor

A Parser-Generating System for Constructing Compressed Compilers

M.D. Mickunas and V.B. Schneider
Purdue University

This paper describes a parser-generating system (PGS) currently in use on the CDC-6500 computer at Purdue University. The PGS is a Fortran-coded program that accepts a translation grammar as input and constructs from it a compact, machine-coded compiler. In the input translation grammar, each BNF syntactic rule corresponds to a (possibly empty) "code generator" realizable as an assembly language, Fortran or Algol, subroutine that is called whenever that syntactic rule is applied in the parse of a program.

Typical one-pass compilers constructed by the PGS translate source programs at speeds approaching 14,000 cards per minute. For an XPL compiler, the parser program and its tables currently occupy 288 words of 60-bit core memory of which 140 words are parsing table entries and 82 words are links to code generators.

Key Words and Phrases: parser generators, translator writing systems, syntactic analysis, normal-form grammars, pushdown automata, translation grammars, translator optimization, compression algorithm

CR Categories: 4.12, 5.22, 5.23

Work on this paper was supported by NSF Grant GJ-851 to Purdue University. Authors' address: Computer Sciences Department, Purdue University, Lafayette, IN 47907.

1. Introduction

The PGS is a FORTRAN-coded system that accepts a "translation grammar" [8, 9, 10] as input and constructs from it a compact, machine-coded parser. Each BNF rule of the grammar corresponds to a "code generator" that is called whenever that syntactic rule is applied in a parse of some program. Because the PGS assumes very little about its code generators, the nature and quality of object code generated by a constructed compiler depends on the language implementor's discretion. (However, prepackaged systems of code generators for implementing a family of PL/I-like languages are available, and other systems are being developed at Purdue.) In fact, object code need not be generated by a compiler; an implementor could choose instead to construct program trees, produce macro expansions, or execute source programs interpretively.

The PGS operates in five distinct phases. In the first phase, the translation grammar input to the PGS is converted into a simplified normal form for internal use. The second phase computes the contexts in which each rule of the resulting grammar is to be applied during the process of parsing. The third phase uses these contexts and the normal-form rules to generate the table description of a bounded-context pushdown automaton translator system. The fourth phase compresses the table description so as to minimize the number of tests necessary in the syntactic portion of the compiler under construction. The fifth phase generates a table-driven compiler that is linked to the code generators by subroutine calls.

2. Notation and Basic Definitions

A translation grammar (TG) is a six-tuple (V, T, Q, S, P, R) where:

(a) V is a finite set of symbols called the *vocabulary*.
(b) T is a proper subset of V called the *terminals*.
(c) Q is a finite set of symbols, including the empty symbol e, called *code generators*.
(d) S is a distinguished member of $V - T$ called the *initial symbol*.
(e) P is a finite set of syntactic rules P_i of the form $A \rightarrow x$, where $A \neq x$, A is in $V - T$, and x is in V^+.
(f) R is a single-valued function from P into Q.

V^* denotes the set of all strings composed of symbols from V, including the empty string (denoted by e); and $V^+ = V \times V^*$. A is called the *left part* and x the *right part* of any rule of the form $P_i = A \rightarrow x$. Early capital letters, $A, \ldots, S$ are in $V - T$, and early small letters a, b, c are in T. Late small letters $u, \ldots, z$ are in V^*, except when they appear as the right parts in rules (in which case, they are in V^+). Late capital letters are in V.

If $A \rightarrow w$ is a rule, an *immediate derivation* of one

string $y = uwv$ from another $x = uAv$ is written $x \Rightarrow y$. The transitive completion of this relation is a *derivation* and is written $x \overset{*}{\Rightarrow} y$, which means that there exists a sequence of strings $(w_0, w_1, \ldots, w_n)$ such that $x = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n = y$ for $n \geq 0$. For the canonical derivation, we choose the right derivation in which each step is of the form $uAv \Rightarrow uwv$, with v in T^*.

x is a *sentence* of G if x is in T^+ and x can be derived from S. The *language* of a TG is then the set of sentences that can be derived from S in G:

$$L(G) = \{x : (S \overset{*}{\Rightarrow} x) \ \& \ (x \in T^+)\}.$$

If x is in $L(G)$, then a *parse* of x (written Parse(x)) is the sequence of rules $(P_1, \ldots, P_n)$ such that P_j immediately derives w_{j-1} from w_j $(j = 1, \ldots, n)$ and $x = w_0$, $S = w_n$. Corresponding to the parse of x is the translation of x (written T(Parse(x)) consisting of the sequence of code generators $(R(P_1), \ldots, R(P_n))$.

If $(P_1, \ldots, P_n)$ is a parse of string x into symbol S, there exists a permutation of $(P_1, \ldots, P_n)$ that is *leftmost*; i.e. its rules cause a right derivation when taken in reverse order and applied to S. We define an *unambiguous* grammar G to be one in which every x in $L(G)$ has exactly one leftmost parse, denoted by LParse(x). The translation $T(L\text{Parse}(x))$ will be the canonical code generation sequence of our sentence x. Of course, this code generation sequence for x may include "error messages" produced by one or more code generators, in which case x is syntactically well formed, but anomalous for some semantic reason.

We next define a normal form for TG's, in terms of which a left-most parsing algorithm can be designed. This normal form simplifies the design of a conversion process from TG's into leftmost parsing algorithms by standardizing the calculations necessary for converting syntactic rules into decisions of the parser.

3. Normal Form of Translation Grammars

A translation grammar $G = (V, T, Q, S, P, R)$ is said to be in *normal form* if all the rules in P are of the forms $K'_{r-1} \rightarrow K_{r-1}K_r$ or $K_r' \rightarrow K_r$ or $K_r' \rightarrow K_r a_s$ or $K_{r+1} \rightarrow a_s$. (The motivation for using indices r and s will be apparent in the next section.)

A simple algorithm exists for converting any TG H into a TG H' in normal form such that $L(H) = L(H')$ and $T(L\text{Parse}(x))_H = T(L\text{Parse}(x))_{H'}$ for all x in $L(H)$. Because of this algorithm, all derivations of sentences in $L(H)$ are in one-to-one correspondence with derivations of sentences in $L(H')$. The algorithm works as follows:

Step 1. Inspect the rules of H to find the largest subset of rules of the form $A \rightarrow a_1 \cdots a_j \cdots a_k A_{k+1} u$ where $k \geq j \geq 1$ and all rules in the subset have the same prefix $y = a_1 \cdots a_j$. In all rules of this subset, replace y by a new symbol A' and add the rule $A' \rightarrow y$ to the rules of H and the production $(A' \rightarrow y, e)$ to R of H. Continue Step 1 until no subsets of P remain to be merged.

Step 2. The rules of H that result from this prefix merging are either in normal form already (and so transfer into P') or are of the form $P_i = A \rightarrow X_1 \cdots X_n$, $(n \geq 2)$. Each such rule is transformed into the following sequence of normal form rules in P': $B_v \rightarrow B_{v+1}X_{n-v}$ for $v = 0, 1, \ldots, n-2$, where B_0 is A, and B_{n-1} is X_1, if X_1 is in $V - T$ of H; otherwise, an additional rule of the form $B_{n-1} \rightarrow X_1$ is included in P' of H'.

Step 3. The B_v are treated as new and distinct elements of $V' - T'$ of H'. Each new rule generated by this transformation becomes an additional production $(B_v \rightarrow B_{v+1}X_{n-v}, e)$ of R' for $v = 1, \ldots, n-2$. For $v = 0$, the production $(A \rightarrow B_1X_n, Q_i)$ goes into R', where $(A \rightarrow X_1 \cdots X_n, Q_i)$ is the corresponding production in R.

The fact that the B_v of the algorithm are "new and distinct" leads to a simple proof of the one-to-one correspondence between parses of sentences in $L(H)$ and $L(H')$. Since each rule of P corresponds to a unique rule or sequence of rules in P', it follows that, for every parse possible in H, there is a corresponding parse in H', and conversely. Hence, in particular, $T(L\text{Parse}(x))_H = T(L\text{Parse}(x))_{H'}$ for all x in $L(H)$. In addition, ambiguity in $L(H)$ is equivalent to ambiguity in $L(H')$.

4. Leftmost Parses and Normal Form Grammars

We introduce boundary markers # to the vocabulary of G. A new initial symbol S' now takes the place of S in G, and three new rules are added to P:

$$P_{n+1} = S' \rightarrow J_1\#$$
$$P_n = J_1 \rightarrow J_2 S$$
$$P_1 = J_2 \rightarrow \#$$

The corresponding productions in R are (P_1, e), (P_n, e), and (P_{n+1}, e); i.e. no added code is generated.[1] With these rules, boundary markers are made to occur at both ends of all strings produced by the grammar.

Let $w_0 = \#a_1 \cdots a_n\#$ be a string in the language of such a grammar. In the initial step of the leftmost parsing algorithm, rule P_1 is applied, yielding string $w_1 = J_2 a_1 \cdots a_n\#$. After j steps, w_0 has been reduced to $w_j = J_2 K_1 \cdots K_r a_s \cdots a_n\#$ $(1 \leq r < s \leq n+1)$. If w_0 is in $L(G)$, the leftmost sequence of rules $(P_1, \ldots, P_j)$ are precisely the first j reductions of the leftmost parse of w_0 to S'.

For the $(j+1)$-th reduction, five different cases must be distinguished.

Case 0. S' does not produce w_j.

If S' does produce w_j, we have to distinguish between the following possibilities.

[1] In practice, our TWS provides "system functions" to be performed immediately prior to and following the translation of a program.

Case 1. A rule of the form $P_{j+1} = K'_{r+1} \rightarrow a_s$ reduces w_j to w_{j+1}.

Case 2. A rule of the form $P_{j+1} = K_r' \rightarrow K_r$ reduces w_j to w_{j+1}.

Case 3. A rule of the form $P_{j+1} = K'_{r-1} \rightarrow K_{r-1}K_r$ reduces w_j to w_{j+1}.

Case 4. A rule of the form $P_{j+1} = K_r' \rightarrow K_r a_s$ reduces w_j to w_{j+1}. That only these cases need be considered is proved in [8].

In general, the decision concerning which of the Cases 1–4 apply for the $(j + 1)$-th step of a leftmost parse must be made in terms of context. (As an example, there may exist rules in the grammar having $K_{r-1}K_r$ and $K_r a_s$ on the right part.) To decide which case applies at a given step of the parse requires algorithms for discovering what symbols can legally be adjacent to the symbols being replaced in that step of the reduction while w_0 is a sentence of G. The algorithms to be given here are similar to those of Floyd [1] and Wirth and Weber [11] in that they construct all legal co-occurrences of triples of symbols in some language from that language's grammar.

Case A. For a rule of the form $P_{j+1} = K' \rightarrow K_r a_s$ (or $P_{j+1} = K' \rightarrow a_s$) to apply in the $(j + 1)$th reduction, the symbol K_{r-1} to the left of K' (or K_r to the left of K') must be present in some derivation of w_{j+1} from S'. That is, there must be some nonterminal Q for which $(Q \rightarrow K_{r-1}A \in P)$ & $(A \overset{*}{\Rightarrow} K'u)$ (or $(Q \rightarrow K_r A \in P)$ & $(A \overset{*}{\Rightarrow} K'u)$). Then, the pairs (K_{r-1}, a_s) (alternatively, (K_r, a_s)) are the *contexts* in which P_{j+1} applies.

Case B. A rule of the form $P_{j+1} = K'_{r-1} \rightarrow K_{r-1}K_r$ is applied in the context (K_{r-1}, a_s) if and only if there is some nonterminal Q for which $(Q \rightarrow AB \in P)$ & $(B \overset{*}{\Rightarrow} a_s v)$ & $(A \overset{*}{\Rightarrow} uK'_{r-1})$.

Case C. A rule of the form $P_{j+1} = K_r' \rightarrow K_r$ is applied in the context (K_{r-1}, a_s) if and only if there is some nonterminal Q for which either $(Q \rightarrow K_{r-1}A \in P)$ & $(A \overset{*}{\Rightarrow} Bv)$ & $(B \rightarrow CD \in P)$ & $(D \overset{*}{\Rightarrow} a_s u)$ & $(C \overset{*}{\Rightarrow} K_r')$ or $(Q \rightarrow AB \in P)$ & $(B \overset{*}{\Rightarrow} a_s v)$ & $(A \overset{*}{\Rightarrow} uC)$ & $(C \rightarrow K_{r-1}D \in P)$ & $(D \overset{*}{\Rightarrow} K_r')$.

After the contexts for some grammar have been determined as in Cases A, B, and C above, there may exist a subset of the rules for which, when their right parts and contexts are taken together, the resulting triples $K_{r-1}K_r a_s$ are the same. We deal with this situation by using a one-symbol lookahead algorithm that tests symbol a_{s+1} in the input sentence of the parser to decide which of the conflicting rules to apply as the next step in a parse. As a simple example, the necessity for this lookahead may arise in ALGOL 60 multiple assignment statements of the form $v := y := u + 1$; where the parser must look past the variable identifier to see whether it is followed by an assignment symbol $(:=)$ or an operator $(+)$. Appendix I gives the algorithm for calculating lookahead symbols.

Experience in producing ALGOL W and EULER compilers at Purdue has shown that one-symbol lookahead is adequate to handle parsers for these languages, and our XPL compiler [12] gets by without any one-symbol lookahead. Where one-symbol lookahead is inadequate for constructing a unique parser, the PGS currently in use signals a possible ambiguity in the grammar undergoing conversion. Another approach, which involves transforming $LR(k)$ grammars (by Knuth's criterion) into grammars that can be parsed in our system without lookahead, is currently being developed and tested at Purdue.

5. Pushdown Automaton Parsing Model

The most natural method for implementing leftmost parsers constructed from our normal-form grammars is to use a table-driven algorithm in which the parsing table contains the triples of symbols defined in Section 4. An automaton model for this algorithm was chosen, and we will refer to this model as a *bounded-context acceptor* (BCA).

Our BCA has the following inventory of vocabularies and auxiliary data structures: It has a finite set of states (each state identified with a region of the parsing table), a finite vocabulary consisting of symbols that occur in its source programs, and an auxiliary stack (also called a *pushdown store*) having a stack vocabulary. In addition, it uses an initial (or starting) state, a final (or accepting) state, and the same boundary symbol # that was introduced in Section 4.

During a parse, the BCA always operates in some state where it scans the topmost stack symbol and the next source program symbol. By applying one of the normal form rules of its grammar, the BCA transfers to another state, possibly consuming either the current stack symbol or source program symbol in the process, and possibly storing a new symbol on top of its stack.

The states, stack symbols, and source program symbols for a BCA can be determined by inspection of its normal-form grammar as follows:

1. The terminals of the grammar are the source program symbols.
2. The nonterminals K_{r-1} in rules of the form $K'_{r-1} \rightarrow K_{r-1}\, K_r$ are the stack symbols of the BCA.
3. The nonterminals K_r in rules of the form $K'_{r-1} \rightarrow K_{r-1}\, K_r$, $K_r' \rightarrow K_r a_s$, and $K_r' \rightarrow K_r$ are the states of the BCA, where each possible transition from state K_r corresponds to one of the leftmost parse reductions indicated by the three normal-form rules above.
4. In the starting state S_0, occur all transitions corresponding to leftmost parse reductions caused by rules of the form $K_{r+1} \rightarrow a_s$.
5. The final state is the initial symbol S' of the input grammar.

Our choice of stack and state symbols is motivated by the following observations: As defined in (2), the K_{r-1} stack symbols are the roots of parse subtrees which must await in sequence the construction of the next subtree in the leftmost parse before they may be joined

to that subtree by a rule $K_{r1} \rightarrow K_{r-1}K_r$. Conversely, the K_r state symbols are the roots of parse subtrees which may be joined *immediately* to a larger subtree of the leftmost parse through application of one of the rule forms in (3) above.

Thus, for our BCA scheme to work with a given normal-form grammar, its stack and state symbols must be disjoint. Appendix II contains a simple transformation that can be applied to all normal form grammars for ensuring disjointedness of stack and state symbols.

With these preliminaries, we arrive at the following table-driven parser definition: A BCA P is defined to be a seven-tuple $P = (Q, T, N, M, \#, S_0, F)$ where:

Q is the finite set of states.
T is the finite set of source program symbols.
N is the finite set of symbols in the stack vocabulary.
S_0 is the initial state and F is called the final state.
$T \cap N = \{\#\}$.
M is the table that maps a subset of $N \times Q \times T$ into the subsets of $\{e\} \times Q \times T \cup N \times Q \times \{e\} \cup N \times Q \times T \cup N \times N \times \{S_0\} \times (T \cup \{e\})$.

The BCA for a given normal-form grammar G accepts $L(G)$ when its M-table is constructed using the following correspondence between steps in a leftmost parse and states in the BCA for G (see [7] for a proof of this). The correspondence gives a four-step algorithm for constructing M-table entries, and transforms the initial grammar symbol S' into the final BCA state F.

Step I. Rule $A_i \rightarrow A_{i1}A_{i2}$ with contexts (A_{i1}, a_s):
If $A_i \in N$, then $(A_i, S_0, a_s) \in M(A_{i1}, A_{i2}, a_s)$.
If $A_i \in Q$, then $(e, A_i, a_s) \in M(A_{i1}, A_{i2}, a_s)$.
These transitions take care of all possibilities arising from Case 3 of the leftmost parsing algorithm.

Step II. Rule $A_k \rightarrow A_{k1}a_{k2}$ with contexts (K_{r-1}, a_{k2}):
If $A_k \in N$, then $(K_{r-1}A_k, S_0, e) \in M(K_{r-1}, A_{k1}, a_{k2})$.
If $A_k \in Q$, then $(K_{r-1}, A_k, e) \in M(K_{r-1}, A_{k1}, a_{k2})$.
These transitions take care of all possibilities arising from Case 4 of the leftmost parsing algorithm.

Step III. Rule $A_j \rightarrow a_s$ with contexts (K_r, a_s):
If $A_j \in N$, then $(K_rA_j, S_0, e) \in M(K_r, S_0, a_s)$.
If $A_j \in Q$, then $(K_r, A_j, e) \in M(K_r, S_0, a_s)$.
These transitions take care of all possibilities arising from Case 1 of the leftmost parsing algorithm.

Step IV. Rule $A_j \rightarrow A_{j1}$ with contexts (K_{r-1}, a_s):
If $A_j \in N$, then $(K_{r-1}A_j, S_0, a_s) \in M(K_{r-1}, A_{j1}, a_s)$.
If $A_j \in Q$, then $(K_{r-1}, A_j, a_s) \in M(K_{r-1}, A_{j1}, a_s)$.
These transitions take care of all possibilities arising from Case 2 of the leftmost parsing algorithm.

When all transitions of the BCA have been defined as described above, the PGS has completed the construction of the full M-table for a BCA that performs leftmost parses on the sentences of its language. By causing the BCA to emit code generator symbols from appropriate transitions of the M-table, we obtain a leftmost compiler.

6. Compression of M-Tables

The BCA parser contains more information than needed for a successful parsing algorithm. As we will demonstrate here, it is often not necessary to know all contexts for which one step of a derivation is valid. It suffices instead to know that the current reduction is permissible on the condition that its ignored left or right context is the correct symbol to appear in some later step of that derivation. For example, consider some BCA having a state A in which all transitions arise from n rules of the form $A_i \rightarrow A_{i1}A$ $(i = 1, \ldots, n)$. If the A_{i1} are all distinct symbols, then they alone are sufficient to determine the state transitions, and so we can write the following entries into the compressed M-table: $M(A_{i1}, A, *) = \xi_i \quad (i = 1, \ldots, n)$ where $\xi_i = (A_i, S_0, *)$ if A_i is a stack symbol, and $\xi_i = (e, A_i, *)$ if A_i is a state of the BCA. Note in the above that the asterisk (*) means that the corresponding input symbol of the BCA is ignored in determining the transition.

Likewise, we can compress a state B of the BCA in which all transitions arise from m rules of the form $B_j \rightarrow Bb_{j1}$ $(j = 1, \ldots, m)$: If the b_{j1} are all distinct symbols, then they alone are sufficient to determine unique state transitions, and so we can write the following entries into the compressed M-table: $M(*, B, b_{j1}) = \xi_j, \quad (j = 1, \ldots, m)$. An identical analysis applies to the initial state S_0, whose p entries all arise from rules of the form $C_k \rightarrow c_{k1} (k = 1, \ldots, p)$.

Next, suppose in the first example above that two or more of the symbols A_{i1} are not distinct. In this case, assuming that the full contexts (A_{i1}, a_{i1}) are distinct, the M-table for state A can be reordered so that those transitions requiring full context inspection are considered before those for which it is only necessary to inspect the A_{i1} symbol. Similar considerations apply for the second example above involving full-context inspection followed by right-context only inspection of the M-table entries for a given state.

In more complex situations, we are often faced with a combination of the first two examples above, namely that all transitions from a state C arise from n rules of the form $C_i \rightarrow C_{i1}C (i = 1, \ldots, n)$ and from m rules of the form $C_j' \rightarrow Cc_{j2}$ $(j = 1, \ldots, m)$. The further complication here is that one (or more) of the symbols C_{p1} $(1 \leq p \leq n)$ may also be a left context for some rule $C_q' \rightarrow Cc_{q2}'$ $(1 \leq q \leq m)$. In this case, the specification of C_{p1} alone in an M-table entry is not sufficient to determine uniquely which state transition, $C_p \rightarrow C_{p1}C$ or $C_q' \rightarrow Cc_{q2}'$ should apply. This anomaly can be removed by first inspecting the full context (C_{p1}, c_{q2}') to determine if the transition $C_q \rightarrow Cc_{q2}'$ should apply. Thereafter, inspection of C_{p1} alone is sufficient to determine the transition $C_p \rightarrow C_{p1}C$ and inspection of c_{q2}' alone is sufficient to determine the transition $C_q' \rightarrow Cc_{q2}'$. It could be argued that this anomaly is more easily removed by first inspecting c_{q2}' (via a right-context only entry) to determine if the transition $C_q' \rightarrow Cc_{q2}'$ should

apply. But then we are faced with the symmetric anomaly that c'_{q2} may be a right context for some rule $C_r \rightarrow C_{r1}C$ $(1 \leq r \leq n)$. An algorithm for determining such a mixed inspection order for left-context only and right-context only entries must further decide where to "break the chain" (by specifying a full context entry) in circular cases, the simplest of which is a state arising from the following rules:

$C_p \rightarrow C_{p1}C$ one of whose contexts is (C_{p1}, c'_{q2}).
$C_q' \rightarrow Cc'_{q2}$ one of whose contexts is (C_{r1}, c'_{q2}).
$C_r \rightarrow C_{r1}C$ one of whose contexts is (C_{r1}, c'_{s2}).
$C_s' \rightarrow Cc'_{s2}$ one of whose contexts is (C_{p1}, c'_{s2}).

Besides the fact that such an algorithm is too complex to be of practical applicability, the inspection of the resulting "mixed left-context only and right-context only" subtable requires more testing and masking than does inspection of a "pure left- (or right-) context only" subtable. For these reasons, we have chosen to remove such anomalies by immediately specifying a full context entry and by producing a pure left-context only subtable which must be inspected before a pure right-context only subtable.

With these examples in mind, and ignoring for the present the problem of one-symbol lookahead, we can develop an algorithm that generates a compressed M-table by breaking up the M-table for each state into as many as four separate subtables that must be inspected in sequence, namely:

1. A full-context (FC) subtable followed by
2. A left-context (LC) with unspecified right-context subtable followed by
3. A right-context (RC) with unspecified left-context subtable followed by
4. A no-context (NC) table whose one entry specifies a transition in which no contexts need be inspected.

After describing this compression technique, we can explain why it is suboptimal and what further refinements of the technique might be attempted.

Basically, the compression algorithm works as follows: The M-table entries for a given state $A \neq S_0$ are partitioned into three disjoint subsets of transitions,

$$M_{1A} = \{M(A_{i1}, A, a_{i1}) = \xi_i : A_i \rightarrow A_{i1}A \in P\},$$
$$M_{2A} = \{M(A_{j2}, A, a_{j2}) = \xi_j : A_j \rightarrow Aa_{j2} \in P\},$$
$$M_{3A} = \{M(A_{k3}, A, a_{k3}) = \xi_k : A_k \rightarrow A \in P\}.$$

The algorithm then attempts to construct FC, LC, RC, and NC subtables for this state, denoted by

M_{FCA}, M_{LCA}, M_{RCA}, and M_{NCA}

respectively. This is done by drawing entries from M_{1A} until empty, then successively exhausting the entries in M_{2A} and M_{3A}, as shown in the State Compression Algorithm.

The tables M_{FCA}, M_{LCA}, M_{RCA}, and M_{NCA} that result in each state of a BCA from the application of the State Compression Algorithm are then the entries of the compressed M-table for the BCA. In the special case of the initial state S_0, only the M_{FCA} and M_{RCA} tables are needed, and the compression method sketched at the beginning of this section is used for compressing the entries of S_0. Normally, there are other states of the BCA for which one or more of the partial-context tables resulting from compression are empty.

We can now consider the problem of compression in some state A of a BCA for which one-symbol lookahead is necessary. Since in practice lookahead is required only for a very small percentage of M-table entries, it is dealt with by first marking those entries of M_{1A}, M_{2A}, and M_{3A} for which $M(K_r, A, a_s) = \xi_u$, $(u = 1, \ldots, n)$ & $(n \geq 2)$, and by creating a fifth table for state A, $M_{1SA} = \{M(*, A, a_s a_{s+1}) = \xi_u : (M(K_r, A, a_s) = \xi_u)$ & $(u \in \{(1, \ldots, n)$ & $(n \geq 2)\}$. With the lookahead conflicts marked for state A, the compression algorithm can then operate on the remaining M-table entries for that state. In the resulting compressed BCA, state A has up to five context tables that are inspected in the sequence $(M_{1SA}, M_{FCA}, M_{LCA}, M_{RCA}, M_{NCA})$.

The algorithm for calculating the one-symbol lookahead entries is given in Appendix I.

The State Compression Algorithm

```
for ∀ entries M(A_i1, A, a_i1) = ξ_i of M_1A
  do if M_LCA contains an entry of the form M(A_i1, A, *) = ξ_p
    then if ξ_p ≠ ξ_i
        then enter M(A_i1, A, a_i1) = ξ_i into M_FCA
    else enter M(A_i1, A, *) = ξ_i into M_LCA
  end
for ∀ entries M(A_j1, A, a_j1) = ξ_j of M_2A
  do if M_RCA contains an entry of the form M(*, A, a_j1) = ξ_q
    then if ξ_q ≠ ξ_j
        then enter M(A_j1, A, a_j1) = ξ_j into M_FCA
    else enter M(*, A, a_j1) = ξ_j into M_RCA
  end
for ∀ entries M(A_k1, A, a_k1) = ξ_k of M_3A
  do if M_LCA contains an entry of the form M(A_k1, A, *) = ξ_p
    then if ξ_p ≠ ξ_k
        then enter M(A_k1, A, a_k1) = ξ_k into M_FCA
    else if M_RCA contains an entry of the form M(*, A, a_k1) = ξ_q
        then if ξ_q ≠ ξ_k
            then enter M(A_k1, A, *) = ξ_k into M_LCA
        else if M_NCA contains an entry of the form M(*, A, *) = ξ_r
            then if ξ_r ≠ ξ_k
                then enter M(*, A, a_k1) = ξ_k into M_RCA
            else enter M(*, A, *) = ξ_k into M_NCA
end
halt
```

7. Results Obtained Using the Compression Algorithm

We might first ask just how well could the compression algorithm be expected to work, given the BCA parser model that we are using. For n normal-form rules input to the BCA construction phase of our PGS, it is easy to see that the most compressed parser can have no fewer than n entries in its tables. This theoretical minimum number of entries is actually reached by our

compression algorithm for the following special cases of normal-form grammars:

Case a. Finite automaton grammars having rules of the form $A \rightarrow a$ and $B \rightarrow Cd$ exclusively; or

Case b. Grammars having rules of the form $A \rightarrow a$, $B \rightarrow CD$, and $D \rightarrow Ef$, where the E's and D's are disjoint.

For more complex grammars, it is easy to show by example that additional context will usually be required to determine some transitions, and so the theoretical minimum of table size is not generally attainable. As can be seen in Table I, however, our compression algorithm allows us to approach this minimum for representative programming language grammars.

As shown in Table I, our minimum table size is better than that obtained by the Ichbiah and Morse algorithm [3, 4]. This occurs because the lower bound on table size reached by their algorithm is given by the number of normal-form rules for their input grammar (normal form in our sense, because their parsing tables consume one symbol at a time) *plus* the number of distinct terminals in rules of the form $B \rightarrow Cd$. Since our lower bound of table size for the same grammar is only the number of normal-form rules, our algorithm yields inherently smaller parsing tables.

8. Improvements Possible in the Compression Algorithm

As can be readily seen from an inspection of the compression algorithm, the size and contents of a compressed M-table are affected by the sequence in which entries are selected from tables M_{1A}, M_{2A}, and M_{3A} of each state A in the original BCA. Moreover, for each possible sequence of entry choices in the compression of some BCA state, a different compression may result from re-ordering the inspection of partial-context tables so that M_{RCA} precedes M_{LCA}. In fact, to find the smallest compressed M-table for some state A of a BCA involves looking for a minimum over all possible sequences for selecting entries in M_{1A}, M_{2A}, and M_{3A} and over the two possible orderings of partial-context tables, namely (M_{FCA}, M_{LCA}, M_{RCA}, M_{NCA}) and (M_{FCA}, M_{RCA}, M_{LCA}, M_{NCA}). Such an "optimal" compression scheme is unworkable for any but the smallest of M-tables, because the number of steps for compressing any state with $n = n_{1A} + n_{2A} + n_{3A}$ entries grows in size by $2n_{1A}!n_{2A}!n_{3A}!$

We are currently developing near-optimal, heuristic strategies for improving the PGS compression algorithm while avoiding the combinatoric problem outlined above.

Without further M-table compression, it is possible to economize on the number of computer words needed to represent an M-table by merging the full context table entries within each state. This merging is accomplished by implementing each full context entry as a pair of partial context tests, and then merging the second tests so that m full context (60-bit) entries having one context in common are coalesced into $m + 1$ partial context (30-bit) table entries. By optimally merging full-context entries on common contexts, we also reduce the sequential search time for each state of the compressed BCA.

Since our parser performs sequential tests in each of its states, speed of parsing depends critically on minimization of state size. Experience with our PGS shows that large programming language compilers tend to have many small states (two or three entries when compressed), together with a large initial state (around 40 or 50 entries). Furthermore, it is easy to demonstrate [8] that as many as 50 per cent of the steps in a leftmost parse involve a transfer out of the initial state of the parser. We thus extract the most speed from our parser by doing a one-step, indexed search of the initial state using the preprocessor-supplied symbol index and sequential searches of the remaining parser state subtables.

9. Implementation of PGS Compilers

The current version of the PGS supplies a standard, assembly-language driver program for inspecting M-table entries. This subroutine receives input program symbol indices in sequence from a standard, FORTRAN preprocessor and searches the current M-table state to determine stack actions, code generator calls, and transfers to the next state of the M-table.

When all entries of a state have been searched without finding a match (where M_{NC} is empty for that

Table I. Statistics for Three PGS Compilers

Grammar	Expression sequence (Appendix III)	XPL [12]	Algol 60 [12]
Number of BNF productions	16	119	162
Total of contexts generated	116	5019	7344
Theoretical minimum number of M-table entries	28	182	248
Number of entries in compressed M-table	37	217	361
Size of parser in 60-bit words[1]	92	207	292
Size of compiler driver in 60-bit words[2]	103	288	—
Size of Ichbiah-Morse parser in 60-bit words[3]	105	—	484

[1] Assuming no code generators, a 66-word driver program (excluding the stack) is used, together with an M-table having 30-bit packed entries in which 8 bits (for specifying the index of code generators) are wasted.

[2] Assuming code generator subroutines linked to the parser, a 66-word driver program (excluding the stack) is used with an M-table having 30-bit packed entries. Eight bits of each entry are used as an index into a table of code generator subroutine entry addresses, stored one address per 60-bit word. The added size of the compiler driver is due to this table of addresses.

[3] Based on a computer printout supplied to us by J.D. Ichbiah, we have calculated that the Ichbiah-Morse driver program occupies 70 60-bit words when machine coded for the CDC-6500. Thirty-bit packed entries specify its actions and 15-bit packed entries specify their indirect jump table used for parsing. Each 30-bit entry carries a vacant 7-bit field that can be used to specify the index of code generators.

state), a syntactic error has been found in the input program. For this case, the driver program calls an "error-message code generator," erases the current input symbol, and resets itself to the initial state. Thus, standard recovery from an input program error consists of a sequence of returns to S_0 until a segment of the input program is reached that can be parsed without reference to previous segments of the input program. Of course, syntactic errors block execution of the compiled program.

As currently implemented, a typical PGS-generated XPL compiler occupies a total of 14,165 words of core memory and compiles typical programs at a speed of 14,000 cards per minute on the CDC-6500 computer. This compares with a measured speed of 13,000 cpm for the CDC supported, nonoptimizing, RUN FORTRAN compiler and 16,000 cpm measured on RUM (the University of Washington version of RUN). The XPL parser and its stack occupy under 6 per cent of the space required for the entire compiler; 31 per cent of compilation time is currently spent in the parser, 15 per cent in the preprocessor, 47 per cent in the code generators, and the remaining time in doing input-output.

Acknowledgments. An earlier version of this PGS was programmed by D. Browning, T. Davis, and C. Rieger while they were students in the Formal Compiling Techniques course offered by the Purdue University Computer Sciences Department. Many improvements in the PGS algorithms are due to suggestions by Ronald L. Lancaster of Purdue University. The authors wish to acknowledge lively and stimulating discussions on the subject of the PGS with J.D. Ichbiah and F. DeRemer.

Appendix I. One-Symbol Lookahead Calculations

The PGS provides for one-symbol lookahead whenever full-context specification is insufficient to drive the BCA deterministically. When this necessity arises, the method used in Section 4 to calculate contexts is extended to construct all legal co-occurrences of four-tuples of symbols, thus calculating contexts which include a lookahead symbol.

Case 1. A rule of the form $P_{j+1} = K_r' \to K_r a_s$ (or $P_{j+1} = K_r' \to a_s$) is applied in the lookahead context $(K_{r-1}, a_s a_{s+1})$ if and only if there is some nonterminal Q for which either

$(Q \to K_{r-1}A \in P) \;\&\; (A \overset{*}{\Rightarrow} Bv) \;\&\; (B \to CD \in P)$
$\;\&\; (D \overset{*}{\Rightarrow} a_{s+1}u) \;\&\; (C \overset{*}{\Rightarrow} K_r')$ or

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} a_{s+1}v) \;\&\; (A \overset{*}{\Rightarrow} uC)$
$\;\&\; (C \to K_{r-1}D \in P) \;\&\; (D \overset{*}{\Rightarrow} K_r').$

Case 2. A rule of the form $P_{j+1} = K_{r-1}' \to K_{r-1}K_r$ is applied in the lookahead context $(K_{r-1}, a_s a_{s+1})$ if and only if there is some nonterminal Q for which either

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} a_{s+1}w) \;\&\; (A \overset{*}{\Rightarrow} uC)$
$\;\&\; (C \to DE \in P) \;\&\; (E \overset{*}{\Rightarrow} a_s) \;\&\; (D \overset{*}{\Rightarrow} vK_{r-1}')$ or

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} Cw) \;\&\; (C \to DE \in P)$
$\;\&\; (E \overset{*}{\Rightarrow} a_{s+1}v) \;\&\; (D \overset{*}{\Rightarrow} a_s) \;\&\; (A \overset{*}{\Rightarrow} uK_{r-1}')$

Case 3. A rule of the form $P_{j+1} = K_r' \to K_r$ is applied in the lookahead context $(K_{r-1}, a_s a_{s+1})$ if and only if there is some nonterminal Q for which either

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} a_{s+1}w) \;\&\; (A \overset{*}{\Rightarrow} uC)$
$\;\&\; (C \to DE \in P) \;\&\; (E \overset{*}{\Rightarrow} a_s) \;\&\; (D \overset{*}{\Rightarrow} vF)$
$\;\&\; (F \to K_{r-1}G \in P) \;\&\; (G \overset{*}{\Rightarrow} K_r')$ or

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} Cw) \;\&\; (C \to DE \in P)$
$\;\&\; (E \overset{*}{\Rightarrow} a_{s+1}v) \;\&\; (D \overset{*}{\Rightarrow} a_s) \;\&\; (A \overset{*}{\Rightarrow} uF)$
$\;\&\; (F \to K_{r-1}G \in P) \;\&\; (G \overset{*}{\Rightarrow} K_r')$ or

$(Q \to K_{r-1}A \in P) \;\&\; (A \overset{*}{\Rightarrow} Bv) \;\&\; (B \to CD \in P)$
$\;\&\; (D \overset{*}{\Rightarrow} a_{s+1}u) \;\&\; (C \overset{*}{\Rightarrow} E) \;\&\; (E \to FG \in P)$
$\;\&\; (G \overset{*}{\Rightarrow} a_s) \;\&\; (F \overset{*}{\Rightarrow} K_r')$ or —

$(Q \to K_{r-1}A \in P) \;\&\; (A \overset{*}{\Rightarrow} Bw) \;\&\; (B \to CD \in P)$
$\;\&\; (D \overset{*}{\Rightarrow} Ev) \;\&\; (E \to FG \in P) \;\&\; (G \overset{*}{\Rightarrow} a_{s+1}u)$
$\;\&\; (F \overset{*}{\Rightarrow} a_s) \;\&\; (C \overset{*}{\Rightarrow} K_r')$ or

$(Q \to AB \in P) \;\&\; (B \overset{*}{\Rightarrow} a_{s+1}u) \;\&\; (A \overset{*}{\Rightarrow} vC)$
$\;\&\; (C \to K_{r-1}D \in P) \;\&\; (D \overset{*}{\Rightarrow} E) \;\&\; (E \to FG \in P)$
$\;\&\; (G \overset{*}{\Rightarrow} a_s) \;\&\; (F \overset{*}{\Rightarrow} K_r')$

Appendix II. Prevention of State-Stack Conflicts

The BCA construction algorithm generates a nonunique acceptor for a normal-form grammar having some symbol J that is both a state and a stack symbol. When this conflict arises, we transform the normal-form grammar as follows:

1. Create a new symbol J' and include J' in $V' - T'$.
2. Wherever nonterminal J appears as a stack symbol in the right part of a rule $A \to JB$, replace that rule by the rule $A \to J'B$.
3. Add the rule $J' \to J$ to P' and the production $(J' \to J, e)$ to R.

This transformation makes J' a stack symbol and J a state symbol in the resulting BCA, and leaves unchanged the language generated by the resulting normal-form grammar.

Appendix III. A Simple Expression Sequence Grammar Input to the PGS

```
EXPRG < > NAME NUMBER .
PROGRAM = $< ASSIGNMENT-LIST $> .
ASSIGNMENT-LIST = ASSIGNMENT ,
     ASSIGNMENT-LIST ASSIGNMENT .
ASSIGNMENT = NAME /L2A1 $= EXPRESSION /L2A2
EXPRESSION = TERM ,
     ADD-OP TERM /L3B1 ,
     EXPRESSION ADD-OP TERM /L3C1
TERM = FACTOR ,
     TERM MULT-OP FACTOR /L4B1
FACTOR = NAME /L5A1 ,
     NUMBER /L5B1 ,
     $( EXPRESSION $)
ADD-OP = $+ /L6A1 ,
     $- /L6B1 .
MULT-OP = $* /L7A1
     $/ /L7B1 .
```

```
***STATISTICS***
        9 TYPE 1 RULES ( K → N S ) WITH    29 CONTEXTS
        3 TYPE 2 RULES ( K → S T ) WITH     8 CONTEXTS
        7 TYPE 3 RULES ( K → S   ) WITH    53 CONTEXTS
        9 TYPE 4 RULES ( K → T   ) WITH    26 CONTEXTS
     ----                                  -----
       28 TOTAL NORMAL FORM RULES WITH     116 TOTAL CONTEXTS

        0 FULL-CONTEXT-WITH-LOOK-AHEAD M-TABLE ENTRIES
              0 ENTRIES RESULTED FROM TYPE 4 RULES
       11 FULL-CONTEXT M-TABLE ENTRIES
              2 ENTRIES RESULTED FROM TYPE 4 RULES
        9 RIGHT-CONTEXT-ONLY M-TABLE ENTRIES
              0 ENTRIES RESULTED FROM TYPE 4 RULES
       13 LEFT-CONTEXT-ONLY M-TABLE ENTRIES
        4 UNSPECIFIED-CONTEXT M-TABLE ENTRIES
     ----
       37 TOTAL M-TABLE ENTRIES
       69 PER-CENT M-TABLE COMPRESSION, EXCLUSIVE OF LOOK-AHEAD
       69 PER-CENT TOTAL COMPRESSION

        0 APPARENT AMBIGUITIES DETECTED
```

Received January 1972; revised January 1973

References

1. Floyd, R.W. A descriptive language for symbol manipulation *J. ACM 8*, 4 (Oct. 1961), 579–584.
2. Horning, J.J., and Lalonde, W.R. Empirical comparison of LR(k) and precedence parsers. ACM SIGPLAN Notices *5*, 11 (Nov. 1970), 10–24.
3. Ichbiah, J.D., and Morse, S.P. A technique for generating almost optimal Floyd-Evans Productions for precedence grammars. *Comm. ACM 13*, 8 (Aug. 1970), 501–508.
4. Ichbiah, J.D. Computer printout of an ALGOL 60 parser run through the Ichbiah-Morse system, 1971.
5. McKeeman, W.M., Horning, J.J. and Wortman, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
6. Mickunas, M.D. User's manual for the PUCSD translator-writing system. Comput. Sci. Dep., Purdue U., Lafayette, Ind., 1971.
7. Schneider, V. Pushdown-store processors of context-free languages. Ph.D. diss., Northwestern U., Evanston, Ill., 1966.
8. Schneider, V. A system for designing fast programming language translators. Proc. AFIPS 1969 SJCC, AFIPS Press, Montvale, N.J.; pp. 777–792.
9. Schneider, V. Some syntactic methods for specifying extendible programming languages. Proc. AFIPS 1969 SJCC, AFIPS Press, Montvale, N.J.; pp. 145–156.
10. Schneider, V. A translation grammar for ALGOL 68. Proc. AFIPS 1970 SJCC, AFIPS Press, Montvale, N.J., pp. 493–505.
11. Wirth, N., and Weber, H. A generalization of ALGOL and its formal definition: Parts I and II. *Comm ACM 9* (1966), 13–25; 89–99.
12. Mickunas, M.D. and Schneider, V. B. Translation grammars for XPL and ALGOL 60. Tech. Rep., Comput. Sci. Dep., Purdue U., Lafayette, Ind., 1972.

Corrigendum

In "Adaptive Correction of Program Statements" by E.B. James and D.P. Partridge, *Comm. ACM 16*, 1 (Jan. 1973), 27–37, the following correction should be made. On page 31, first column, line 10 from the bottom, the word CONTINUE is incorrect. The correct word is COTINUE.

Graphics and Image Processing

W. Newman Editor

A Scan Conversion Algorithm with Reduced Storage Requirements

B.W. Jordan Jr.
Northwestern University
and
R.C. Barrett
Hughes Aircraft Co.

Most graphics systems using a raster scan output device (CRT or hardcopy) maintain a display file in the XY or random scan format. Scan converters, hardware or software, must be provided to translate the picture description from the XY format to the raster format. Published scan conversion algorithms which are fast will reserve a buffer area large enough to accommodate the entire screen. On the other hand, those which use a small buffer area are slow because they require multiple passes through the XY display file. The scan conversion algorithm described here uses a linked list data structure to process the lines of the drawing in strips corresponding to groups of scan lines. A relatively small primary memory buffer area is used to accumulate the binary image for a group of scan lines. When this portion of the drawing has been plotted, the buffer is reused for the next portion. Because of the list processing procedures used, only a single pass through the XY display file is required when generating the binary image and only a slight increase in execution time over the fully buffered core results. Results show that storage requirements can be reduced by more than 80 percent while causing less than a 10 percent increase in execution time.

Key Words and Phrases: graphics, scan conversion, raster plotter, line drawing, discrete image, dot generation

CR Categories: 4.41, 6.35, 8.2

This work supported in part by the Office of Naval Research under Contract NO 014-67-A-0356 Mod. AE. Authors' addresses: B. W. Jordan, Departments of Computer Sciences and Electrical Engineering, Northwestern University, Evanston, IL 60201; R. C. Barrett, Computer Applications Department, Hughes Aircraft Co., Culver City, CA 90230.