A system for designing fast programming language translators

by VICTOR SCHNEIDER

University of Maryland
College Park, Maryland

INTRODUCTION

This paper demonstrates a straightforward algorithm for converting programming language grammars into pushdown-store automata translators. The language grammar is written as a "translation grammar" in which, for each syntactic rule, there is a corresponding "rule of translation" that recursively specifies the reverse Polish string translation of the objects in the syntactic rule. This augmented grammar is transformed directly into a flow chart for the appropriate translator.

To prevent the necessity of backtracking during translation, an algorithm is presented for converting nondeterministic translators into deterministic (i.e., single-scan) translators. Those languages for which this backtracking elimination algorithm fails contain as a subset the ambiguous programming languages.

From considerations of machine topology, upper bounds on memory storage requirements and computational times are derived for this class of translators. The upper bound on memory storage is shown to be proportional to the length of the input program. The upper bound on translation time required by these machines is also shown to be proportional to the length of the input program.

A detailed example, drawn from the ALGOL grammar, illustrates an actual application of the ideas in this paper.

Notation and basic definitions

Let V be a finite set of symbols, which we will call the vocabulary. Elements of V are denoted by letters, such as d, e, f, G, H, I, etc. Finite sequences of symbols, including the empty sequence e, are called strings and are denoted by late small letters, such as x, y, z, etc. The set of all strings over a set such as V is denoted by V*.

A context-free grammar (abbreviated CFG) is an ordered four-tuple

\[ G = (V, T, P, S) \]

where

(a) V is a vocabulary of symbols.
(b) T is a proper subset of V called the terminals.
(c) P is a finite, nonempty set of syntactic rules \( P_i \) of the form \( U \rightarrow x \), where \( U \neq x \), \( U \) is in \( V - T \), and \( x \) is in \( V^* - \{e\} \). For a rule \( P_i = U \rightarrow x \), \( U \) is called the left part and \( x \) the right part of \( P_i \).
(d) S is a special symbol in \( V - T \), the initial symbol.

As is usual, we say that \( x \) directly produces \( y \) \( (x = > y) \), and conversely \( y \) directly reduces to \( x \) if and only if there exist strings \( u, v \), such that \( x = uZv \) and \( y = u\overline{w}v \) and \( Z \overrightarrow{w} \) is in \( P \).

\( x \) produces \( y \) \( (x \overset{*}{=} > y) \), and conversely \( y \) reduces to \( x \) if and only if either

\[ x = y \]

or there exists a sequence of nonempty strings \( (w_0, w_1, \ldots, w_n) \) such that \( x = w_0 \) and \( y = w_n \) and

\[ w_i = > w_{i+1} \quad (i = 0, 1, \ldots, n - 1 \text{ and } n \geq 1) \].

\( x \) is a sentence of \( G \) if \( x \) is in \( T^* - \{e\} \) and \( S \) produces \( x \).

A context-free language (abbreviated CFL) is then the set of terminal strings that can be produced by grammar \( G \) from its initial symbol \( S \):

\[ L(G) = \{x: (S \overset{*}{=} > x) \text{ and } (x \in T^* - \{e\})\} \]
Let $S$ produce $x$. A *parse* of the string $x$ into the symbol $S$ is a sequence of rules $P_1, \ldots, P_n$ such that $P_j$ directly reduces $w_{j-1}$ into $w_j$ ($j = 1, \ldots, n$) and $x = w_0 \quad S = w_n$.

Let $x = a_1 \ldots a_n$ be a string of symbols $a_i$ in $T$. Then, in some reduction sequence in which $x = w_0$, let $x$ reduce to $w_j = u_{a_k} \ldots a_r$ with $u \in V^*$ and $1 \leq k \leq r$. If $P_i$ directly reduces string $w_j$ into $w_{j+1}$ and $P_{j+1}$ directly reduces $w_{j+1}$ into $w_{j+2}$, then $(P_i, P_{j+1})$ is called a *leftmost reduction sequence* if

$w_{j+1} = u'a_{i'} \ldots a_r \quad u' \in V^* x (V - T)$

and

$w_{j+2} = u'a_{i''} \ldots a_r \quad u'' \in V^* x (V - T)$

and

$k \leq k' \leq k'' \leq r$.

or

$w_{j+1} = u'w$ and $w_{j+2} = u'A$

and

$A \rightarrow wE$.

A parse $(P_1, \ldots, P_n)$ is called a *leftmost parse* if and only if the sequences $(P_i, P_{i+1})$ are leftmost reduction sequences for $i = 1, \ldots, n - 1$.

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. We define an *unambiguous* grammar $G$ to be one in which every $x$ in $L(G)$ has exactly one leftmost parse.

We next define a normal form for CFG's, in terms of which a leftmost parsing algorithm can be designed. The correspondence between this leftmost parsing algorithm and a pushdown automaton model to be introduced will then become apparent. Subsequently, an algorithm for facilitating single-scan leftmost parsing in a large class of grammars will be developed.

**Normal form grammars**

A grammar $G = (V, T, P, S)$ will be said to be in *normal form* if all the rules in $P$ are of the forms

$A_i \rightarrow A_{i1}A_{i2}$ or $A_j \rightarrow A_{j1}$

or

$A_k \rightarrow A_{k1}A_{k2}$ or $A_m \rightarrow a_{m1}$

with $A_{i0}, A_{i1}, A_{j1}, A_{k1}$ in $V - T$ and $a_{k2}, a_{m1}$ in $T$. A very simple algorithm exists for converting any grammar $H$ into a grammar $H'$ in normal form such that $L(H) = L(H')$. 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:

All productions in $P$ of $H$ that are already in normal form are taken into $P'$ of $H'$. The remaining productions in $P$ are of the form

$X \rightarrow X_1 \ldots X_n \quad (n > 2) \& (X_i \in V)$.

Each production of this form is transferred to $P'$ as a sequence of productions.

$J_v \rightarrow J_{v-1}X_{v+1}$ for $v = 1, \ldots, n - 1$

where $J_{n-1}$ is $X$. $J_0$ is $X_1$ if $X_1$ is in $V - T$ of $H$; otherwise, an additional rule of the form

$J_0 \rightarrow X_1$

is included in $P'$. The $J_v$ are treated as new elements in $V' - T'$ of $H'$, and the $J_v$ are distinct from the elements in $V - T$ of $H$.

The fact that the $J_v$ of the algorithm are “new and distinct” leads to a simple proof of the one-to-one correspondence between derivations of sentences in $L(H)$ and $L(H')$: Since each rule of $P$ corresponds to a particular rule or sequence of rules in $P'$, it follows that, for each derivation possible in $H$, there is a corresponding derivation in $H'$, and conversely. Because of this unique correspondence, it also follows that ambiguity in $L(H)$ is equivalent to ambiguity in $L(H')$.

**Leftmost parses and normal-form grammars**

In order to describe the algorithm for producing leftmost parses of the sentences of a grammar $G$ in normal form, 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 $G$:

$P'_1 = S' \rightarrow J_1 \#$

$P'_2 = J_1 \rightarrow J_2S$  \quad $P'_3 = J_2 \rightarrow \#$

This has the effect of putting boundary markers at both ends of all strings produced by the grammar.

Let $w_0 = \# a_1 \ldots a_n \#$ 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

$$w_1 = J_2 a_1 \ldots a_n$$

After $j$ steps, $w_j$ has been reduced to

$$w_j = J_{j+1} K_1 \ldots K_r a_{j+1} \ldots a_n$$

In this configuration, $K_1, \ldots, K_r$ are all symbols of $V - T$ in the grammar. If $w_j$ is in $L(G)$, the leftmost sequence of rules $P_1 = P_2, \ldots, P_j$ are precisely the first $j$ reductions of the leftmost parse of $w_j$ to $S'$. Capital letters are assumed to be members of $V - T$ of the grammars in the remaining discussion.

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

1. $S'$ does not produce $w_j$, where $w_j = J_2 K_1 \ldots K_r a_{j+1} \ldots a_n$.

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

1. A rule of the form $P_{j+1} = K_{j+1} \rightarrow a_j$ reduces $w_j$ to $w_{j+1}$.
2. A rule of the form $P_{j+1} = K'_r \rightarrow K_r$ reduces $w_j$ to $w_{j+1}$.
3. A rule of the form $P_{j+1} = K'_r \rightarrow K_{j+1} K_r$ reduces $w_j$ to $w_{j+1}$.
4. A rule of the form $P_{j+1} = K'_r \rightarrow K_r a_j$ reduces $w_j$ to $w_{j+1}$.

That only these cases need be considered is proved in [13]. Note that each of the applications of rules $P_{j+1}$ in cases (1)–(4) leaves the length of string $w_{j+1}$ either the same as that of $w_j$ or reduced in length by one symbol. This fact is used in a later proof of the upper bound on computation time required by the leftmost parsing algorithm.

In general, the decision concerning which of the cases (1) to (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 $K_{j+1} K_r$ and $K_r a_j$ on the right part. To decide which case applies at a given step of the parse then requires algorithms for discovering what symbols can legally be adjacent to the symbols being reduced in that step of the reduction while $w_j$ is a sentence of $G$. The algorithms to be given in what follows are similar to those of Floyd and Wirth and Weber in that they construct all legal cooccurrences of triples of symbols in some language's grammar.
the \((j + 1)\) - th reduction, there must be one or more
symbols \(Z\) in \(V - T\) such that
\[ Z \rightarrow K_{r-1}Y \text{ is in } P \]
and \[ Y \preceq > K'_u \text{ with } u \text{ in } V^* \].

The pairs \((K_{r-1}, a_u)\) are the contexts in which rule \(K'_r \rightarrow K_a\) applies.

After the contexts for which cases (1)–(4) apply have been determined, there may in general still exist rules having the same contexts. The existence of such rules in a grammar may imply the necessity of backtracking methods for use in parsing a given string of that grammar. Or, such a grammar may be ambiguous. In the following section, we sketch a formal model for this normal-form leftmost parsing algorithm. In terms of this model, we can present an algorithm for eliminating the necessity of backtracking in a large class of unambiguous CFL's.

**Pushdown automaton parsing model**

In this section, we present an automaton model of our leftmost parsing algorithm. This model will be called a **bounded-context acceptor** (BCA), and can be described intuitively as follows: It has a finite number of states, a finite input vocabulary, and an auxiliary memory stack mechanism (called a pushdown store). In addition, it has an initial (or starting) state, a final (or accepting) state, and a boundary symbol \(\#\) whose function in the scheme is analogous to the control cards used before and after a computer program. In general, the BCA is in some state, and is scanning the topmost pushdown-store symbol \(K\) and an input-string symbol \(a\). In a possibly nondeterministic manner, the BCA goes to another state, either by erasing \(K\) or by erasing \(a\) in the process, and possibly storing a new symbol on top of the pushdown-store. All these notions are restated more precisely in what follows.

A **bounded-context acceptor** \(P\) is defined to be a seven-tuple:

\[ P = (Q, T, N, M, \#, S_0, F) \]

where

- \(Q\) is a finite set, called the **states** of the machine.
- \(T\) is a finite set of symbols, called the **input-tape vocabulary**.
- \(N\) is a finite set of symbols, called the **pushdown-store vocabulary**.
- \(M\) is a mapping of \(N \times Q \times T\) into the finite subsets of \(N \times \{\#\} \times (T \cup \{\#\}) \times (N \cup \{\#\})\).
- \(S_0\) is the initial state of the computation, and \(F\) is called the **final state**.
- \(\#\) is a special symbol such that \(T \cap N = \{\#\}\).

Let \(A = (Q, T, N, M, \#, S_0, F)\) be a BCA. A **configuration** of \(A\) is an element of
\[ \{\#\} \times (N \cup \{\#\})^\ast \times Q \times (T \cup \{\#\})^\ast \times \{\#\} . \]

the configuration \((\#, S_1, y, \#)\) denotes the fact that the acceptor \(A\) is in state \(S_1\), with string \(\#\) on the pushdown store and string \(y\) remaining to be read on the input tape.

By analogy to our notation for CFL's, we can define an **initial configuration** of a computation to be
\[ C_0 = (\#, S_0, x, \#) , \]

where \(x \in T^* \cup \{\#\}\) is the input string to be accepted. The final configuration is \((\#, F, \#)\). The computation performed by a BCA is essentially a reduction sequence that reduces \(C_0\) to the final configuration.

Let \(C_j\) and \(C_{j+1}\) be two configurations of a computation. Then, \(C_j\) **directly reduces to** \(C_{j+1}\), or \(C_j \rightarrow C_{j+1}\), if
\[ C_j = (t Z S_1 a w) \text{, } C_{j+1} = (t y S_1 b w) \]

and where
\[ (y, S_1, b) \in M(Z, S_1, a) \text{ and } \]

and
\[ [(t = e) \& (Z = \#) \lor (t \in \{\#\} \times (N - \{\#\})^\ast \& (Z \in (N - \{\#\}))]) \]
\[ \& [(w = e) \& (a = \#) \lor (w \in (T - \{\#\})^\ast \times \{\#\} \& (a \in (T - \{\#\}))]) \]
\[ \& \sim ([Z = \#] \& (a = \#)] \]

and
\[ (b \in \{\#\} \cup \{e\}) \& [(y \in (Z \cup \{e\}) \times (N \cup \{e\}) \& \sim (Z \neq \#) \lor (y \in \{\#\} \times (N \cup \{e\}) \& (Z = \#))]. \]

We next define the sequence of configurations that leads to a complete computation of the acceptor.
Let
\[ C_1 = (\# u S b_1 \ldots a v \#), u \in N^* \text{ and } v \in T^* \]
and
\[ C_2 = (\# u' S b_2 \#), u' \in N^* , \]
be configurations of the computation. Then \( C_1 \) reduces to \( C_2 \), or \( C_1 \) \( \rightarrow \) \( C_2 \), if there exists a sequence of configurations \((H_0, H_1, \ldots, H_j)\), with \( C_1 = H_0 \) and \( C_2 = H_j \) and
\[ H_{i+1} \leftarrow H_i \quad (i = 1, \ldots, j) . \]

Then, the language accepted by a BCA \( P \) is the set of input strings given by
\[ L(P) = \{ x : [(\# S_0 x \#) \vdash (\# F \#)] \& [x \in T^* - \{ e \}] \} . \]

In the more standard pushdown-automaton acceptor model, a mapping from \((N - \{ \# \}) \times Q \times (T \cup \{ e \} - \{ \# \})\) into the finite subsets of \((N - \{ \# \}) \times Q \times (T \cup \{ e \} - \{ \# \})\) implies the existence of transitions between states in which the current input-tape symbol is ignored (i.e., the empty symbol \( e \) is erased from the input tape). When an input-tape symbol is used for a transition in this model, it is always erased during the transition. Thus, while a standard pushdown automaton having the same language can always be constructed from a BCA, the notion of context in determining a transition is lost, and the resulting acceptor will generally have fewer uniquely defined transitions than the original BCA. Moreover, given a standard pushdown automaton, a grammar for its language can always be found, and, from this grammar, a BCA can be constructed to accept the same language, as will be seen. Hence, BCA's and standard pushdown automata are equivalent in computational power, but a BCA with uniquely defined transitions does not always correspond to a pushdown automaton with uniquely defined transitions.

Automaton realization of leftmost parses

With the BCA model defined above, it is possible to introduce a correspondence between rules of a normal-form grammar and the states and symbols of a BCA. In this correspondence, the initial symbol \( S \) of a grammar becomes the initial state \( F \) of the BCA. Furthermore, the BCA constructed from a normal-form grammar is a slightly restricted version of the BCA model. This is because transitions from initial state \( S_0 \) of a constructed BCA can only occur together with the erasure of an input-tape symbol. Since the full BCA model only allows the pushdown store to be increased in size during a transition into state \( S_0 \), this additional restriction means that the pushdown store of a constructed BCA can increase in size by at most one symbol for each input-tape symbol read during a computation.

The BCA is constructed from a normal-form grammar as follows: For all rules in the grammar of the forms
\[ A_i \rightarrow A_\alpha A_\beta \quad \text{and} \quad A_j \rightarrow A_\alpha a_\beta \]
(where the capital letters are nonterminals and the small letters are terminal symbols), the \( A_\alpha \)'s and \( A_\beta \)'s become states of the BCA. The \( A_\alpha \)'s become members of \( N \), the stack vocabulary, and the \( a_\beta \)'s become members of \( T \), the input-string vocabulary. What follows is the algorithm for constructing a BCA that accepts the language of a normal-form grammar. The algorithm is in four sections corresponding to the four rule types allowed in normal-form grammars.

I. Rule \( A_i \rightarrow A_\alpha A_\beta \) with contexts \((A_i, a_\alpha)\):
- If \( A_i \in N \), then \((A_i, S_0, a_\alpha) \in M(A_i, A_\alpha, a_\alpha)\).
- If \( A_i \in Q \), then \((e, A_i, a_\alpha) \in M(A_i, A_\alpha, a_\alpha)\).

These transitions take care of all possibilities arising from case (3) of the leftmost parsing algorithm. If \( A_i \) is in \( N \), that means that a pair of nonterminals \( A_\alpha A_\beta \) appears on the right part of some rule of the grammar. Hence, \( A_i \) is stored on top of the stack, and the automaton transfers to its initial state, from which it proceeds to discover \( A_\beta \). If \( A_i \) is in \( Q \), then \( A_i \) is either the second nonterminal of some rule of the grammar or is the first nonterminal in a rule of the form \( A_i \rightarrow A_\alpha a_\beta \). In either case, the stack does not increase in length during the transition.

II. Rule \( A_k \rightarrow A_\alpha \beta \) with contexts \((K_{r-1}, a_\beta)\):
- If \( A_k \in N \), then \((K_{r-1}, S_0, e) \in M(K_{r-1}, A_\alpha, a_\beta)\).
- If \( A_k \in Q \), then \((K_{r-1}, A_i, e) \in M(K_{r-1}, A_\alpha, a_\beta)\).

These transitions take care of all possibilities arising from case (4) of the leftmost parsing algorithm.

III. Rule \( A_j \rightarrow a_\alpha \) with contexts \((K_r, a_\alpha)\):
- If \( A_j \in N \), then \((K_r, A_i, S_0, e) \in M(K_r, S_0, a_\alpha)\).
- If \( A_j \in Q \), then \((K_r, A_i, e) \in M(K_r, S_0, a_\alpha)\).

These transitions take care of all possibilities arising from case (1) of the leftmost parsing algorithm.

From the collection of the Computer History Museum (www.computerhistory.org)
IV. Rule $A_j \rightarrow A_\beta$ with contexts $(K_{r-1}, a_r)$:

For every chain of rules in the grammar of the form

$$
P_1 = A \rightarrow A^{(1)}
$$
$$
P_2 = A^{(0)} \rightarrow A^{(2)}, \ldots,
$$
$$
P_n = A^{(n-1)} \rightarrow A^{(n)} \quad \text{with } n \geq 2,
$$
and such that there is at least one context $(K_{r-1}, a_r)$ common to rules $(P_1, \ldots, P_n)$, we introduce automaton transitions of the form

$$(K_{r-1}, A^{(n-1)}, a_r) \in M(K_{r-1}, A^{(n)}, a_r).$$

These $A^{(0)}, \ldots, A^{(n)}$ are thus treated as states of the BCA.

For all contexts $(K_{r-1}, a_r)$ associated with individual rules $A_j \rightarrow A_\beta$, we have the following:

If $A_j \in N$, then $(K_{r-1}, A_j, a_r) \in M(K_{r-1}, A_\beta, a_r)$.

If $A_j \in Q$, then $(K_{r-1}, A_\beta, a_r) \in M(K_{r-1}, A_\beta, a_r)$.

These transitions take care of all possibilities arising from case (2) of the leftmost parsing algorithm.

When all transitions of a machine have been defined as described above, the language accepted by that machine is the language of the grammar from which it is constructed.$^{14}$

A simple programming language translator

The following is a simplified grammar for a computer programming language having nested block structure, conditional statements, and arithmetic assignment statements. The ALGOL conventions are used for representing symbols of the grammar; i.e., members of $V - T$ are enclosed by the metasyntactic brackets "(" and ")", and members of $T$ are not. The symbol "$|" is a separator that allows two or more rules having the same left part to be written together.

Table I—A grammar for a simple programming language

$$G: \langle \text{program} \rangle \rightarrow \langle \text{body} \rangle \langle \text{stat} \rangle \text{ end}$$

$$\langle \text{body} \rangle \rightarrow \langle \text{body} \rangle \langle \text{stat} \rangle;$$

$$\langle \text{stat} \rangle \rightarrow \langle \text{program} \rangle | \langle \text{assignment} \rangle$$

$$\langle \text{assignment} \rangle \rightarrow \langle \text{var} \rangle := \langle \text{expr} \rangle$$

$$\langle \text{expr} \rangle \rightarrow \langle \text{simple expr} \rangle | \langle \text{if clause} \rangle$$

$$\langle \text{if clause} \rangle \rightarrow \langle \text{relation} \rangle \text{ then}$$

$$\langle \text{relation} \rangle \rightarrow \langle \text{simple expr} \rangle = \langle \text{simple expr} \rangle$$

$$\langle \text{var} \rangle \rightarrow A|B|C|\ldots|Z$$

$$\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle | \langle \text{number} \rangle \langle \text{digit} \rangle$$

$$\langle \text{digit} \rangle \rightarrow 0|1|\ldots|9$$

The programming language $G$ easily reduces to the following normal-form grammar $G'$ augmented by the addition of endmarkers:

Table II—The normal-form version of grammar $G$

$G': S \rightarrow Y_1 S$

$$Y_1 \rightarrow Y_2 \langle \text{program} \rangle$$

$$Y_2 \rightarrow$$

Case (2) of the leftmost parsing algorithm.

What follows is a table of contexts in which the rules of $G'$ can be applied during a leftmost parse of some string in $L(G')$:
Table III—Contexts associated with rules in Table II

<table>
<thead>
<tr>
<th>Rule</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow Y_1 #$</td>
<td>$(e, #)$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow Y_2 \langle \text{program} \rangle$</td>
<td>$(Y_2, #)$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow #$</td>
<td>$(e, #)$, $(\langle \text{program} \rangle, #)$</td>
</tr>
<tr>
<td>$\langle \text{program} \rangle \rightarrow X_1 \text{ end}$</td>
<td>$(Y_2, \text{ end})$, $(\langle \text{body} \rangle, \text{ end})$</td>
</tr>
<tr>
<td>$X_1 \rightarrow \langle \text{body} \rangle \langle \text{stat} \rangle$</td>
<td>$(\langle \text{body} \rangle, \text{ end})$</td>
</tr>
<tr>
<td>$\langle \text{body} \rangle \rightarrow \text{begin}$</td>
<td>$(Y_2, \text{ begin})$, $(\langle \text{body} \rangle, \text{ begin})$</td>
</tr>
<tr>
<td>$\langle \text{body} \rangle \rightarrow X_4;$</td>
<td>$(Y_2, ;)$, $(\langle \text{body} \rangle, ;)$</td>
</tr>
<tr>
<td>$X_2 \rightarrow \langle \text{body} \rangle \langle \text{stat} \rangle$</td>
<td>$(\langle \text{body} \rangle, ;)$</td>
</tr>
<tr>
<td>$\langle \text{stat} \rangle \rightarrow \langle \text{program} \rangle$</td>
<td>$(\langle \text{body} \rangle, \text{ end})$, $(\langle \text{body} \rangle, ;)$</td>
</tr>
<tr>
<td>$\langle \text{stat} \rangle \rightarrow \langle \text{assignment} \rangle$</td>
<td>$(\langle \text{body} \rangle, \text{ end})$, $(\langle \text{body} \rangle, ;)$</td>
</tr>
<tr>
<td>$\langle \text{assignment} \rangle \rightarrow X_3 \langle \text{expr} \rangle$</td>
<td>$(X_2, \text{ end})$, $(X_2, ;)$</td>
</tr>
<tr>
<td>$X_3 \rightarrow \langle \text{var} \rangle :=$</td>
<td>$(\langle \text{body} \rangle, :=)$</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle \rightarrow \langle \text{simple expr} \rangle$</td>
<td>$(X_5, ;)$, $(X_2, \text{ end})$, $(X_8, ;)$</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle \rightarrow X_4 \langle \text{expr} \rangle$</td>
<td>$(X_4, \text{ end})$, $(X_4, ;)$</td>
</tr>
<tr>
<td>$X_4 \rightarrow X_4 \text{ else}$</td>
<td>$(X_3, \text{ else})$, $(X_8, \text{ else})$</td>
</tr>
<tr>
<td>$X_5 \rightarrow \langle \text{if clause} \rangle \langle \text{simple expr} \rangle$</td>
<td>$(\langle \text{if clause} \rangle, \text{ else})$</td>
</tr>
<tr>
<td>$\langle \text{simple expr} \rangle \rightarrow \langle \text{term} \rangle$</td>
<td>$(\langle \text{if clause} \rangle, \text{ else}), (X_3, ;), (X_3, \text{ end}, (X_4)), (\langle \text{if clause} \rangle, ;), (X_3, +), (X_3, +), (X_9, +), (X_9, +), (X_{11}, =)$</td>
</tr>
<tr>
<td>$\langle \text{simple expr} \rangle \rightarrow X_6 \langle \text{term} \rangle$</td>
<td>$(X_4, +), (X_5, \text{ else}), (X_8, ;), (X_4, \text{ end}), (X_4, +), (X_8, \text{ then}), (X_8, ])$</td>
</tr>
<tr>
<td>$X_6 \rightarrow \langle \text{simple expr} \rangle +$</td>
<td>$(\langle \text{if clause} \rangle, +), (X_4, +), (X_3, +), (X_9, +), (X_{11}, +), (X_{11}, +)$</td>
</tr>
<tr>
<td>$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle$</td>
<td>$(X_6, +), (X_8, =), (X_8, \text{ then}), (X_6, \text{ else}) (X_6, #), (\langle \text{if clause} \rangle, #), (X_9, +), (X_9, +), (X_9, +), (X_{11}, +), (X_{11}, +)$</td>
</tr>
<tr>
<td>Rule</td>
<td>Contexts</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>⟨term⟩ → X₇ ⟨factor⟩</td>
<td>⟨X₇, +⟩, ⟨X₇, =⟩, ⟨X₇, then⟩, ⟨X₇, else⟩</td>
</tr>
<tr>
<td></td>
<td>⟨X₇⟩, ⟨X₇, ;⟩, ⟨X₇, end⟩</td>
</tr>
<tr>
<td>X₇ → (term)*</td>
<td>⟨X₅, *⟩, ⟨(if clause), *⟩, ⟨X₁₂, *⟩, ⟨X₁₁, *⟩</td>
</tr>
<tr>
<td></td>
<td>⟨X₄, *⟩, ⟨X₃, *⟩, ⟨X₉, *⟩</td>
</tr>
<tr>
<td>⟨factor⟩ → ⟨var⟩</td>
<td>⟨X₇, *⟩, ⟨X₇, +⟩, ⟨X₇, =⟩, ⟨X₇, then⟩</td>
</tr>
<tr>
<td>⟨factor⟩ → ⟨number⟩</td>
<td>⟨X₅, else⟩, ⟨X₅, +⟩, ⟨X₅, =⟩, ⟨X₅, then⟩</td>
</tr>
<tr>
<td></td>
<td>⟨X₆, else⟩, ⟨X₆, else⟩, ⟨X₆, *⟩, ⟨(if clause), *⟩, ⟨X₁₂, *⟩, ⟨X₉, *⟩, ⟨X₃, *⟩, ⟨(if clause), else⟩, ⟨X₃, ;⟩, ⟨X₃, end⟩, ⟨X₉, !⟩, ⟨(if clause), +⟩, ⟨X₆, +⟩, ⟨X₅, +⟩, ⟨S₁₁, =⟩</td>
</tr>
<tr>
<td>⟨factor⟩ → X₉ ]</td>
<td>⟨X₇⟩, ⟨X₆⟩, ⟨(if clause), ]⟩, ⟨X₁₁⟩</td>
</tr>
<tr>
<td></td>
<td>⟨X₁₁⟩, ⟨X₄⟩, ⟨X₃⟩, X₉</td>
</tr>
<tr>
<td>X₉ → X₉ ⟨expr⟩</td>
<td>⟨X₉⟩</td>
</tr>
<tr>
<td>X₉ → [</td>
<td>⟨X₉, ], ⟨X₅, ], ⟨X₉, ]⟩, ⟨(if clause), ]⟩</td>
</tr>
<tr>
<td></td>
<td>⟨X₁₁⟩, ⟨X₁₁, ], ⟨X₅, ], ⟨X₆, ]⟩</td>
</tr>
<tr>
<td>⟨if clause⟩ → X₁₀ then</td>
<td>⟨X₅, then⟩, ⟨X₄, then⟩, ⟨X₃, then⟩</td>
</tr>
<tr>
<td>X₁₀ → X₁₁ ⟨relation⟩</td>
<td>⟨X₁₁, then⟩</td>
</tr>
<tr>
<td>X₁₁ → if</td>
<td>⟨X₃, if⟩, ⟨X₄, if⟩, ⟨X₉, if⟩</td>
</tr>
<tr>
<td>⟨relation⟩ → X₁₂ ⟨simple expr⟩</td>
<td>⟨X₁₂, then⟩</td>
</tr>
<tr>
<td>X₁₂ → ⟨simple expr⟩ =</td>
<td>⟨X₁₁, =⟩</td>
</tr>
<tr>
<td>⟨var⟩ → A</td>
<td>⟨X₇, A⟩, ⟨(body), A⟩, ⟨X₆, A⟩, ⟨X₆, A⟩, ⟨X₅, A⟩, ⟨X₅, A⟩, ⟨X₁₂, A⟩, ⟨X₁₁, A⟩, ⟨(if clause), A⟩</td>
</tr>
<tr>
<td>⟨number⟩ → ⟨number⟩ ⟨digit⟩</td>
<td>⟨⟨number⟩, 1⟩, ..., ⟨(number), 9⟩</td>
</tr>
<tr>
<td></td>
<td>⟨⟨number⟩, *⟩, ⟨(number), +⟩</td>
</tr>
<tr>
<td></td>
<td>⟨⟨number⟩, =⟩, ⟨(number), then⟩</td>
</tr>
<tr>
<td></td>
<td>⟨(number), else⟩, ⟨(number), ;⟩</td>
</tr>
<tr>
<td></td>
<td>⟨(number), end⟩</td>
</tr>
<tr>
<td>⟨digit⟩ → 0</td>
<td>1</td>
</tr>
</tbody>
</table>
From Table III, a flow chart of the BCA that accepts L(G') can be constructed. This flow chart is abbreviated in that, for a given state, only those contexts necessary for determining a transition are presented. Thus, when no ambiguity will be introduced, only the stack symbol or the input-string symbol is used for determining which of several possible transitions can occur. In the flow chart, the array N, with index i, is used to represent the symbols of the stack, and the array S, with index j, represents the symbols of the input string. Note that “error exits,” i.e., the instances of case (0) in the leftmost parsing algorithm, are omitted from the flow chart. An error is assumed to exist at any transition in which the appropriate symbols are not present on either the input string or the stack.

The next step after synthesizing a BCA as in Figure 1 is to design a translator for the language accepted by that BCA. To do this, we employ a notation similar to that used in [13], and available in numerous versions in the current literature. The basic idea of the notation is to introduce rules of translation in a one-to-one correspondence with the rules of the original grammar. These rules of translation describe the effect of translating the right parts of the syntactic rules with which they are associated. As an example, we might have the following pairing in some grammar:

\[
\text{Syntactic Rule: } \quad \text{Rule of Translation:} \\
\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \ast \langle \text{factor} \rangle \quad \langle \text{term} \rangle \ast \langle \text{factor} \rangle \quad \text{multiply}
\]

This pairing of rules can be represented as a translation grammar G' = (G, 0, f), where G = (V, T, P, S) is the programming language syntax, 0 is a translated program vocabulary, and f is a one-to-one mapping from P into PxO*. The rule of translation given in the above example is easily recognized as one rule for converting from standard arithmetic notation to reverse Polish notation. In the translated sequence ‘\langle \text{term} \rangle \ast \langle \text{factor} \rangle \quad \text{multiply}', the translated objects corresponding to \langle \text{term} \rangle are written out in the sequence determined by the rules of translation associated with the syntactic rules derived from \langle \text{term} \rangle, and likewise with \langle \text{factor} \rangle. In general, if a rule of translation is identical to the right part of its associated syntax rule, we write the symbol ‘I’ in place of the rule of translation.

![Figure 1—The BCA acceptor of L(G')](image-url)
We can next present the simple programming language given above as a translation grammar:

Table IV—Translation grammar for a small language

<table>
<thead>
<tr>
<th>Syntactic Rules:</th>
<th>Rules of Translation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{G: } (\text{program}) \to (\text{body}) (\text{stat}) \text{ end} )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{body}) \to \text{begin} )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{body}) \to (\text{body}) (\text{stat}); )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{stat}) \to (\text{program}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{stat}) \to (\text{assignment}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{assignment}) \to (\text{var}) := (\text{expr}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{expr}) \to (\text{simple expr}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{expr}) \to (\text{if clause}) (\text{simple expr}) \text{ else } (\text{expr}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{simple expr}) \to (\text{term}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{term}) \to (\text{simple expr}) + (\text{term}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{term}) \to (\text{factor}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{factor}) \to (\text{var}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{factor}) \to (\text{number}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{factor}) \to [ (\text{expr}) ] )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{if clause}) \to \text{if } (\text{relation}) \text{ then } (\text{expr}) \text{ else } (\text{expr}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{relation}) \to (\text{simple expr}) = (\text{simple expr}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{simple expr})^{(1)} = (\text{simple expr})^{(2)} )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{var}) \to A )</td>
<td>I</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{var}) \to Z )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{number}) \to (\text{digit}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{number}) \to (\text{number}) (\text{digit}) )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{digit}) \to 0 )</td>
<td>I</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>I</td>
</tr>
<tr>
<td>( (\text{digit}) \to 9 )</td>
<td>I</td>
</tr>
</tbody>
</table>
The details of the translator grammar can be explained briefly: Essentially, arithmetic expressions and relations are translated into reverse-Polish strings through the rules of translation. Conditional expressions are rearranged so that if, then, and else become place-markers in the translated program, and the device that interprets the translated program contains routines for passing to the statements directly following if, then, or else as appropriate. Since the effect of interpreting the translated program is to coalesce assignment statements into a single resultant operand that is the "value" of the assigned expression, the semicolon ";" that separates program statements is written into the translated program so that the interpreting mechanism can erase the resultant operand of an assignment. begin and end are likewise written in sequence into the translated program so that the interpreter of this program can maintain a list of valid identifiers corresponding to the program's nested block structure.

The translator of Figure 2 is thus a relatively straightforward extension of the PDA in Figure 1, with the additional structure arising from the appropriate rules of translation. The sequencing of operators to follow pairs of operands is accomplished by noting that state transitions such as the one that recognizes the sequence

\[(\text{simple expr}) + (\text{term})\]

in Figure 1 are appropriate points for writing out operators (here, "add") into the translated program. Likewise, a rule of translation such as

\[(\text{number operand}) \ (\text{number})\]

requires some temporary storage in the translator to store the symbols that comprise (number), finally writing out the translated sequence

Code ('number operand')

Code (temporary store).

![Figure 2—The BCA translator of L(G')](image-url)
Deterministic and extended-deterministic acceptors

An acceptor A is called deterministic if M is a partial mapping from

\[ \text{N} \times (\text{Q} \times \text{T} \cup \{ \epsilon \}) \]

\[ \cup (\text{N} \cup \{ \epsilon \}) \times \text{Q} \times (\text{T} \cup \{ \epsilon \}) - \{ \epsilon \} \times \text{Q} \times \{ \epsilon \}. \]

This is equivalent to saying that, for every configuration \( C_i \) of A, there is at most one configuration \( C_k \) for which \( C_i \rightarrow C_k \). By an induction argument, if \( x \) is in \( L(A) \), there is only one sequence of configurations by which \( (\# \text{S}_x \#) \rightarrow^* (\# F \#) \). Thus, if A were constructed from some grammar G, there would exist exactly one leftmost parse for each \( x \) in \( L(G) \), and G would be unambiguous. However, it is fairly easy to construct languages that are unambiguous and for which no deterministic BCA can be constructed.13

Since not every unambiguous grammar leads to a deterministic BCA, it is of interest to consider methods for extending the BCA model to handle a larger class of unambiguous languages in some “almost deterministic” fashion. This is important because any non-deterministic automaton is inefficient to use, owing to the necessity of repeating its computations until the correct sequences of configurations are found. This necessity for backtracking during computations of such an automaton A with input string \( x \) in \( L(A) \) occurs when a configuration \( C_i \) is reached for which

\[ C_1 \rightarrow C_{12}, \ldots, C_1 \rightarrow C_n, \quad \text{and} \quad n \geq 2. \]

Since we assume that \( L(A) \) is unambiguous, there can exist only one \( C_{ij} \) above for which \( C_{ij} \rightarrow^* (\# F \#) \). The problem then becomes one of finding a general algorithm for processing each of the \( n \) possibilities above, together with their descendents, in some parallel fashion, perhaps similar to the methods used for “real-time” languages.4,11

The algorithm that we present here for implementing parallel computations of nondeterministic BCA’s involves a basic computational strategy: No matter how many alternative configurations exist at some step in a computation, all the configurations must have the same input-string symbol in common. Thus, simulation of a parallel computation having no more than \( n \) configurations active requires only one input string and \( n \) pushdown stores. Moreover, the number of these stacks in use can shrink and grow during a computation, as possible configurations are rejected or added. As will become apparent, at most \( p \) stacks will ever be in use at once, where \( p \) is the number of states in the original BCA. In what follows, it will also be apparent that there is no “communication” between the stacks; i.e., the extended BCA that results is still equivalent in computational power to the BCA from which it originated.

Given an initially nondeterministic acceptor A, the algorithm constructs sets of states from A, and extends the transition table \( M \) to include transitions between these sets of states. Transitions are also defined between individual states of A and sets of states, and between sets of states and individual states. In these transitions involving sets of states, each state in a set is associated with a single stack, and the number of stacks in operation during a computation increases or decreases depending on the relative sizes of the sets between which transitions occur.

The algorithm for constructing sets of states operates in two phases: In the first phase, sets of states \( S_{ij} \) are constructed from individual states \( S_i \) of the original acceptor. These constructed sets have the property that, for a given context \( (K_{<b}, a_b) \), each state in \( S_{ij} \) corresponds to one of the configurations that can result from a configuration containing state \( S_i \). In the second phase, the sets of states \( S_{ij} \) from the first phase are used to construct further sets of states \( S_{ik} \). These sets have the property that, for a given context \( (K', a') \), each state in \( S_{ik} \) appears in a configuration derivable from some state in \( S_{ij} \). There are two cases in each phase of the extension algorithm, corresponding to the erasure or non-erasure of an input-string symbol by the configurations active in some step of a computation. The extension algorithm acts so as to force the state-set transitions of an acceptor to proceed by first erasing as many pushdown-store symbols as can be erased before all states in the set that results are ready to erase an input-string symbol. Hence, on transitions from a set of states in which, for some context, at least one state in the set erases a stack symbol, the remaining states that take part in the transition are “inactive.” These inactive states present in a transition are carried along from one transition to the next until a configuration is reached in which all states in that state set can read the input-string symbol simultaneously. The formalism of the extension algorithm in the following section merely implements these ideas, and provides a method of determining whether, for a given BCA, this extension algorithm is adequate to prevent the necessity of backtracking.

Multiple configurations

A multiple configuration \( C' \) of some acceptor A is a triple

\[ C' = \left( (\# v_1, \ldots, \# v_m) S, w \# \right) \]

where the \( v_i \) are in \( N^* \), \( S \) is in \( (P(Q) - Q) (P(Q) \) is the
set of subsets of Q, w is in T*, and the number of states in S8 is m.

Given a BCA, let 1- be the relation on

\[ (S \times Q \times T \times S) \cup (S \times \cdots \times S \times (P(Q) - Q) \times T \times S) \]

defined as follows:

I. Let C1 = (vK1, a1, w),

where 

\[ [(v = e) \& (K = \emptyset) \& (v \in \{K\} \times (N - \{\emptyset\})) \& (a = \emptyset) \& (w \in (T - \{\emptyset\}))] \]

\& \( (K = \emptyset) \& (a = \emptyset) \& (w \in (T - \{\emptyset\})) \]

\& \( ~[(K = \emptyset) \& (a = \emptyset)] \& ~[S \in P(Q) - Q] \).

II. Let Cj = ((vKj, \ldots, vKj)Sjdw),

where 

\[ [(v = e) \& (K = \emptyset) \& (v \in \{K\} \times (N - \{\emptyset\})) \& (K \in (N - \{\emptyset\}))] \]

\& \( (w = e) \& (d = \emptyset) \& (w \in (T - \{\emptyset\})) \]

\& \( (d \in (T - \{\emptyset\})) \]

\& \( ~[(K = \emptyset) \& (d = \emptyset)] \& [S \in P(Q) - Q] \)

In reality, Cj stands for t different BCA configurations, one configuration corresponding to each triple

\( (vKj, a, d) \), where j = 1, \ldots, t and a = Sa.

To discover what configurations can result from C1 in a computation, it is necessary to trace the descendants of each of the t configurations represented by Cj. In the process of tracing descendant configurations, our strategy will be to force as many transitions as possible in which pushdown-store symbols are erased by states in the sets constructed. Ultimately, a state or a set of states will result in which, for one or more contexts, that state or state set can erase an input-string symbol. The following construction illustrates this principle:

(a) Let S = \{B1, (a1) \[(a1 \in S) \& \[(y1, B1, d) \in M(K1, a1, d)] \}

\& \( [y1 \in (\{K1\} \cup \{e\}) \times (N \cup \{e\})] \}

Then,

\[ C2 = ((y1, \ldots, ym) S \times aw) \]

and

\[ C1 \vdash C2. \]

We say that

\[ M(K, S, a) = ((y1, \ldots, ym), S, a). \]

(b) Let S = \{B2, (a2) \[(a2 \in S) \& \[(v1, B2, a) \in M(K2, a2, d)] \}

\& \( [v1 \in (\{K2\} \cup \{e\}) \times (N \cup \{e\})] \}

Then,

\[ C2 = ((v1, \ldots, vy) S \times w) \]

and

\[ C1 \vdash C2. \]

We say that

\[ M(K, S, a) = ((y1, \ldots, ym), S, a). \]
Note here that \( S_B \) in one of the transitions above could be a set consisting of one or more states. When \( S_B \) is a set consisting of one state, then the transition defined from \( S \) to \( S_B \) has gotten rid of the alternative configurations represented by \( S_B \) and its contexts.

We see that the transitions constructed above from a single configuration to a multiple configuration and from one multiple configuration to another preserve the actions that would be taken by the original BCA. In particular, the one successful reduction sequence of a BCA over a string \( x \) is contained in the extended reduction sequence involving multiple configurations. The extra stacks used during a multiple configuration reduction sequence simply keep track of additional possibilities until all but one sequence of configurations is eliminated.

In part I of the extension algorithm, the transitions are not uniquely defined for those BCA's in which, for a state \( s \) and context \( (K, a) \), the following two conditions apply for the same context \( (K, a) \):

\[
[(K \cup \{e\}) \times (N \cup \{e\}) \times Q \times \{a\}] \supseteq M(K, s, a) \\
\& [(K \times (N \cup \{e\}) \times Q \times \{e\}] \supseteq M(K, s, a)
\]

When these conditions occur simultaneously, the necessity of simultaneously erasing and not erasing the same input-string symbol during a transition is incompatible with our parallel-processing algorithm. It may be possible to use "lookahead" techniques to decide which of the two types of transitions above should take place by scanning further symbols of the input tape. The use of these lookahead techniques to improve the extension algorithm will be discussed in another paper.

The remaining condition that leads to difficulties in the algorithm arises when there exists a state \( g \) in some multiple configuration \( S_g \), such that \( g \) is immediately descended from two or more states \( y_1, \ldots, y_n \) in some \( S_y \) for which

\[
M((w_1, \ldots, w_m), S_y, v) = ((y_1, \ldots, y_n), S_y, u)
\]

For this transition, \( \vdash \) is not uniquely defined, since there is no longer a one-to-one correspondence of pushdown-store strings and states of \( S_y \). In such a case, more than one reduction sequence may exist for a string in the acceptor's language. When these two degenerate conditions arise during the extension of a BCA, it is instructive to rewrite BCA as a rightmost parsing algorithm to see whether the same degenerate conditions arise when parsing strings of that language from right to left.

We can next define the language accepted by a BCA in terms of a computation involving sequences of multiple configurations. We say that \( C_1 \vdash C_m \) when there exists a sequence of (possibly multiple) configurations \( (C_1, \ldots, C_m) \) such that

\[
C_k \vdash C_k \quad \text{for} \quad k = 2, \ldots, m.
\]

The language of an acceptor \( A \) which is extended to handle multiple configurations is then given by

\[
L(A) = \{x: (x \in T^* \setminus \{e\}) \& \left[ \left( \left( S_B \right) \vdash (S_F) \right) \lor \left( (S_B) \vdash (S_F) \right) \right] \lor \left( (S_B) \vdash (S_F) \right) \}
\]

With these preliminaries in mind, we can state the following theorems that are proved in (13).

**Theorem 1** Let \( P \) be a BCA for which \( \vdash \) is uniquely defined for multiple configurations. Let

\[
P' = (Q', T, N', M', \delta, S_0, F)
\]

with \( Q' \subseteq P(Q), N' \subseteq N \cup N \times N \times \ldots \times N \times N \), and \( M' \) the original \( M \) of \( P \) together with the transitions defined on multiple configurations.

Then,

\[
L(P) = L(P').
\]

That this is so follows from the observation that, for \( \vdash \) uniquely defined on multiple configurations of \( P \), all computations of \( P \) over some input string \( x \) in \( L(P) \) are contained in a single computation of \( P' \) over \( x \). Conversely, no computation of \( P' \) over some input string \( y \) will succeed unless \( y \) is in \( L(P) \).

**Theorem 2** Let \( P \) be a BCA constructed from grammar \( G \) having multiple configurations for which \( \vdash \) is uniquely defined. Then \( L(G) \) is unambiguous. That this is so arises from the fact that the conditions for uniqueness of \( \vdash \) also insure uniqueness of leftmost parses in \( L(G) \).

### Upper bounds on storage and computation times

This concluding section contains a fairly elementary proof of the fact that BCA's can accept their languages in time directly proportional to the length of their input strings (or programs to be translated). Our reason for including this proof is to emphasize the need for a basis of comparison between different compiler-writing systems. Thus, if compiler-writing system \( A \) can produce a single-scan FORTRAN compiler whose translation
speed is bounded by \( n^2 \) (with \( n \) the length of an input program), and compiler-writing system B claims a speed of, say, \( n \log(n) \), it would seem fairly obvious that system C, with speed \( 3n \), would be the economic choice for a fast, single-scan compiler system. Again, if there are limitations on computer memory size available for the compiler, and if the compiler is to run in a "load and go" or, possibly, a "reentrant" mode, it is desirable to pick a compiler-writing system that uses as little "core" as possible.

The actual size of one of our compilers is determined by the number of rules in the grammar of its language and, also, by the length of these rules. Roughly speaking, the number of "states" of a compiler is less than or equal to the number of rules in its normal-form grammar of the types

\[ A_i \rightarrow A_\alpha A_\beta \quad \text{and} \quad A_j \rightarrow A_\rho A_\sigma. \]

The number of comparisons necessary to specify a transition away from one state is bounded by the number of contexts that can determine a transition from that state. Thus, the size of the compiler-program is related to the number of rules used in the language and to their lengths.

We can speak more quantitatively about the amount of space required for the pushdown store of a deterministic BCA. Let \( x = a_1 \ldots a_n \) be an input string to some BCA, and let \( y = K_1 \ldots K_m \) represent the string of symbols on the pushdown store at some point during a computation. Then, after symbol \( a_k (k = 1, \ldots, n) \), is "erased" by the BCA,

\[ m \leq k + 1. \]

This is so because, by our definition of a BCA constructed from a grammar \( G \):

(a) For each input symbol erased, the stack can increase in length by at most one symbol.
(b) For each stack symbol erased, at most one symbol can take its place.

Hence, there are never more than \((n + 1)\) symbols on the stack, where \( n \) is the length of the input string. In the case of an extended BCA with multiple-state configurations, there can never be more than \( k \) stacks active at once, where \( k \) is the number of states in the original BCA. For such a BCA, then, there is an upper bound of \( k(n + 1) \) symbols stored on stacks during a computation.

In order to derive an upper bound for computation time of a BCA, we must first discover an upper bound on the number of actions that can be taken by a BCA without erasing an input-string symbol during a computation. Let \( p \) be the number of rules in the grammar for a BCA such that the rules form a chain

\[ A^{(i-1)} \rightarrow A^{(i)} \quad i = 2, \ldots, p, \]

such that all rules of the chain have at least one context in common, and such that \( p \) denotes the length of the longest chain of rules of this sort in the grammar. Then, without the erasure of a symbol from the pushdown store and the input string, at most \( p \) state transitions can occur.

If \( g_k \) symbols are on the pushdown store after input string symbol \( a_k \) has been erased, at most

\[ (1 + g_k)(1 + p) \]

state transitions can occur when erasure of stack symbols is allowed before symbol \( a_{k+1} \) is erased. However, if only \( w_k \) symbols are removed from the stack, then at most

\[ (1 + w_k)(1 + p), \quad w_k = 0, 1, \ldots, p \]

state transitions can take place before \( a_{k+1} \) is erased.

Next, we can ask what total number of symbols can be erased from the stack during any computation, i.e., what is the maximum value of

\[ \sum_{k=1}^n w_k \ ? \]

To answer this question, we note again that our BCA model only allows a new stack symbol to be written as a result of the erasure of an input-string symbol. Since, for an input string of length \( n \), no more than \( n \) symbols can be written on the stack, no more than \( n \) symbols can be extracted from the stack during any computation. Thus,

\[ \sum_{k=1}^n w_k \leq n. \]

Finally, we can arrive at an upper bound for the number of configurations that can appear during the computation of a deterministic BCA over a string \( x \) of length \( n \):

\[ \text{MAX} \leq (n + 1) + (p + 1)(w_1 + \ldots + w_n + n) \] or \[ \text{MAX} \leq n(2p + 3) + 1 \]

We know also that

\[ \text{MAX} \geq n + 1, \]
where this lower bound is reached when the BCA of some grammar has an empty pushdown-store vocabulary; i.e., is a finite-state acceptor. Hence,

\[ n + 1 \leq \text{MAX} \leq n(2p + 3) + 1 \]

The upper bound on the number of configurations during a computation also holds for extended BCA's having multiple-state configurations. This is because the computations of the nondeterministic BCA from which the extended BCA was constructed are all included in the computations of that extended acceptor.

ACKNOWLEDGMENTS

The author expresses his sincere gratitude to Dr. Michael Harrison for his thoughtful criticisms and suggestions on the subject of this paper. The research for this paper was supported in part by N.A.S.A. grant NSG-398 to the Computer Science Center of the University of Maryland.

REFERENCES

1 J. Eickel, M. Paul, F. L. Bauer, K. Samelson
   A syntax-controlled generator of formal language processors
   Comm ACM 6 August 1963 451-455
2 R. W. Floyd
   Syntax-directed operator precedence
   J ACM 10 July 1963 216-233
3 S. Ginsberg, S. Greibach, M. Harrison
   Stack automata and compiling
   J ACM 14 January 1967 172-201
4 S. Ginsberg, M. Harrison
   One-way nondeterministic real-time list-storage languages
   J ACM 15 July 1968 428-446
5 S. Ginsburg
   The Mathematical Theory of Context-Free Languages 1966
6 S. Greibach
   Inversion of phrase structure generators
   Doctoral Dissertation Harvard University 1963
7 L. H. Haines
   Generation and recognition of formal languages
   Doctoral Dissertation MIT 1965
8 E. T. Irons
   A syntax-directed compiler for ALGOL 60
   Comm ACM 4 January 1961 51-55
9 D. E. Knuth
   On the translation of languages from left to right
   Information and Control 8 December 1965 607-639
10 P. M. Lewis II, R. E. Stearns
    Syntax-directed translation
    J ACM 15 July 1968 465-488
11 C. Pair
    Arbres, piles et compilation
    Rev. Francaise Trait Inf 7 1964 199-216
12 A. Rosenberg
   On the independence of real-time definability and certain structural properties of context-free languages
   J ACM 15 October 1968 672-679
13 V. B. Schneider
   The design of processors for context-free languages
   National Science Foundation Memorandum Northwestern University August 1965
14 V. B. Schneider
   Pushdown-store processors of context-free languages
   Doctoral Dissertation Northwestern University 1966
15 V. B. Schneider
   Syntax checking and parsing of context-free languages by pushdown-store automata
   Proc S J C C 1967
16 V. B. Schneider
   A fast translator system for the EULER programming language
   Computer Science Center Technical Report University of Maryland 1969
17 N. Wirth, H. Weber
   EULER: A generalization of ALGOL and its formal definition: parts I and II
   C ACM 3 January-February 1966 13-25 and 89-99