T. E. CHEATHAM, JI., Editor

On the Relative Efficiencies of Context-Free Grammar Recognizers

T. V. GRIFFITHS AND S. R. PETRICK
Air Force Cambridge Research Laboratories, Bedford, Mass.*

A number of diverse recognition procedures that have been proposed for parsing sentences with respect to a context-free grammar are described in this paper by means of a common device. Each procedure is defined by giving an algorithm for obtaining a nondeterministic Turing Machine recognizer that is equivalent to a given context-free grammar. The formalization of the Turing Machine has been chosen to make possible particularly simple descriptions of the parsing procedures considered.

An attempt has been made to compare recognition efficiencies for the procedures defined. For a few simple grammars and sentences a formal comparison has been made. Empirical comparison of the recognition of more realistic programming languages such as LISP and ALGOL has been made by means of a program which simulates the Turing Machine on the Univac M-460 Computer. Several algorithms for producing grammars equivalent to a given context-free grammar have been considered, and the increase in recognition efficiency they afford has been empirically investigated.

1. Introduction

The class of grammars called *context-free* (CF) by Chomsky [1] has been utilized in various linguistic theories, both as the sole component and as just one of several components of a natural language grammar. In addition, CF grammars have come to play a dominant role in the specification and translation of programming languages. It has been found that CF grammars are at least to some extent adequate for specifying the syntax of programming languages and that the structural descriptions assigned by these grammars are of practical utility in producing compilers.

The first application of CF grammars to computer programming seems to have been made by Backus [2] in officially defining the syntax of the ALGOL language. Except for notation, so-called "Backus Normal Form" is identical to a CF grammar specification.

* Applied Mathematics Branch.

It is not our purpose here to discuss the appropriateness of the CF grammar for specific applications. Indeed, the authors' basic disagreement here makes a joint statement impossible. Instead we assume the utility of the CF grammar for some applications and confine ourselves to investigating the efficiencies of different recognition procedures for this class of grammars.

2. Context-Free Grammars

A CF grammar is represented by a quadruple (I, T, S, P) where $I \cup T$ is an alphabet and P contains rules for rewriting symbols from I as strings of symbols from $I \cup T$. These rules have the form:

$$A \rightarrow B_1 \cdots B_n , \qquad (n \geqq 1)$$

where the symbol A is rewritten as the string $B_1 \cdots B_n$. The string $D_1 \cdots D_q$ is derivable from the string $C_1 \cdots C_p$ if $D_1 \cdots D_q$ can be obtained from $C_1 \cdots C_p$ by a finite sequence of applications of rules from P. The sets I and T are such that

$$I = \{C:C \rightarrow D_1 \cdots D_q \in P\}$$

and

$$I \cap T = \Phi .$$

The members of T are called *terminals*, the members of I, *nonterminals*. There is a designated symbol $S \in I$, and terminal strings derivable from S are called *sentences of the grammar*.

We define the structural description of a sentence with respect to a given CF grammar $G = (I, T, S, P)$ by giving a CF grammar $G' = (I, T', S, P')$ where

$$T' = T \cup \{]\} \cup \{\underset{X}{[}:X \in I\} ,$$

and for every rule $A \rightarrow B_1 \cdots B_n \in P$ there corresponds a rule

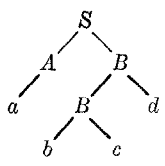
$$A \rightarrow \underset{A}{[}B_1 \cdots B_n]$$

in P'. The sentences of G' are said to be structural descriptions of the debracketed strings, which are sentences of G. These structural descriptions can be represented by labeled tree diagrams.

As an example, consider the grammar G_1 :

1.	$S \rightarrow AB$	4.	$B \rightarrow bc$
2.	$A \rightarrow a$	5.	$B \rightarrow Bd$
3.	$A \rightarrow ABb$		

The string *abcd* is a sentence generated by G_1 and a structural description of *abcd* is given by the labeled bracketing $[_S [_A a] [_B [_B bc] d]]$ or by the tree



The recognition problem with which we are concerned may now be stated. Given a CF grammar G with designated symbol S, and given a string ϕ consisting of a finite number of terminal symbols, determine whether or not ϕ is a sentence of the language given by G. If it is, find all of the structural descriptions of ϕ with respect to G.

A number of diverse algorithms have been proposed as practical solutions to this problem. A good many of these have been addressed not to the problem of finding all structural descriptions, but rather to the problem of finding only a single structural description, often with the hope that the language in question is not ambiguous. Most of the algorithms have been described in some sense by flowcharts and by informal discussion, but actual definition has been accomplished for the most part by means of detailed computer programs in which "the forest is inevitably lost for the trees." A few algorithms have been rather elegantly presented, however, through the use of various pushdown store automata models [1, 3, 4]. In the present paper we explicitly state and test various recognition algorithms by expressing them as Turing machine programs, not only for purposes of clarity and conciseness, but also to attempt to hold certain variables constant so as to compare apples and oranges rather than apples and elephants. The extent to which we were successful in this regard is left to the reader to estimate from the description of the procedure we followed. The only previous effort to compare different algorithms seems to be a study of "syntax-directed" compiling undertaken by a working group of the Los Angeles Chapter of the ACM and reported at the August 1963 ACM National Conference [5, 6]. Not only were two different algorithms separately considered by two groups, but their choice of algorithms differed with respect to the property of *selectivity* that we will subsequently define and show to be of paramount importance.

3. The Turing Machine

In a preliminary draft of this paper we expressed our recognition algorithms for CF grammars by giving constructions for obtaining equivalent pushdown store automata. For pedagogical reasons we have decided here to use a more convenient Turing machine (TM) instead. In doing so we are, of course, making weaker claims about the particular recognition procedures we present, but the equivalence between CF grammars and pushdown store automata is well known and not of interest here.

Generalizing the notion of a CF rewriting rule, we allow rules of the form:

$$A_1 \cdots A_m \to B_1 \cdots B_n$$

where $A_1 \cdots A_m$ and $B_1 \cdots B_n$ are strings of symbols. In such a rule either string, or both, may be the null string, which we denote by Λ.

A TM as we use it here is represented by a pair (V, R) where V is an alphabet and R is a set of TM instructions. This machine has two pushdown tapes, α and β, whose ends are marked by a special symbol $\not\# \notin V$. The rules in R have the form:

$$(A_1 \cdots A_m, C_1 \cdots C_p) \to (B_1 \cdots B_n, D_1 \cdots D_q)$$

meaning that the rule

$$A_1 \cdots A_m \to B_1 \cdots B_n$$

is applied to the top m symbols of α and the rule

$$C_1 \cdots C_p \to D_1 \cdots D_q$$

is applied to the top p symbols of β if both applications are possible.

We make no provision that the TM be deterministic, that is, that to each tape configuration only one rule from R be applicable. When m rules are applicable to a given configuration, the TM is to be thought of as following all m paths in parallel. A TM path terminates when no TM instruction is applicable to the path's current configuration. The TM halts when all its paths have terminated.

A TM T recognizes a string $A_1 \cdots A_m$ as some syntactic unit S if there is some finite sequence of T's rules producing $\not\#$ on α and β when T is started with $A_1 \cdots A_m \not\#$ on α and $S\not\#$ on β. Every such sequence of rules is a *parsing sehuence* for $A_1 \cdots A_m$ (recognized as S).

Having defined a machine, the TM, convenient for our purposes, it is rather easy to compactly and precisely define a variety of recognition algorithms for CF grammars. Each of these algorithms will be specified by giving a TM program which recognizes the same strings as those defined by a given CF grammar. It is our purpose in this paper merely to compare relative parsing efficiencies and not to actually find structural descriptions. Hence, the TM we are using is merely an acceptance device. However, we assert that for all of the algorithms defined in this paper, the structural descriptions are in one to one correspondence with the parsing sequences of TM instructions, and for each parsing sequence, there is an effective procedure for obtaining the corresponding structural description.

Furthermore, we assert that these programs capture the essence of the parsing philosophies we seek to compare. Of course the extent to which they are identical to implemented procedures now in use hinges on the way in which those procedures are defined. In the last analysis we are precisely specifying and comparing certain recognition algorithms for CF grammars, and the extent to which they are accurate models of existing algorithms is in varying degrees subject to debate.

4. Recognition Algorithms Considered

There are many ways by which the typology of recognition algorithms for CF grammars can be approached. For example, one means of classification is related to the general directions in which creation of a structural description tree proceeds: top-to-bottom, bottom-to-top, left-to-right and right-to-left. Some procedures are described in these terms only with difficulty, and others seem to allow no such classification. Nevertheless, the principal algorithms we describe are at least roughly characterized by these descriptions.

Another means of classification is related to the devices which are used to decrease the total number of TM instructions a given algorithm type uses to find all structural descriptions. These devices are of two basic types: (1) techniques for merging similar sections of different TM paths and (2) techniques for shortening the length of fruitless TM paths. Section 5 on equivalent grammars deals with a small subset of the type 1 (merging) reduction techniques, but the rest of this survey is exclusively devoted to type 2 (selective) reduction techniques. In order to roughly illustrate selective devices we note that in some top-to-bottom procedures, where the tree structure is pieced together from the designated symbol on down, the number of alternate substructures considered can be sharply reduced by taking cognizance of the terminal string being analyzed and realizing that certain choices cannot possibly lead to the required terminal string. A similar selective use of nonterminals can be made in determining structure from the bottom up.

For a grammar $G = (I, T, S, P)$ we give first a simple nonselective top-to-bottom algorithm hereafter identified as algorithm NTB. This is essentially the construction given by Matthews [33] as a corollary to his proof that for each one-way context-sensitive grammar there is an equivalent CF grammar.

NTB Algorithm

Conditions	*TM Instructions*
$A \to V_1 \cdots V_n \in P$	$(\Lambda, A) \to (\Lambda, V_1 \cdots V_n)$
$a \in T$	$(a, a) \to (\Lambda, \Lambda)$

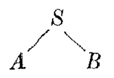
As an example, the TM corresponding to the previously given sample grammar G_1 is:

1. $(\Lambda, S) \to (\Lambda, AB)$
2. $(\Lambda, A) \to (\Lambda, a)$
3. $(\Lambda, A) \to (\Lambda, ABb)$
4. $(\Lambda, B) \to (\Lambda, bc)$
5. $(\Lambda, B) \to (\Lambda, Bd)$
6. $(a, a) \to (\Lambda, \Lambda)$
7. $(b, b) \to (\Lambda, \Lambda)$
8. $(c, c) \to (\Lambda, \Lambda)$
9. $(d, d) \to (\Lambda, \Lambda)$

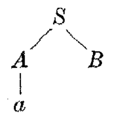
To illustrate the operation of this device we include below a sequence of instructions that accepts the tape *abcd*.

Instruction Applied	*Tape α*	*Tape β*
	abcd#	*S*#
1. $(\Lambda, S) \to (\Lambda, AB)$		
	abcd#	*AB*#
2. $(\Lambda, A) \to (\Lambda, a)$		
	abcd#	*aB*#
6. $(a, a) \to (\Lambda, \Lambda)$		
	bcd#	*B*#
5. $(\Lambda, B) \to (\Lambda, Bd)$		
	bcd#	*Bd*#
4. $(\Lambda, B) \to (\Lambda, bc)$		
	bcd#	*bcd*#
7. $(b, b) \to (\Lambda, \Lambda)$		
	cd#	*cd*#
8. $(c, c) \to (\Lambda, \Lambda)$		
	d#	*d*#
9. $(d, d) \to (\Lambda, \Lambda)$		
	#	#

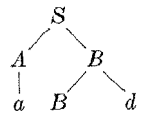
To give an intuitive feeling for this acceptance sequence we offer a description of the way it corresponds to systematically tracing through a structural description tree. Instruction 1 corresponds to obtaining the structure:



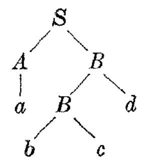
Instruction 2 postulates that we have



and instruction 6 confirms this is compatible with the input tape. Instruction 5 expands B to Bd giving



and instruction 4 expands B to bc giving the structural description:



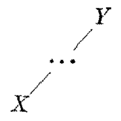
The remaining three instructions 7, 8 and 9 confirm that the postulated terminal symbols are compatible with the symbols on the input tape.

Basically, the tree is constructed from the top down by applying at each step a rule of P which expands the current leftmost free nonterminal symbol.

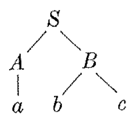
We next show how to modify this NTB algorithm to obtain a selective top-to-bottom (STB) procedure. To do this we make use of the so-called precedence matrix (complete-connectivity matrix). This is a Boolean matrix $P(X, Y)$ which has one row and column corresponding to each symbol in $I \cup T$ of G. The element $P(X, Y)$ is a truth function whose value is t, if and only if the grammar permits a bracketing of the form

$\underset{Y}{[}\overbrace{[\cdots[}^{\text{any number of [’s}}X\cdots]$ if $X \in T$ and $\underset{Y}{[}\overbrace{[\cdots}^{\text{any number of [’s}}\underset{X}{[}\cdots]$ if $X \in I$,

that is, if the grammar permits any sequence of left branchings that take us from Y down to X



For example, in the previously given grammar G_1, $P(a, S)$ is true because G_1 permits $\underset{S}{[}\underset{A}{[}\,a]\underset{B}{[}bc]]$ or equivalently, the tree structure



There exist several quick, mechanical procedures for obtaining the matrix $P(X, Y)$ which corresponds to a CF grammar G [16, 17].

Using precedence matrices we now give the STB procedure.

STB Algorithm

Conditions	*TM Instructions*
$A \rightarrow V_1 \cdots V_n \in P$ and either $P(a, V_1)$ or else $a = V_1$	$(a, A) \rightarrow (a, V_1 \cdots V_n)$
$a \in T$	$(a, a) \rightarrow (\Lambda, \Lambda)$

Corresponding to a rule $A \rightarrow V_1 \cdots V_n$ there are as many TM instructions as there are pairs (a, V_1) satisfying the above conditions. For example, the STB automaton for Grammar G_1 is:

1.	$(a, S) \rightarrow (a, AB)$	6.	$(a, a) \rightarrow (\Lambda, \Lambda)$
2.	$(a, A) \rightarrow (a, a)$	7.	$(b, b) \rightarrow (\Lambda, \Lambda)$
3.	$(a, A) \rightarrow (a, ABb)$	8.	$(c, c) \rightarrow (\Lambda, \Lambda)$
4.	$(b, B) \rightarrow (b, bc)$	9.	$(d, d) \rightarrow (\Lambda, \Lambda$
5.	$(b, B) \rightarrow (b, Bd)$		

It is merely fortuitous that one TM instruction per rule of G_1 is required. This is a consequence of the fact that in the precedence matrix corresponding to G_1, each column contains just a single one in the rows denoting symbols of I.

The reader may have noted that both algorithms so far considered will produce nonhalting TM programs for a grammar which allows $P(X, X)$ for some nonterminal X. We preclude this trouble for a large class of CF grammars by using the following type 2 reduction technique: If at any stage in a TM path the number of symbols on β exceeds the number of symbols on α, terminate the path. The group at Harvard proposed this expedient and called it the *shaper*. The class of CF grammars for which this technique is not sufficient is the class each of whose members has a cycle of rules

$$A_1 \rightarrow A_2\,, A_2 \rightarrow A_3\,, \cdots, A_{n-1} \rightarrow A_n = A_1\,.$$

If, for some sentence in such a grammar, there is a structural description involving one of the cyclic nonterminals, then it is clear that there must be an infinite number of structural descriptions of the sentence. By using the infinity lemma, we can show there must be an infinite TM path, and hence the TM can never halt.

It is not hard to see that for all other CF grammars the shaper is sufficient to guarantee halting. Consider a CF grammar G having no cyclic nonterminals. If G has p nonterminals, then after every $p+1$ rule applications on a TM path one of the following must be true: (1) α is shorter, or (2) β is longer. Under such circumstances, the shaper must terminate all unfruitful paths sooner or later, and the TM must therefore stop.

Predictive syntactic analysis [8, 34] is related to our top-to-bottom algorithms as follows: Let us define a CF grammar to be in standard form if every production is of the form $X \rightarrow V_1 \cdots V_n$ where $X \in I$, $V_1 \in T$ and $V_i \in I$ for $i = 2, \cdots, n$. Greibach [19] has shown that every CF grammar without cyclic nonterminals is weakly equivalent, with preservation of ambiguity, to a CF grammar in standard form, and has given a construction for obtaining equivalent standard form rules. The method of predictive analysis assumes a set of productions in standard form, and its underlying CF recognition procedure[1] is modeled by the following algorithm (PA):

PA Algorithm

Conditions	*TM Instructions*
$A \rightarrow V_1 \cdots V_n \in P$ $V_1 \in T, V_i \in I\ (i = 2, \cdots, n)$	$(V_1\,, A) \rightarrow (\Lambda, V_2 \cdots V_n)$

Note that the PA algorithm is basically the same as both the NTB and the STB algorithms applied to CF grammars in standard form. For the STB algorithm the rules $(a, a) \rightarrow (\Lambda, \Lambda)$ can be eliminated if we replace rule $(a, A) \rightarrow (a, V_1 \cdots V_n)$ by the rule $(a, A) \rightarrow (\Lambda, V_2 \cdots V_n)$.

The next algorithm we consider is nonselective and basically bottom-to-top (NBT).

NBT Algorithm

Conditions	*TM Instructions*
$A \rightarrow V_1 \cdots V_n \in P$ $X \in I$	$(V_1\,, X) \rightarrow (\Lambda, V_2 \cdots V_n tAX)$
	$(\Lambda, t) \rightarrow (t, \Lambda)$
$A \in I$	$(t, A) \rightarrow (A, \Lambda)$
$B \in I \cup T$	$(B, B) \rightarrow (\Lambda, \Lambda)$

The symbol t used in this description is a special marker symbol that belongs neither to T nor I. Let us denote by the use of I (or T) in a TM pseudo-rule the set of rules in which each member of I (T) replaces every I (T) in the pseudo-rule. For example, if as in grammar G_1 $I = \{S, A, B\}$ then $(t, I) \rightarrow (I, \Lambda)$ stands for the three TM rules

$$(t, S) \rightarrow (S, \Lambda) \qquad (t, A) \rightarrow (A, \Lambda) \qquad (t, B) \rightarrow (B, \Lambda)$$

With this convention the NBT program corresponding to grammar G_1 is the 26 rules denoted by:

1.	$(A, I) \rightarrow (\Lambda, BtSI)$	6.	$(\Lambda, t) \rightarrow (t, \Lambda)$
2.	$(a, I) \rightarrow (\Lambda, tAI)$	7.	$(t, I) \rightarrow (I, \Lambda)$
3.	$(A, I) \rightarrow (\Lambda, BbtAI)$	8.	$(I, I) \rightarrow (\Lambda, \Lambda)$
4.	$(b, I) \rightarrow (\Lambda, ctBI)$	9.	$(T, T) \rightarrow (\Lambda, \Lambda)$
5.	$(B, I) \rightarrow (\Lambda, dtBI)$		
	$I = \{S, A, B\}$		$T = \{a, b, c, d\}$.

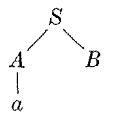
To give an idea of how this NBT program operates we include the acceptance sequence for the string *abcd* as we did for the NTB case. The slightly greater number of

[1] Several other devices have been employed by the group at Harvard to increase recognition efficiency for their grammar, but we have not considered these devices because they are not applicable to arbitrary CF grammars or have been proposed too recently to permit their inclusion in this paper.

instructions for the NBT case is of no significance because we have not considered other instructions which the TM must execute in finding all possible structural descriptions.

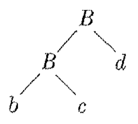
Instruction Applied	*Tape α*	*Tape β*
	$abcd\#$	$S\#$
2. $(a, S) \rightarrow (\Lambda, tAS)$		
	$bcd\#$	$tAS\#$
6. $(\Lambda, t) \rightarrow (t, \Lambda)$		
	$tbcd\#$	$AS\#$
7. $(t, A) \rightarrow (A, \Lambda)$		
	$Abcd\#$	$S\#$
1. $(A, S) \rightarrow (\Lambda, BtSS)$		
	$bcd\#$	$BtSS\#$
4. $(b, B) \rightarrow (\Lambda, ctBB)$		
	$cd\#$	$ctBBtSS\#$
9. $(c, c) \rightarrow (\Lambda, \Lambda)$		
	$d\#$	$tBBtSS\#$
6. $(\Lambda, t) \rightarrow (t, \Lambda)$		
	$td\#$	$BBtSS\#$
7. $(t, B) \rightarrow (B, \Lambda)$		
	$Bd\#$	$BtSS\#$
5. $(B, B) \rightarrow (\Lambda, dtBB)$		
	$d\#$	$dtBBtSS\#$
9. $(d, d) \rightarrow (\Lambda, \Lambda)$		
	$\#$	$tBBtSS\#$
6. $(\Lambda, t) \rightarrow (t, \Lambda)$		
	$t\#$	$BBtSS\#$
7. $(t, B) \rightarrow (B, \Lambda)$		
	$B\#$	$BtSS\#$
8. $(B, B) \rightarrow (\Lambda, \Lambda)$		
	$\#$	$tSS\#$
6. $(\Lambda, t) \rightarrow (t, \Lambda)$		
	$t\#$	$SS\#$
7. $(t, S) \rightarrow (S, \Lambda)$		
	$S\#$	$S\#$
8. $(S, S) \rightarrow (\Lambda, \Lambda)$		
	$\#$	$\#$

To give the reader an intuitive idea of the nature of the above acceptance sequence we offer the following commentary: First, instructions 2, 6 and 7 recognize that a is a sole daughter of A. Instruction 1 postulates that if the syntactic unit A which has been formed is followed by a B, a higher level unit S will be determined according to $S \rightarrow AB$. (That is, the structure

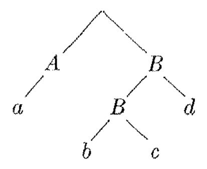


is postulated.)

Instruction 4 postulates that the required construction B may be built up from the string $bcd\#$ by using the rule $B \rightarrow bc$. Instruction 9 verifies that the required character c occurs next on the input tape and instructions 6 and 7 again signal that a new syntactic construction B has been determined. Instruction 5 postulates that the syntactic construction B may be composed of the immediate constituents B (which has just been obtained) and d. Instruction 9 again verifies that the required d appears next on the input tape, instructions 6 and 7 indicate that a new construction B has been obtained, and instruction 8 indicates that this new construction



fits into the previously determined structure to give



The remaining instructions indicate that the topmost node of the previous construction is S as required.

It again requires only a trivial modification to specify a selective version (SBT) of the previous algorithm.

SBT Algorithm

Conditions	*TM Instructions*
$A \rightarrow V_1 \cdots V_n \in P$ and either $P(A, X)$ or $A = X$	$(V_1, X) \rightarrow (\Lambda, V_2 \cdots V_n tAX)$
	$(\Lambda, t) \rightarrow (t, \Lambda)$
$A \in I$	$(t, A) \rightarrow (A, \Lambda)$
$B \in I \cup T$	$(B, B) \rightarrow (\Lambda, \Lambda)$

For Grammar G_1 the SBT program defined by these correspondences is identical to the previously given NBT rules except in place of the first five NBT rules we have the following SBT rules:

$(A, S) \rightarrow (\Lambda, BtSS)$	$(A, A) \rightarrow (\Lambda, BbtAA)$
$(a, S) \rightarrow (\Lambda, tAS)$	$(b, B) \rightarrow (\Lambda, ctBB)$
$(a, A) \rightarrow (\Lambda, tAA)$	$(B, B) \rightarrow (\Lambda, dtBB)$
$(A, S) \rightarrow (\Lambda, BbtAS)$	

We assert that this algorithm is essentially the one described by Irons [10, 11] and adopted by Bastian [13], Willett and Helwig [14], and Ingerman [15]. Instead of the condition "$P(A, X)$ or $A = X$," Irons used the condition "$P(V_1, X)$;" and, hence, his procedure is somewhat less selective than the SBT algorithm given above. The more selective condition was apparently first suggested by Bastian (who called it *look-ahead*). We see that it is the analog of the STB condition "$P(a, V_1)$ or $a = V_1$."

No analog of the shaper technique is necessary for the NBT and SBT algorithms: both produce halting TM programs for CF grammars having no cyclic nonterminals. If G is such a CF grammar, with p nonterminals then the number of symbols on α must decrease every $6p$ instructions along any TM path in the NBT and SBT algorithms for G.

It is not hard to see that for a standard form grammar, the NBT and SBT algorithms can be reduced to the predictive analysis (PA) algorithm. Thus for standard grammars, the NTB, STB, NBT, SBT and PA algorithms are strategically the same.

Besides the restriction of predictive analysis to standard grammars, another difference between the predictive analysis and Irons type approaches to date is that the former have been addressed to the finding of all structural descriptions while the latter have been satisfied with a single analysis.

The reasons for these decisions are in no way related to the underlying algorithms but are instead reflections of economic and practical utility. The predictive analyzers need all structural descriptions, while the programming people have been content to risk the dangers of ambiguity or failure to find any valid analysis in the interest of economy. An assumption frequently made is that a labeled bracket must always be embedded as the leftmost immediate constituent of a bracket of the same type if this is possible. This type of construction must, of course, be considered in finding all structural descriptions, but the compilers in question preclude all constructions of the type $\underset{B}{[}\ \underset{A}{[}\cdots]\cdots]$, $B \neq A$, if there exists a construction $\underset{A}{[}\ \underset{A}{[}\cdots]\cdots]$.

As an illustration of the case when this procedure leads to trouble, consider the grammar G_2 which is identical to G_1 except that rule 3 of G_1, $A \rightarrow ABb$, is missing in G_2 and G_2 has the additional rule $A \rightarrow Ab$. If the construction $[a]$ is irretrievably lost upon constructing the larger unit $\underset{A}{[}\ \underset{A}{[}a]b]$, the only possible structural description $\underset{S}{[}\ \underset{A}{[}a]\underset{B}{[}\ \underset{B}{[}bc]d]]$ is never found.

Our next algorithm, the nonselective direct substitution (NDS) algorithm, is due to Abbott [36], though it was also suggested earlier by Greibach in a form applicable only to normal grammars; i.e. CF grammars for which all rules are of the type $A \rightarrow BC$ or $A \rightarrow a$ where A, B and $C \in I$ and $a \in T$. This algorithm is also similar to Kay's string rewriting system [37] insofar as general philosophy is concerned. The particular form in which we present the algorithm requires the restriction that it not be applied to a CF grammar in which the designated symbol S appears on the right side of a rule. Any CF grammar not satisfying this restriction can easily be modified to obtain one that does by substituting S' for all occurences of S in the original grammar and adding the rule $S \rightarrow S'$. (S remains the designated symbol.)

NDS Algorithm

Conditions	*TM Instructions*
	$(S, S) \rightarrow (\Lambda, \Lambda)$
$A \rightarrow V_1 \cdots V_n \in P$	$(V_n, V_{n-1} \cdots V_1) \rightarrow (A, \Lambda)$
$B \in I \cup T$	$(B, \Lambda) \rightarrow (\Lambda, B)$

Note that in this algorithm, except when the grammar is a normal grammar, we have departed from the custom of looking only at the top symbols on α and β.

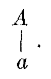
In illustration of this algorithm we present an acceptance sequence for the string *abcd* with respect to grammar G_1. From this grammar we obtain the corresponding NDS TM:

1. $(S, S) \rightarrow (\Lambda, \Lambda)$
2. $(a, \Lambda) \rightarrow (A, \Lambda)$
3. $(B, A) \rightarrow (S, \Lambda)$
4. $(c, b) \rightarrow (B, \Lambda)$
5. $(d, B) \rightarrow (B, \Lambda)$
6. $(b, BA) \rightarrow (A, \Lambda)$
7. $(E, \Lambda) \rightarrow (\Lambda, E)$ for $E \in I \cup T$

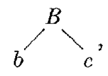
The acceptance sequence for *abcd* is:

Instruction Applied	*Tape* γ	*Tape* $+$
	$abcd\#$	$S\#$
2. $(a, \Lambda) \rightarrow (A, \Lambda)$		
	$Abcd\#$	$S\#$
7. $(A, \Lambda) \rightarrow (\Lambda, A)$		
	$bcd\#$	$AS\#$
7. $(b, \Lambda) \rightarrow (\Lambda, b)$		
	$cd\#$	$bAS\#$
4. $(c, b) \rightarrow (B, \Lambda)$		
	$Bd\#$	$AS\#$
7. $(B, \Lambda) \rightarrow (\Lambda, B)$		
	$d\#$	$BAS\#$
5. $(d, B) \rightarrow (B, \Lambda)$		
	$B\#$	$AS\#$
3. $(B, A) \rightarrow (S, \Lambda)$		
	$S\#$	$S\#$
1. $(S, S) \rightarrow (\Lambda, \Lambda)$		
	$\#$	$\#$

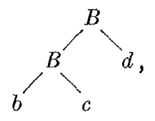
We give an intuitive commentary on the above acceptance sequence: Instruction 2 recognizes a as the syntatic unit A, giving the structure



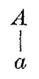
The two 7 instructions postulate that neither A nor b is the rightmost symbol of a higher syntactic unit. The 4 instruction recognizes that we have the structure



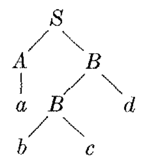
and the 7 instruction postulates that B is not the rightmost symbol of a higher syntactic unit. The 5 instruction recognizes the structure



and the 3 instruction places this with



to obtain



The 1 instruction terminates the sequence.

In giving a selective version of the NDS algorithm, we assume grammars whose rules are of the type $A \rightarrow V_1 \cdots V_n$ where $n = 1$ or 2 only. This makes the selective conditions easier to state, though, of course, these conditions can be extended to the general case.

Denote by $\ll BC \gg$ that there exists a rule $A \rightarrow BC$

Then a selective version of the previous algorithm is given by:

SDS Algorithm

Conditions	*TM Instructions*
	$(S, S) \rightarrow (\Lambda, \Lambda)$
$A \rightarrow D$ $(\exists Q)[\ll NQ \gg \wedge (P(A, Q) \vee A = Q)]$ $\vee [N = S \wedge (P(A, S) \vee A = S)]$	$(D, N) \rightarrow (A, N)$
$A \rightarrow BC$	$(C, B) \rightarrow (A, \Lambda)$
$\{(\exists Q)[\ll NQ \gg \wedge P(E, Q)]$ $\vee [N = S \wedge P(E, S)]\} \wedge (\exists R)[\ll ER \gg]$	$(E, N) \rightarrow (\Lambda, EN)$
$B, C, D, E, N, R \in I \cup T$ $A, Q \in I$	

The SDS instructions for the TM corresponding to grammar G_1 with $A \rightarrow ABb$ deleted are:

1.	$(S, S) \rightarrow (\Lambda, \Lambda)$	5.	$(d, B) \rightarrow (B, \Lambda)$
2.	$(a, S) \rightarrow (A, S)$	6.	$(B, A) \rightarrow (\Lambda, BA)$
3.	$(B, A) \rightarrow (S, \Lambda)$	7.	$(b, A) \rightarrow (\Lambda, bA)$
4.	$(c, b) \rightarrow (B, \Lambda)$	8.	$(A, S) \rightarrow (\Lambda, AS)$

It is not hard to see that the NDS and SDS algorithms both produce halting TM programs for CF grammars having no cyclic nonterminals.

5. Equivalent Grammars

Two CF grammars without cyclic nonterminals that specify the same terminal strings with the same ambiguity are defined to be equivalent. If two grammars are equivalent, there is clearly a procedure for relating structural descriptions in one to structural descriptions in the other. Given an arbitrary CF grammar without cyclic nonterminals, we can easily construct a normal grammar that is equivalent to it. Furthermore, it is easy to convert the structural descriptions assigned by such a normal grammar to the structural descriptions of the given CF grammar, that is, to obtain all of the structural descriptions of a given CF grammar from the structural descriptions of an equivalent normal grammar.

We have already observed that Greibach has shown that for any CF grammar without cyclic nonterminals an equivalent standard form grammar can be constructed. To date, no efficient procedure for relating the structural descriptions of standard form grammars to the CF grammars from which they were constructed has been found.

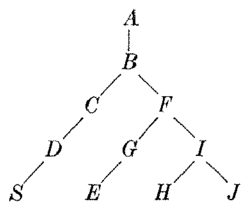
Finally, we will discuss a few other procedures for using equivalent grammars in improving CF recognition efficiency. The first procedure we discuss was suggested by Irons' linked list representation of CF rules. This procedure is of use in those cases where two or more rules share identical leftmost constituents. For example, in the grammar G_1 we have the rules $S \rightarrow AB$ and $A \rightarrow ABb$. Although we devised a CF to TM conversion routine (making use of Irons' linked-list procedure and a subsequent generalization of Bastian's look-ahead procedure to linked-list rules as suggested by Willett and Helwig), it was not implemented because a simpler alternative was found. By appropriately writing an equivalent CF grammar the effect of linked lists can be realized with no additional programming required. The procedure is merely to define new nonterminals so that common leftmost constituents do not occur in the new grammar. For example, the previously cited two rules can be written as the equivalent set

$$S \rightarrow C \qquad A \rightarrow Cb \qquad C \rightarrow AB$$

Single identical initial constituents are not considered in obtaining the new grammar. As a further example, the rules

$$S \rightarrow ABCD \qquad E \rightarrow ABFG \qquad H \rightarrow ABFI \qquad J \rightarrow ABFI$$

with linked-list (tree) representation



can be replaced by the rules

$S \rightarrow WCD$	$H \rightarrow Y$
$W \rightarrow AB$	$J \rightarrow Y$
$E \rightarrow XG$	$Y \rightarrow XI$
$X \rightarrow WF$	

Although we have formalized the procedure for obtaining such a new set of rules from a linked-list rule description, the general idea involved is obvious and will not be elaborated further.

An analogous linked-list procedure which we reference as P_2, distinguishing it from the previous procedure P_1, is applicable to the (STB) algorithm except that here not only must two or more rules share common leftmost constituents, but also they must be rewrite rules of the same nonterminal. This is, of course, a condition less frequently met in actual programming language grammars.

As an example of procedure P_2 consider the previous example with H and J replaced by E and S respectively. We can then write:

$S \rightarrow ABX$		$S \rightarrow ABCD$
$X \rightarrow CD$		$S \rightarrow ABFI$
$X \rightarrow FI$	for	$E \rightarrow ABFG$
$E \rightarrow ABFY$		$E \rightarrow ABFI$
$Y \rightarrow G$		
$Y \rightarrow I$		

Procedures P_1 and P_2 are examples of the type 1 or merging techniques mentioned in Section 3. For the NBT and SBT algorithms, P_1 and P_2 both make TM paths split later than they would for an unmodified grammar thus decreasing the overall number of TM instructions executed. Procedure P_2 has essentially the same effect for the NTB and STB algorithms.

An example of much more extensive use of merging techniques can be found in Kuno's repetitive path eliminator [35]. Such techniques are also implicit in the algorithms of Kay [37] and Robinson [25].

We have already mentioned the extent to which a number of recognition procedures now in use are modeled by the algorithms we have defined. There are, of course, many other recognizers currently in use, and they can be classified into four types.

One set includes other recognizers which clearly are based on an underlying CF model and which reflect at least to some extent one of the algorithms of this paper. Included in this category are the top-to-bottom recognizers of Glennie [20] and Brooker and Morris [21, 22, 23].

In a second category can be grouped those algorithms which, although based on a CF or sequentially definable language model, utilize different algorithms than those we have defined. In this category belongs the procedure of Ledley and Wilson [24] and also the normal grammar procedure of Robinson [25]. The latter is a bottom-to-top method which combines adjacent constituents exhaustively. This procedure is similar to Kay's algorithm [37] except that the order of combining elements differs, and Kay's algorithm is not confined to normal grammars. It is in fact applicable to general rewriting systems. The procedure of Ledley and Wilson is applicable only to sequentially definable languages, which Ginsburg [26] has shown are properly included among the CF languages. This procedure was never programmed, and it is somewhat doubtful whether it ever has been defined. The authors give no proof that their recognition procedure is in fact a recognition procedure, and, of course, this cannot even be considered unless the algorithm is precisely stated.

In a third category we lump those recognizers based on grammars which are properly included among the CF grammars, as is Floyd's precedence grammar [27], or which differ from CF grammar in various ways. In the latter category belongs Gilbert, Hosler, and Schager's analytic grammar [28] which is a greatly restricted context-sensitive grammar for which derivations are not defined in the usual way. The motivation behind both of these grammars was to obtain efficient recognition and to eliminate ambiguity. They raise significant problems of empirical adequacy (as does the CF grammar, for that matter) which we cannot discuss here.

Finally, in a fourth category belong a host of recognizers that are based on no formal model at all. Such approaches as those of Barnett [29] and Ross [30] have not been sufficiently formalized to determine their relationship, if any, to formal models in automata theory or the theory of formal grammars.

6. Empirical Results

Having defined a number of recognition procedures, we now examine their relative efficiencies in accepting particular strings with respect to particular grammars. We have been principally concerned with time (which is roughly proportional to the number of TM instructions required to find all structural descriptions) rather than storage because storage requirements for typical programming language grammars of interest are all within reasonable limits for use with large scale computers, but time requirements are more critical.

We consider first three grammars that are simple examples of grammars that are left branching, right branching and self-embedding:

$$\begin{array}{llllll} G1: & S \to Ab & G2: & S \to aB & G3: & S \to aSb \\ & A \to Ab & & B \to aB & & S \to ab \\ & A \to a & & B \to b & & \end{array}$$

For grammar $G1$, both the NTB and STB algorithms can be shown to take $(n^2 + 7n + 2)/2$ Turing Machine instructions to recognize the sentence ab^n. The corresponding number of instructions required by both the NBT and SBT algorithms is $9n + 5$. For grammar $G2$ the NTB algorithm takes $3n + 2$ instructions to recognize the sentence a^nb as compared to $2n + 2$ instructions for the STB algorithm, $11 \cdot 2^n - 7$ instructions for the NBT algorithm and $4n + 4$ instructions for the SBT algorithm. For grammar $G3$ the number of instructions required to recognize the sentence a^nb^n by the algorithms NTB, STB, NBT, and SBT are $5n - 1$, $5n - 1$, $11 \cdot 2^{n-1} - 5$, and $6n$ respectively.

About all we can conclude from these grammars is that (1) the NBT algorithm can be enormously less efficient than its selective counterpart, being sometimes of exponential as opposed to linear order, and (2) the SBT algorithm required a number of instructions linearly proportional to the length of the sentence recognized in all cases, whereas the number of instructions required by the STB algorithm was proportional to the square of the sentence length for the left-branching grammar.

We consider next a slightly more complicated grammar:

$$\begin{array}{ll} S \to AB & B \to bc \\ A \to Ab & B \to Bd \\ B \to bB & A \to a \end{array}$$

It can be shown that this grammar assigns $(1/2)n(n + 1)$ different structural descriptions to the string ab^ncd. The number of instructions required to accept this string is $(\frac{1}{2})(2^{n+6} + 3n^2 - 7n - 38)$ for the NTB algorithm, $21 \cdot 2^{n+2} - 4n^2 - 37n - 79$ for the NBT algorithm, $2^{n+5} - 11n - 27$ for the STB algorithm, and $(\frac{1}{3})(2n^3 + 21n^2 + 46n + 15)$ for the SBT algorithm. We note that these figures are of exponential order except for the SBT figure which is merely a cubic polynomial.

The closed form expressions we obtained above were for classes of sentences characterized by a single parameter. For more interesting grammars it is difficult to find parameters that characterize sentences in a meaningful way. Sentence length is, of course, one possibility, but bounds on the computation required by our TM procedures as a function of sentence length are extremely difficult to obtain. Consequently, the rest of our empirical results are in the form of raw comparative data for particular grammars and sentences.

As previously mentioned, the UNIVAC M-460 program simulated the MPDA pushdown-store automaton of the authors' technical memorandum rather than the TM.

Our tables give figures for algorithms expressed as equivalent MPDA programs, but the corresponding TM figures do not differ appreciably. The number of TM instructions required in NTB, STB, NBT and SBT recognition of a sentence, for example, is just the corresponding number of MPDA instructions minus one more than the number of structural descriptions of that sentence.

A normal grammar equivalent to the previous grammar is:

$S \rightarrow AB$	
$B \rightarrow CB$	$A \rightarrow a$
$B \rightarrow BE$	$C \rightarrow b$
$A \rightarrow AC$	$F \rightarrow c$
$B \rightarrow CF$	$E \rightarrow d$

Data relevant to this grammar on recognition efficiency for the STB, SBT, NDS and SDS algorithms are included in the AFCRL technical memorandum on which this paper is based. Basically, the SBT algorithm appeared to require about three times as many TM instructions as the SDS algorithm for all sentences recognized, while the STB algorithm was significantly less efficient than the SBT algorithm for some sentences, as we have already seen for the equivalent non-normal grammar.

Another grammar studied was a Polish prefix grammar for the propositional calculus, which produced deterministic STB and SBT TM programs. It was found that the SBT recognizer required slightly more than twice as many TM instructions as its STB counterpart. As might be expected of deterministic recognizers, both were very fast relative to all of the nondeterministic recognizers considered.

Another propositional calculus grammar considered is given by

$F \rightarrow C$	$L \rightarrow L'$
$F \rightarrow S$	$L \rightarrow p$
$F \rightarrow P$	$L \rightarrow q$
$F \rightarrow U$	$L \rightarrow r$
$C \rightarrow U \supset U$	$S \rightarrow U \vee S$
$U \rightarrow (F)$	$S \rightarrow U \vee U$
$U \rightarrow \sim U$	$P \rightarrow U \wedge P$
$U \rightarrow L$	$P \rightarrow U \wedge U$

where the designated symbol is F. We will refer to this grammar as Grammar 4, and the grammars resulting from applying equivalent grammar generating procedures $P2$ and $P1$ to Grammar 4 will be called Grammar 5 and Grammar 6 respectively. Grammar 5 has the same productions as Grammar 4 except that in place of the last four rules it has:

$S \rightarrow U \wedge J$	$P \rightarrow U \vee H$
$J \rightarrow U$	$H \rightarrow U$
$J \rightarrow S$	$H \rightarrow P$

Grammar 6 also has all but the last four productions of Grammar 4, and in addition the productions:

$S \rightarrow TU$	$P \rightarrow MU$
$S \rightarrow TS$	$P \rightarrow MP$
$T \rightarrow U\vee$	$M \rightarrow U\wedge$

Grammar 7 is a standard form grammar equivalent to Grammar 4. We consider it in order to compare the PA algorithm with the STB and SBT algorithms.

Table 1 gives results compatible with our previous observations on procedures $P1$ and $P2$. Procedure $P2$ increases the efficiency of both SBT and STB recognition and procedure $P1$ makes SBT recognition more efficient but STB recognition less efficient. Procedure $P2$ is more efficient than $P1$ for SBT recognition of the sentences of Table 1. For all of the Grammars of Table 1 SBT recognition is decidedly faster than STB recognition, the ratio of their speeds increasing with sentence size. It is also faster than PA recognition for all of these grammars. Ranked in decreasing order the relative efficiencies of the various procedures with respect to these grammars and sentences are given by the list: SBT with equivalent grammar procedure $P2$; SBT with procedure $P1$; SBT, PA, STB with procedure $P2$; STB and STB with procedure $P1$.

In order to determine the extent to which the disparity in efficiency between the STB and the SBT procedures was due to the inclusion of the left-branching rule $L \rightarrow L'$, this rule was removed and a set of sentences not containing primes was recognized. Although the STB-SBT disparity was found to be slightly reduced, the results were substantially the same.

The next grammar we consider is a non-self-embedding grammar given as an example by Chomsky [1]. The grammar itself and data relevant to it are given in Table 2. In addition to the STB and SBT algorithms we have included Chomsky's finite state recognition procedure (FS), the NDS procedure, and its selective version SDS. The data show that for this grammar the procedures are ranked in order of decreasing efficiency as follows: SDS, FS, SBT, STB, NDS.

The next two grammars we consider are programming language grammars for Lisp and Algol. The former grammar is a slightly simplified (obvious ambiguities were removed) version of a previously given Lisp S-expression grammar [31]. The latter is essentially the official Algol 62 grammar [32]. (Declarations were not included and multilettered primitives were assigned structure in the manner of identifiers.)

In Table 3 are displayed the results for a STB-SBT recognition comparison of a few Lisp function definitions. Recognition of the simple function (FUNCT, (LAMBDA, (X), (CAR, (CDR, X))) required a sequence of 156,543 STB TM instructions taking more than 22 minutes as compared with figures of 172 instructions and 1.3 seconds for SBT recognition. The excessive machine time required by STB recognition precluded consideration of larger functions. One source of difficulty for STB recognition is the frequent occurrence of identifiers constructed by a left-branching rule. For example, complicating the function (A, (LAMBDA, (X), (B, X))) by replacing A with FUNCT increased the number of TM instructions required for recognition by a factor of almost ten.

A few comments are also called for on the speed of

Lisp recognition. Even making use of the SBT algorithm, the times required for long or even moderately long Lisp functions are enormously greater than similar times reported for "syntax-directed" compilers such as those of Irons and Bastian. There are three possible explanations we can suggest for this discrepancy.

(1) Termination of the recognition procedure upon determination of one structural description may save an appreciable amount of time.

(2) Use of the embedding assumption expedient previously discussed may save much time by avoiding a substantial number of blocked sequences.

(3) Our M-460 program may be significantly less efficient than the programs of other investigators. Machine time comparisons always raise this question of programming efficiency.

Empirical results relevant to (1) indicate that although this can be a significant factor in the recognition of very ambiguous sentences, the number of instructions required

TABLE 1

Sentence	Grammar 7	Grammar 4		Grammar 5 (*P2*)		Grammar 6 (*P1*)	
	PA	STB	SBT	STB	SBT	STB	SBT
p	14	15	18	13	16	15	16
$(p \wedge q)$	89	136	56	88	46	144	54
$(p' \wedge q) \vee r \vee p \vee q'$	232	1591	185	789	165	1637	186
$p \supset ((q \supset \sim(r' \vee (p \wedge q))) \supset (q' \vee r))$	712	7019	277	3037	205	7199	267
$\sim(\sim(p' \wedge (q \vee r) \wedge p'))$	1955	7738	223	2697	158	8242	218
$((p \wedge q) \vee (q \wedge r) \vee (r \wedge p')) \supset \sim((p' \vee q') \wedge (r' \vee p))$	2040		562	16157	334		553

TABLE 2

	Sentence	STB	SBT	FS	NDS	SDS
1.	$adbcddb$	43	43	26	77	19
2.	$ad^3bcbcd^3bcd^4b$	233	111	63	1033	48
3.	$adbcd^2bcd^5bcd^3b$	260	117	66	2127	51
4.	$ad^{18}b$	323	120	63	101	60
5.	$a(bc)^3d^3(bcd)^2dbcd^4b$	374	150	87	5856	60
6.	$a(bcd)^2dbcd^3bcb$	177	100	59	1088	40

$S \rightarrow AB$	$A \rightarrow a$
$A \rightarrow SC$	$B \rightarrow b$
$B \rightarrow DB$	$C \rightarrow c$
	$D \rightarrow d$

TABLE 3

Lisp Functions	STB	SBT
(FUNCT, (LAMBDA, (X), (CAR (CDR, X))))	156543	172
(MEMBER, (LAMBDA, (A, X), (COND, ((NULL, X), F), ((EQ, A, (CAR, X)), T), (T, (MEMBER, A, (CDR, X))))))		506
(LENGTH, (LAMBDA, (L), (COND, ((NULL, (CDR, I)), L), (T, (SUM, 1, (LENGTH, (CDR, L)))))))		450
(UNION, (LAMBDA, (X, Y), (COND, ((NULL, X), Y), ((MEMBER, (CAR, X), Y), (UNION, (CDR, X), Y)), (T, (CONS, (CAR, X), (UNION, (CDR, X), Y))))))		752
(AJORKJ6CMAJOR, (LAMBDA, (M), (COND, ((NULL, M), F), ((AND, (MAJORSUIT, (CAR, M)), (GREATERP, (LENGTH, (CAR, M)), 5), (OR, (AND, (EQUAL, (CAAAR, M), (QUOTE, A)), (EQUAL, (PTSINSUIT, (CAR, M)), 5)), (AND, (EQUAL, (CAAAR, M), (QUOTE, K)), EQUAL, (PTSINSUIT, (CAR, M)), 4)))), T), (T, (AJORKJ6CMAJOR, (CDR, M))))))		2047
(A, (LAMBDA, (X), (B, X)))	395	77
(FUNCT, (LAMBDA, (X), (B, X)))	3747	

TABLE 4

	STB	SBT
begin s := 0 **end**	299	92
begin s := a **end**	405	160
begin go to *next* **end**	571	116
begin 17: s = 0; **go to** 17 **end**		174
begin s := a−0.78 **end**	16466	302
begin go to if T **then step** 2 **else if** F **then step** 4 **else step** 6 **end**		429
begin s := 0 **for** k := 1 **step** 1 **until** n **do** s := $s + a[k, k]$ **end**		650
begin s := 0; **if** $g \equiv \neg a \wedge b \wedge \neg cdef$ **then** s := 1 **end**		660
begin *step* := **if** $0 \leqq m \wedge m \leqq 1$ **then** 1 **else** 0 **end**		706
begin t := **if** $x > 0$ **then** $b \times c$ **else if** $x/b < c$ **then** $s + 3 \times m/a$ **else** 0 **end**		1138
begin *next*: *sum* := *sum* + ds/d; t := $t + 1$; **if** $t >$ *tem* **then go to** *next* **end**		1236
begin y := 0; **for** p := 1 **step** 1 **until** n **do for** q := 1 **step** 1 **until** m **do if** *abs*$(a[p,q]) > y$ **then begin** y := *abs*$(a[p,q])$; i := p; k := q **end end**		2972
begin i := n := t := 0; $m[0]$:= *fct*(0); *sum* := $m[0]/2$; *nextterm*: i := $i + 1$; *mn* := *fct*(i); **for** k := 0 **step** 1 **until** n **do begin** *mp* := $(mn+m[k])/2$; $m[k]$:= *mn*; *mn* := *mp* **end**; **if** $(abs(mn)<abs(m[n])) \wedge (n<15)$ **then begin** *ds* := $mn/2$; n := $n + 1$; $m[n]$:= *mn* **end else** *ds* := *mn*; *sum* := *sum* + *ds*; **if** *abs*(*ds*) < *eps* **then** t := $t + 1$ **else** t := 0; **if** $t <$ *tim* **then go to** *nextterm* **end**		200407

TABLE 5

Sentence	PA	SBT	No SD
ededea	35	52	2
ededeabbbb	75	92	2
ededeabbbbbbbbbb	99	152	2
$ededeab^{200}$	859	2052	2
edededed eabb	617	526	14
ededededededeabb	24352	16336	429
ededededededededeabb	86139	54660	1430

to preclude other structural descriptions when only a single description exists is about the same as the number of instructions required to find that structural description. Factors (2) and (3) have not yet been investigated.

The ALGOL data are displayed in Table 4. The results are much the same as for LISP, and the same comments are appropriate. One unexpected result of recognizing the trivial "programs" of Table 4 was that it was observed that $S := A$ is ambiguous. The programmer would not be concerned about the two structural descriptions because semantic considerations (A is declared as either **Boolean, real,** or **integer**) would resolve the syntactic ambiguity, which is, nevertheless, a real one. However, when the article by Floyd [27] was examined to see how he avoided this ambiguity it was discovered that his precedence grammar is also ambiguous. This example shows clearly that precedence grammars are not unambiguous in the usual sense. The difficulty seems to rest in the filling in of the skeletal structural description, a step which Floyd points out is finite but fails to note may be ambiguous.

In Table 5 we display the results of recognizing the illustrative grammars given by Greibach [19]. SBT recognition was performed on the grammar

$$\begin{array}{ll} X \rightarrow A & Y \rightarrow e \\ X \rightarrow Xb & Y \rightarrow YdY \\ X \rightarrow Ya & \end{array}$$

and PA recognition was performed on the equivalent standard form grammar

$$\begin{array}{ll} X \rightarrow a & A \rightarrow a \\ X \rightarrow eA & B \rightarrow bB \\ X \rightarrow eDA & D \rightarrow dY \\ X \rightarrow aB & D \rightarrow dYD \\ X \rightarrow eAB & Y \rightarrow e \\ X \leftarrow eDAB & Y \rightarrow eD \\ B \rightarrow b & \end{array}$$

7. Conclusions

In this paper we have defined a number of recognition procedures for context-free grammars by giving algorithms for constructing equivalent nondeterministic Turing machine programs. Our chief concern has been to compare procedures suggested by but not identical to a procedure of Matthews and that used in the "syntax-directed" compiler of Irons. In this comparison we found our SBT procedure to be enormously more efficient than our STB procedure for the LISP and ALGOL programming language grammars considered, and generally superior for all other grammars considered except those for which the recognizers were deterministic.

In addition, we considered several other recognition procedures less thoroughly and found the most promising to be a selective version of a procedure suggested by Greibach and others. Plans to program the generation of equivalent TM programs corresponding to this recognition procedure and also to program Greibach's procedure for obtaining an equivalent standard form grammar have been at least temporarily replaced by other investigations.

A theoretical treatment of the equivalence questions raised in this paper has been given in [38].

Acknowledgments. The authors are pleased to acknowledge helpful discussions with N. Chomsky, S. Greibach and S. Kuno, and programming assistance from R. Salzman, G. Curtis, M. Heavey and A. Naro.

RECEIVED JANUARY, 1964; REVISED JANUARY, 1965

REFERENCES

1. CHOMSKY, N. Formal properties of grammars. In R. R. Bush, E. H. Galanter and R. D. Luce (Eds.), *Handbook of Mathematical Psychology, Vol. 2,* Ch. 12, Wiley, New York, 1962.
2. BACKUS, J. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. Proc. Int. Conf. Inf. Proc., UNESCO, Paris, June, 1959.
3. EVEY, J. The theory and application of pushdown store machines. Mathematical Linguistics and Automatic Translation, Rep. No. NSF-10, Harvard Comp. Lab., Cambridge, 1963.
4. GROSS, M. On the equivalence of models of language used in the fields of machine translation and information retrieval. Unpublished report, 1962.
5. SCHMIDT, L. Implementation of a symbol manipulator for heuristic translation. Paper, 1963 ACM Natl. Conf., Denver, Colo.
6. METCALFE, H. A parameterized compiler based on mechanical linguistics. 1963, ACM Natl. Conf., Denver, Colo.
7. CHOMSKY, N. On certain formal properties of grammars. *Inform. Contr. 2,* 2 (1959).
8. KUNO, S., AND OETTINGER, A. Multiple path syntactic analyzer. *Information Processing 62,* North-Holland, Amsterdam, 1963.
9. SHERRY, M. Syntactic analysis in automatic translation. AFCRL-TR-61-100, Air Force Cambridge Research Labs., Bedford, Mass., 1961.
10. IRONS, E. A syntax directed compiler for ALGOL 60. *Comm. ACM 4* (Jan. 1961), 51.
11. ——. Maintenance manual for psycho—part one. The Princeton Syntax Compiler, IDA Rep., Princeton, Jan. 1961.
12. WARSHALL, S. A syntax-directed generator. Proc. 1961 Eastern Joint Comput. Conf., MacMillan Co., New York, 1961.
13. BASTIAN, L. A phrase-structure language translator. AFCRL Rep. 62-549, AF Cambridge Research Labs., Bedford, Aug. 1962.
14. WILLETT, H., AND HELWIG, F. Syntax processor. MITRE Working Pap. W-4677, Bedford, Jan. 1962.
15. INGERMAN, P. A translation technique for languages whose syntax is expressible in extended Backus normal form. Proc. of the Int. Symp. on Symbolic Languages in Data Processing, Rome, Italy, Apr. 1962, 741–758.
16. BAKER, J. J. A note on multiplying Boolean matrices. *Comm. ACM 5* (Feb. 1962), 102.
17. WARSHALL, S. A theorem on Boolean matrices. *J. ACM 9* (Jan. 1962), 11.
18. PLATH, W. Mathematical linguistics. In *Trends in European and American Linguistics 1930–1960,* Spectrum Publ., Utrecht, Neth., 1961, 21–57.
19. GREIBACH, S. Inverses of phrase structure generators. Mathematical Linguistics and Automatic Translation, Rep. No. NSF-11, Harvard Comput. Lab.
20. GLENNIE, A. On the syntax machine and the construction of a universal compiler. Tech. Rep. No. 2, Contr. NR 049-141, Carnegie Inst. of Tech., Pittsburgh, July 1960.

21. Brooker, R., and Morris, D. An assembly program for a phrase structure language. *Comput. J. 3* (Oct. 1960).
22. ——, and ——. A general translation program for phrase structure languages. *J. ACM 9* (Jan. 1962), 1.
23. ——. Trees and routines. *Comput. J. 5*, 1 (Apr. 1962).
24. Ledley, R., and Wilson, J. Automatic-programming language translation through syntactic analysis. *Comm. ACM 5* (Mar. 62), 145.
25. Robinson, J. Preliminary codes and rules for the automatic parsing of English. RAND RM 3339, Santa Monica, Dec. 1962.
26. Ginsberg, S., and Rice, H. Two families of languages related to ALGOL. *J. ACM 9* (July 1962), 350.
27. Floyd, R. Syntactic analysis and operator precedence. *J. ACM 10* (July 1963), 316.
28. Gilbert, P., Hosler, J., and Schager, C. Automatic programming techniques. TDR-62-632, Rome Air Dev. Ctr., Rome, N. Y., Dec. 1962.
29. Barnett, M. Syntactic analysis by digital computer. *Comm. ACM 5*, 10 (Oct. 1962), 515.
30. Ross, D. On the algorithmic theory of language. ESL-TM-156, Elect. Systems Lab., MIT, Cambridge, Mass., Nov. 1962.
31. Bastian, L., Foley, J., and Petrick, S. On the implementation and usage of a language for contract bridge bidding. Proc. of the Int. Symp. on Symbolic Languages in Data Processing, Rome, Italy, Apr. 1962, 741–758.
32. Naur, P. (Ed.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM 4* (Jan. 1963), 1.
33. Matthews, H. Discontinuity and asymmetry in phrase structure grammars. *Inform. Contr. 6*, 2 (June 1963), 137–146.
34. Kuno, S., and Oettinger, A. Syntactic structure and ambiguity of English. Proc. 1963 Fall Joint Comput. Conf., Spartan Books, Baltimore, Md., 1963.
35. ——. New techniques for repetitive path elimination. Mathematical Linguistics and Automatic Translation, Sec. XI, Rep. No. NSF-13, Harvard Comput. Lab., Cambridge, Apr. 1964.
36. Abbott, R. Right-to-left parsing using a predictive grammar. Mathematical Linguistics and Automatic Translation, Sec. VIII, Rep. No. NSF-13, Harvard Comput. Lab., Cambridge, Apr. 1964.
37. Kay, M. A general procedure for rewriting strings. Paper, 1964 Ann. Mtg., Assoc. for Machine Translation and Computational Linguistics, U. of Indiana, Bloomington, 1964.
38. Griffiths, T. Turing machine recognizers for general rewriting systems. Proc. IEEE Symp. Switching Circuit Theory and Logical Design, Princeton, Nov. 1964, 47–56.

BLNSYS—A 1401 Operating System with Braille Capabilities

J. B. Landwehr, C. McLaughlin, H. Mueller, M. Lichstein and S. V. Pollack
University of Cincinnati, Cincinnati, Ohio*

BLNSYS is an operating system designed for a 4K 1401 with common optional features and two attached tape drives. Printed output of this system or of executing programs may be in either English or braille. Even though this system was written for a small machine with minimal peripheral equipment, jobs may be batched, so that card handling and lost processing time is at a minimum. This system will perform any or all of the following users specified functions: assemble SPS source decks, post list, produce condensed or uncondensed object decks, execute user's program, list card input to a program, list punched output, provide a storage dump, execute a program submitted for execution as an uncondensed object deck under debugging trace control, card-to-braille conversion, brailled listings of 7040 IBSYS batch output, and update or duplicate the system tape itself. Input-output subroutines are also included in the system.

Introduction

Previous discussions about the suitability of blind people for work in computer programming have pointed out that the production of a readable braille on the conventional high-speed line printer was a pivotal step in the formulation of techniques necessary for the successful training and performance of such personnel [1, 2]. Initial experience with several trainees indicated that the blind programmer equipped with routines which print input, output and/or memory contents in braille is no more dependent on external help than is his sighted colleague. It was also seen that, for training larger groups, it would be desirable to incorporate these braille routines into a generalized operating structure which would allow the processing of a succession of jobs to proceed smoothly and efficiently, with a minimum of intervention. The resulting system (called BLNSYS for Blind Systems and for other obvious reasons) written for the 1401, turned out to be a very effective means for handling general 1401 processing as well as those jobs prepared by blind programmers requesting braille output.

General Description

In its present form BLNSYS requires a 4K 1401 with two attached tape drives (additional drives may be used but are not necessary). (The attempt was made to define the system for the smallest possible machine configuration so as to give it the widest possible usefulness.) The machine must also be equipped with multiply-divide, extended programming, high-low-equal compare, column binary and sense switches. Overall operation is supervised by a monitor so that successive jobs are handled automatically. Input, output, exit, dump and various diagnostic routines

Work reported here was made possible by NIH Facility Research Support Grant FR 00010 and by a grant from VRA RD-1485-S-65.

* Medical Computing Center.