# GRAMMARS WITH MACRO-LIKE PRODUCTIONS

Michael J. Fischer
Harvard University
Cambridge, Mass.

## Summary

Two new classes of grammars based on programming macros are studied. Both involve appending arguments to the intermediate symbols of a context-free grammar. They differ only in the order in which nested terms may be expanded: IO is expansion from the inside-out; OI from the outside-in. Both classes, in common with the context-free, have decidable emptiness and derivation problems, and both are closed under the operations of union, concatenation, Kleene closure (star), reversal, intersection with a regular set, and arbitrary homomorphism. OI languages are also closed under inverse homomorphism while IO languages are not. We exhibit two languages, one of which is IO but not OI and the other OI but not IO, showing that neither class contains the other. However, both trivially contain the class of context-free languages, and both are contained in the class of context-sensitive languages. Finally, the class of OI languages is identical to the class of indexed languages studied by Aho [1], and indeed many of the above theorems about OI languages follow directly from the equivalence.

## 1. Introduction

Many features of programming languages can not be adequately described by context-free grammars. Often these have to do with the inability of a context-free grammar to make several identical copies of a string. We generalize context-free grammars in a way which makes copying a primitive operation. The new grammars generate a wider class of languages than the context-free, but they retain many of the decidability and closure properties of the context-free languages. In particular, they have a decidable emptiness problem, a pre-requisite, we feel, for any class of grammars which purports to be a good model for describing programming languages.

In the presentation which follows, we state the main theorems and attempt intuitive sketches of the proofs for many of them. They are not, however, to be regarded as complete proofs; in many cases, important details have been glossed over in the attempt to briefly state the main ideas. The complete proofs may be found in [4].

## 2. Intuitive Definitions and Examples

We generalize the rewriting rules of context-free grammars by borrowing from programming the idea of a macro [10, 13, 14, 17]*. A macro has a name, a list of dummy arguments, and a body or skeleton which may contain occurrences of the dummy argument symbols, other macro names, and terminal characters. A macro is "expanded" by replacing an occurrence of its name and any arguments with the corresponding skeleton into which the arguments have been substituted for the dummy symbols. A context-free rule may be considered to be a macro with no arguments.

We may form a grammar by taking a collection of macros and allowing, unlike macro programming systems, more than one "definition" for each macro name. For example, consider the rules

$$S \rightarrow F(a, b, c)$$
$$F(x, y, z) \rightarrow F(xa, yb, zc)$$
$$F(x, y, z) \rightarrow xyz.$$

Starting from the sentence symbol S, we may rewrite it using the first rule to get F(a, b, c). Applying the second rule to this yields F(aa, bb, cc), for the dummy variables x, y, z take on the values a, b, c, respectively. Applying the second rule again yields F(aaa, bbb, ccc), for this time x=aa, y=bb, and z=cc. Finally, applying the third rule gives aaabbbccc. We thus see that this grammar generates the language

$$\{a^n b^n c^n \mid n \geq 1\},$$

which is not context-free.

We note that the applicability of a particular rule to a sentential form depends only on the function symbol (i.e., macro name or non-terminal symbol) and not on the particular

---

*The original idea for using macros as the basis for a grammar and the proof that they properly contain the context-free was contained in an unpublished paper by J.S. James [10].

arguments present.* In this sense, macro grammars are also independent of context or "context-free." They gain their additional power from being able to make copies of strings (awkward even with context-sensitive grammars) and from being able to insert symbols at more than one point in a string at a time. A context-free grammar can simultaneously build on two strings, the one to the left and the one to the right of the symbol being transformed. It cannot do so on more than two strings. This is intuitively why a context-free grammar can generate the language $\{a^n b^n \mid n \geq 1\}$ but cannot generate the analogous language $\{a^n b^n c^n \mid n \geq 1\}$. It is clear that a macro grammar can simultaneously build on any number of strings.

Example 2.1 (Squares). $\{a^{n^2} \mid n \geq 1\}$ is generated by

$$S \rightarrow F(a, aaa)$$
$$F(x,y) \rightarrow F(xy,yaa) \mid x$$

where the vertical bar is the alternative symbol of BNF used to abbreviate the two rules

$$F(x,y) \rightarrow F(xy,yaa)$$
$$F(x,y) \rightarrow x.$$

This grammar uses the fact that the $n^{th}$ square is the sum of the first n odd numbers. It keeps $a^{n^2}$ in the first argument position and $a^{2n+1}$ in the second. Below is a derivation of $a^{4^2}$ (i.e., a string of 16 a's):

$$S$$
$$F(a,a^3)$$
$$F(a^4,a^5)$$
$$F(a^9,a^7)$$
$$F(a^{16},a^9)$$
$$a^{16}.$$

Example 2.2 (Non-primes). $\{a^n \mid n$ is a non-prime $\geq 2\}$ is generated by

$$S \rightarrow F(aa)$$
$$F(x) \rightarrow F(xa) \mid G(x)$$
$$G(x) \rightarrow xG(x) \mid xx.$$

This grammar produces a string of two or more a's as the argument to G and then makes two or more copies of that string, resulting in a string whose length is the product of two numbers $\geq 2$. To generate $a^{12}$, we may make 3 copies of $a^4$:

| | |
|---|---|
| $S$ | $G(a^4)$ |
| $F(aa)$ | $a^4G(a^4)$ |
| $F(aaa)$ | $a^8G(a^4)$ |
| $F(a^4)$ | $a^{12}$. |

Similarly, we can generate products:

$$\{a^n c a^m c a^k \mid n,m,k \geq 1 \text{ and } n = m \cdot k\};$$

powers of 2:

$$\{a^n c a^m \mid n,m \geq 1 \text{ and } n = 2^m\};$$

and the set of non-squares.

Example 2.3 (Binary Representations). $\{\bar{n}ca^n \mid \bar{n} \in \{0,1\}^*$ and n is the non-negative integer represented by the binary numeral $\bar{n}\}$:

$$S \rightarrow F(e,e) \quad (e \text{ is the empty string})$$
$$F(x,y) \rightarrow F(x0,yy) \mid F(x1,yy1) \mid xcy.$$

None of the examples given so far has used nested macros, that is, one macro name contained within the argument list of another. It is not obvious exactly how to define derivation on nested terms. One might choose the conventional rules for function evaluation in which the innermost macro must be expanded to a terminal string before any outer macros may be expanded. We will call this convention expansion from the inside-out (IO). On the other hand, one may expand from the outside-in (OI), taking the arguments in their unexpanded state. For example, given the macros

$$F(x) \rightarrow xx$$
$$G(x) \rightarrow xa \mid xb$$

and the sentential form F(G(a)), under inside-out rules of evaluation, we could not apply the first rule to F until G(a) was expanded to one of the terminal strings aa or ab. Under outside-in evaluation, we must expand F first to give G(a)G(a). Now, each of the G's may independently give rise to aa or ab, so we could get any of the four strings aaaa, aaab, abaa, abab, whereas with inside-out expansion, we can get only aaaa and abab.

Another reasonable rule of expansion is to simply put no restriction upon the order in which terms are expanded. It can be shown that this unrestricted evaluation is equivalent to

---

* In some of the models we will define later, the arguments may determine whether or not the function symbol can be rewritten at all, but if it is legal to rewrite the function symbol, the arguments play no part in the selection of the particular rule to be applied.

outside-in expansion in the sense that any terminal string derivable using the one type of expansion is also derivable using the other (although not necessarily in the same number of steps).

For a particular grammar, unrestricted expansion (or outside-in) always allows at least as many strings to be derived as does inside-out. However, the classes of languages generated are incomparable, i.e., neither is contained within the other. As a corollary, grammars with nesting can generate strictly more languages than grammars without, for inside-out and outside-in expansion are the same for non-nested terms.

Example 2.4. Let f: $\{0,1\}^* \rightarrow 1^*$ be a homomorphism defined by

$$f(0) = e$$
$$f(1) = 1.$$

Let $L_1 = \{1^{2^n} \mid n \geq 1\}$. Let $L_1' = f^{-1}(L_1) = \{w \in \{0,1\}^* \mid f(w) \in L_1\}$. Hence, a string of 0's and 1's is in $L_1'$ if the number of 1's in it is a positive power of 2.

$L_1$ is generated by a grammar without nesting:

$$S \rightarrow F(1)$$
$$F(x) \rightarrow F(xx) \mid xx.$$

$L_1'$ is generated by the OI (outside-in) grammar:

$$S \rightarrow F(A)$$
$$F(x) \rightarrow F(xx) \mid xx$$
$$A \rightarrow OA \mid AO \mid 1.$$

We note that this grammar first produces the string $A^{2^n}$ for some $n \geq 1$. Then each A generates some string in $0^* 10^*$. We prove later that $L_1'$ cannot be generated by any inside-out grammar.

Example 2.5 (Equal Length Substrings). $\{x_1 c x_2 c \ldots c x_n \mid n,m \geq 1$ and for $1 \leq i \leq n$, $x_i \in \{0,1\}^*$ and $\ell(x_i) = m\}$.

This is generated by the OI grammar:

$$S \rightarrow F(A)$$
$$F(x) \rightarrow F(xA) \mid G(x)$$
$$G(x) \rightarrow xcG(x) \mid x$$
$$A \rightarrow 0 \mid 1.$$

Example 2.6 (Retrieval Language). $\{x_1 c x_2 c \ldots c x_n d y_1 c y_2 c \ldots c y_m \mid m,n \geq 1$, for $1 \leq i \leq n$, $x_i \in \{0,1\}^*$ and for each j, $1 \leq j \leq m$ there exists an i, $1 \leq i \leq n$ such that $y_j = x_i\}$.

In this language, the x's are analogous to the declarations in the block heading of a block-structured programming language and the y's to the body of the block. The language then requires that every variable (each y) used in the body must be declared at least once in the head (must be equal to some x). This language is generated by the following rather complicated OI grammar:

$$S \rightarrow F(e,D)$$
$$F(x,y) \rightarrow F(x0,y) \mid F(x1,y) \mid xcF(e,A(x,y)) \mid xdG(A(x,y))$$
$$G(x) \rightarrow xcG(x) \mid x$$
$$A(x,y) \rightarrow x \mid y.$$

The grammar first builds up the sentential form $x_1 c x_2 c \ldots c x_n d G(A(x_n, A(x_{n-1}, \ldots A(x_1, D)\ldots)))$. G then makes one or more copies of its argument. Each copy may produce any of the strings $x_n, \ldots, x_1$ or the dead symbol D. Of course, if D is generated, the derivation is blocked.

Example 2.7. Let $L_2 = \{1^m (c1^m)^{n-1} \mid m \geq 1, n = 2^m\}$. $L_2$ is generated by the IO (inside-out) grammar:

$$S \rightarrow F(1)$$
$$F(x) \rightarrow G(F(x1)) \mid G(x)$$
$$G(x) \rightarrow xcx.$$

This grammar produces first the form $G(G(G(\ldots G(1^m)\ldots)))$ containing exactly m G's for some m. The last rule is then applied repeatedly to yield the string $1^m (c1^m)^{2^m - 1}$ in L. We prove later that no OI grammar can generate $L_2$.

## 3. Formal Definitions

In this section, we give formal definitions of IO and OI grammars and of some related concepts. Although the development which follows is largely informal, this material serves to indicate how the theorems and proofs will be formalized (as indeed they have been done in [4]) as well as to define precisely some notions introduced by example in the previous section.

Macro grammars operate over structured strings called terms, defined in definition 3.1. Terms are built up from elements of a finite set $\Sigma$ of terminal symbols, a finite set $\mathcal{F}$ of function symbols, and the punctuation characters "(", ")", and ",". Associated with every $F \in \mathcal{F}$ is a number called the rank of F which is a number of arguments that F takes. The rank of F is given by the function $\rho: \mathcal{F} \rightarrow \underset{\sim}{N}$.

Definition 3.1 (Terms). The set of terms over $\Sigma$, $\mathcal{F}, \rho$ is the least set of strings over $\Sigma \cup \mathcal{F} \cup \{"(", ")", ","\}$ satisfying:

i) e is a term;
If $a \in \Sigma$, then a is a term;

ii) If $\sigma$, $\tau$ are terms, then $\sigma\tau$ is a term;

iii) If $F \in \mathcal{F}$ and $\sigma_1, \ldots, \sigma_{\rho(F)}$ are terms, then $F(\sigma_1, \ldots, \sigma_{\rho(F)})$ is a term.

The formal term F( ) can represent either a term comprising a function symbol of rank zero or a term comprising a function symbol of rank one whose argument is the empty string. Knowing $\rho(F)$, of course, resolves the ambiguity. In talking informally, we generally omit the parentheses for function symbols of rank zero. However, the informal notation $F(\xi_1, \ldots, \xi_{\rho(F)})$ does not imply that $\rho(F) > 0$.

Definition 3.2 (Subterm). Let $\sigma$ be term over $\Sigma$, $\mathcal{F}, \rho$. $\tau$ is a subterm of $\sigma$ if $\tau$ is a term over $\Sigma$, $\mathcal{F}, \rho$ and $\tau$ is a substring of $\sigma$.

Definition 3.3 (Top Level). A subterm $\tau$ of a term $\sigma$ is said to occur at the top level in $\sigma$ if there exist terms $\sigma_1$, and $\sigma_2$ such that $\sigma = \sigma_1 \tau \sigma_2$. This is equivalent to saying that $\tau$ does not appear within the argument list of some function symbol in $\sigma$.

Traditionally, a grammar is defined to be an ordered set which lists the sets of terminal and non-terminal symbols, the start symbol, and the set of productions of the grammar. The rule by which one string can be derived from another is generally defined separately and is not included as part of the grammar itself. We will have occasion to consider the same grammar structure (i.e., the same ordered set) under more than one rule or mode of derivation. Thus, when we speak of a grammar, we will mean a grammar structure together with a preferred mode of derivation.

Definition 3.4 (Macro Grammar Structure). A macro grammar structure is a 6-tuple $(\Sigma, \mathcal{F}, \mathcal{V}, \rho, S, P)$ where:

$\Sigma$ is a finite set of terminal symbols;

$\mathcal{F}$ is a finite set of non-terminal or function symbols;

$\mathcal{V}$ is a finite set of argument or variable symbols;

$\rho$ is a function from $\mathcal{F}$ into the non-negative integers ($\rho(F)$ is the number of arguments which F takes);

$S \in \mathcal{F}$ is the start or sentence symbol, $\rho(S) = 0$;

P is a finite set of productions or rules of the form $F(x_1, \ldots, x_{\rho(F)}) \rightarrow \nu$ where $F \in \mathcal{F}$, $x_1, \ldots, x_{\rho(F)}$ are distinct members of $\mathcal{V}$, and $\nu$ is a term over $\Sigma \cup \{x_1, \ldots, x_{\rho(F)}\}, \mathcal{F}, \rho$.

A grammar structure may be used to derive one term from another according to a particular set of conventions called the mode of derivation. In the general or unrestricted mode, one string can be derived from another by applying a production to a function symbol, causing the function symbol and its associated arguments to be replaced by the right-hand side of the production into which the arguments have been substituted for the corresponding variables. We also consider two restrictions on the above scheme of derivation: inside-out (IO) and outside-in (OI). In the inside-out mode, a function symbol may be rewritten only if its arguments are all terminal strings. In the outside-in mode, a function symbol is subject to rewriting only if it occurs at the top level.

Definition 3.5 (Direct Derivation). Let $\sigma$, $\tau$ be terms over $\Sigma', \mathcal{F}, \rho$, $\underset{\sim}{G} = (\Sigma, \mathcal{F}, \mathcal{V}, \rho, S, P)$ a grammar structure, and $\Sigma' \supset \Sigma$, $\Sigma' \cap \mathcal{F} = \phi$. $\sigma \underset{\text{unr}, \underset{\sim}{G}}{\Longrightarrow} \tau$, read $\sigma$ directly produces or directly generates $\tau$ by an unrestricted step of the grammar $\underset{\sim}{G}$, in case $\sigma$ contains a subterm of the form $F(\xi_1, \ldots, \xi_n)$ where $F \in \mathcal{F}$ and $\xi_1, \ldots, \xi_n$ are terms over $\Sigma', \mathcal{F}, \rho$, P contains a rule $F(x_1, \ldots, x_n) \rightarrow \nu$, and $\tau$ results from $\sigma$ by replacing a single occurrence of $F(\xi_1, \ldots, \xi_n)$ by $\nu'$, where $\nu'$ is obtained from $\nu$ by substituting the terms $\xi_1, \ldots, \xi_n$ for corresponding occurrences of the variables $x_1, \ldots, x_n$ in $\nu$. We say that $\sigma \underset{\text{unr}, \underset{\sim}{G}}{\Longrightarrow} \tau$ via the rule $F(x_1, \ldots, x_n) \rightarrow \nu$.

$\sigma \underset{\text{IO}, \underset{\sim}{G}}{\Longrightarrow} \tau$, read $\sigma$ directly produces or directly generates $\tau$ by an inside-out step of the grammar $\underset{\sim}{G}$, in case $\sigma \underset{\text{unr}, \underset{\sim}{G}}{\Longrightarrow} \tau$ and all of the arguments to the rewritten function symbol are terminal strings.

$\sigma \underset{OI, \underset{\sim}{G}}{\Longrightarrow} \tau$, read $\sigma$ directly produces or directly generates $\tau$ by an outside-in step of the grammar $\underset{\sim}{G}$, in case $\sigma \underset{unr, \underset{\sim}{G}}{\Longrightarrow} \tau$ and the subterm of $\sigma$ which is rewritten occurs at the top level of $\sigma$.

Definition 3.6 (Derivation). Let $\mathcal{M}$ be one of the three modes unr, OI, or IO, $\underset{\sim}{G}$ a macro grammar structure, and p a non-negative integer, $\sigma \overset{p}{\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}} \tau$, read $\sigma$ produces or generates $\tau$ in exactly p steps, or $\tau$ is derivable from $\sigma$ in exactly p steps, if there exists a sequence of $\sigma_0, \sigma_1, \ldots, \sigma_p$ such that $\sigma = \sigma_0$, $\tau = \sigma_p$, and $\sigma_i \underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow} \sigma_{i+1}$ for $0 \leq i < p$. In particular, $\sigma \overset{0}{\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}} \tau$ iff $\sigma = \tau$.

Finally, $\sigma \overset{*}{\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}} \tau$, read $\sigma$ produces or generates $\tau$, or $\tau$ is derivable from $\sigma$, if for some $p \geq 0$, $\sigma \overset{p}{\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}} \tau$. We note that $\overset{*}{\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}}$ is the transitive closure of the relation $\underset{\mathcal{M}, \underset{\sim}{G}}{\Longrightarrow}$.

Definition 3.7 (Macro Grammar). We finally define a macro grammar to be a macro grammar structure together with a perferred mode of derivation. The following two will be of interest.

An inside-out (IO) macro grammar is a grammar structure together with the inside-out rule of derivation.

An outside-in (OI) macro grammar is a grammar structure together with the outside-in rule of derivation.

We assume that the mode of a derivation by a grammar $\underset{\sim}{G}$ is the grammar's preferred mode unless specified otherwise. Similarly, we may omit specifying the grammar when it is understood from context. For example, if $\underset{\sim}{G}$ is an IO grammar, then $\sigma \underset{IO, \underset{\sim}{G}}{\Longrightarrow} \tau$ may be shortened to $\sigma \underset{\underset{\sim}{G}}{\Longrightarrow} \tau$, and that may be written simple $\sigma \Longrightarrow \tau$ if $\underset{\sim}{G}$ is understood.

Definition 3.8 (Sentential Form). Let $\underset{\sim}{G} = (\Sigma, \mathcal{F}, \mathcal{V}, \rho, S, P)$ be any macro grammar. A term $\sigma$ over $\Sigma, \mathcal{F}, \rho$ is a sentential form of $\underset{\sim}{G}$ if $S \overset{*}{\underset{\underset{\sim}{G}}{\Longrightarrow}} \sigma$.

Definition 3.9 (Language Generated by a Grammar). Let $\underset{\sim}{G} = (\Sigma, \mathcal{F}, \mathcal{V}, \rho, S, P)$ be a grammar and let $\sigma$ be a term over $\Sigma, \mathcal{F}, \rho$. Then

$$L(\underset{\sim}{G}) = \{w \in \Sigma^* \mid S \overset{*}{\underset{\underset{\sim}{G}}{\Longrightarrow}} w\}.$$

We call $L(\underset{\sim}{G})$ the language generated by $\underset{\sim}{G}$.

Definition 3.10 (Language Class). The class of IO (respectively OI) languages is the set of all languages generable by the family of IO (respectively OI) grammars.

Definition 3.11 (Weak Equivalence). Two grammars $\underset{\sim}{G}$ and $\underset{\sim}{G}'$ are (weakly) equivalent if $L(\underset{\sim}{G}) = L(\underset{\sim}{G}')$.

## 4. IO Grammars

We begin with some miscellaneous theorems which are needed later.

Theorem 4.1 (Closure Properties). The class of IO languages is closed under the operations of union, concatenation, Kleene closure (star), reversal, and arbitrary homomorphism.

Proof. Trivial.

We will see later that the IO languages are also closed under intersection with a regular set but are not closed under inverse homomorphism.

We now show that it is possible to eliminate "useless" rules from an IO grammar.

Definition 4.2. A grammar $\underset{\sim}{G}'$ is a subgrammar of a grammar $\underset{\sim}{G}$ if $\underset{\sim}{G}$ and $\underset{\sim}{G}'$ have the same start symbol and every rule of $\underset{\sim}{G}'$ is also a rule of $\underset{\sim}{G}$.

Definition 4.3. An IO grammar is admissible if for each rule there is some derivation of a terminal string in which that rule is used.

A consequence of the above definition is that in an admissible grammar, every function symbol appears in some sentential form and every sentential form produces some terminal string.

Theorem 4.4 (Admissibility). Given any IO grammar, we can effectively find an equivalent, admissible subgrammar.

Proof. The proof is analogous to that for context-free grammars.

Corollary 4.5 (Emptiness Problem). The emptiness problem for IO grammars is decidable, that is, given any IO grammar $\underset{\sim}{G}$, we can

effectively decide if $L(\underset{\sim}{G}) = \emptyset$.

Proof. The language generated by an admissible grammar is empty iff the grammar has no rules at all.

The next result shows that no power is lost if we prohibit rules from discarding their arguments. An argument-preserving grammar has the important property that any terminal string that appears in a sentential form must also appear as a substring of any word derived from that sentential form.

Definition 4.6. An IO grammar is argument-preserving if in each rule, every argument symbol which appears in the left part of the rule also occurs at least once in the right part of the rule.

Theorem 4.7 (Argument-Preserving). Given any IO grammar, we can effectively find an equivalent IO grammar which is argument-preserving.

Proof. The new grammar "guesses" which arguments will eventually be dropped and simply refrains from producing them in the first place. Thus, associated with each function symbol of the new grammar is information telling which arguments of the corresponding symbol of the old grammar have already been dropped.

Factored Grammars

We show next that it is possible to keep track of a certain finite amount of information about the arguments and value of a function symbol of an IO grammar. The immediate goal is to show that IO languages are closed under intersection with a regular set, but the more general factor theorem is also used later in the proofs of lemma 4.14 and theorem 4.18.

Assume we are given a congruence relation $\equiv$ over $\Sigma^*$ relative to concatenation. Let $\bullet$ be the operation on equivalence classes induced by concatenation. Assume $\equiv$ has finite index q and denote its equivalence classes by $\overline{f}_1, \ldots, \overline{f}_q$. An IO grammar $\underset{\sim}{G}$ is factored relative to $\equiv$ if for each function symbol F there exists a $(\rho(F) + 1)$ - tuple of equivalence classes $\langle f_0; f_1, \ldots, f_{\rho(F)} \rangle$ such that

(1) if $v_1, \ldots, v_{\rho(F)}$ are any terminal strings in classes $f_1, \ldots, f_{\rho(F)}$ respectively and w is any terminal string such that $F(v_1, \ldots, v_{\rho(F)}) \overset{*}{\Longrightarrow} w$, then $w \in f_0$, and

(2) for every sequence of terminal strings $v_1, \ldots, v_{\rho(F)}$, if $F(v_1, \ldots, v_{\rho(F)})$ appears in a sentential form of $\underset{\sim}{G}$, then $v_i \in f_i$, $1 \leq i \leq \rho(F)$.

Note in particular that all the strings derivable from the start symbol in a factored grammar must be in the same equivalence class.

The next theorem asserts that from any IO grammar, we can find a sequence of q factored grammars, each of which generates exactly those strings of the original language that are in the corresponding equivalence class.

Theorem 4.8 (Factor). Given a congruence relation $\equiv$ over $\Sigma^*$ of finite index q and an IO grammar $\underset{\sim}{G}$, we can find a sequence of IO grammars $\underset{\sim}{G}_1, \ldots, \underset{\sim}{G}_q$ such that for each i, $1 \leq i \leq q$, $\underset{\sim}{G}_i$ is factored relative to $\equiv$ and $L(\underset{\sim}{G}_i) = L(\underset{\sim}{G}) \cap \overline{f}_i$, where $\overline{f}_1, \ldots, \overline{f}_q$ are the equivalence classes of $\equiv$.

Proof. The new grammars $\underset{\sim}{G}_1, \ldots, \underset{\sim}{G}_q$ simulate the derivations of $\underset{\sim}{G}$, but attached to each function symbol is some information denoting which class each argument is in and which class the terminal string eventually to be produced will be in. The new grammar "guesses" that a given instance of a function symbol of $\underset{\sim}{G}$ will generate a string in a given equivalence class and eventually verifies that the guess is correct.

The construction is straightforward and is left to the reader. The proof that the construction works may be found in [4] (although the factor theorem is stated there in a slightly different form).

Corollary 4.9 (Intersection with Regular). The class of IO languages is closed under intersection with a regular set.

Proof. Let $\underset{\sim}{G}$ be IO and R be a regular set. By the Myhill theorem, [15,16] R is the union of certain equivalence classes of a congruence relation of finite index q. By theorem 4.8, we can find a sequence of IO grammars $\underset{\sim}{G}_1, \ldots, \underset{\sim}{G}_q$ factored by that congruence relation such that $L(\underset{\sim}{G}) \cap R$ is the union of certain of the $L(\underset{\sim}{G}_i)$, $1 \leq i \leq q$. The corollary then follows from the closure of the IO languages under union (theorem 4.1).

Corollary 4.10 (Derivation Problem). The derivation problem for IO grammars is decidable, that is, there exists an effective procedure which, given an IO grammar $\underset{\sim}{G}$ and a terminal string w, will determine whether or not $w \in L(\underset{\sim}{G})$.

Proof. Construct an IO grammar $\underset{\sim}{G}'$ for $L(G) \cap \{w\}$ (corollary 4.9) and test if $L(\underset{\sim}{G}') = \emptyset$ (corollary 4.5).

Corollary 4.11 (Recursiveness). Every IO language is a recursive set.

Containment in Context-Sensitive

We now turn our attention to a series of lemmas which enable us to find, given an IO grammar $G$, an equivalent (except for e) IO grammar $G'$, in which no step of a derivation can decrease the total number of terminal and function symbols in a sentential form. From this, we construct a non-deterministic linear bounded automaton (theorem 4.16) which recognizes $L(G)$, showing that the IO languages are contained in the context-sensitive languages.[11, 12]

An IO derivation step can decrease the total number of terminal and function symbols in a term in three ways: First, a subterm appearing in an argument list may simply be discarded, e.g.,

$$F(a, bcccab) \Longrightarrow aaa \text{ via } F(x,y) \rightarrow xxx.$$

Second, the term may be shortened even by an argument-preserving rule whose right side has as many symbols as the left if some of the actual arguments are empty, e.g.,

$$F(e) \Longrightarrow e \text{ via } F(x) \rightarrow xxx.$$

Third, a length-shortening rule may be applied, e.g.,

$$F(a,bc) \Longrightarrow bca \text{ via } F(x,y) \rightarrow yx.$$

An argument-preserving grammar precludes steps of the first kind. We show in lemma 4.14 that we can eliminate empty arguments and empty right parts from the rules of a grammar, forming an equivalent (except for e) e-free grammar, and this can be done in such a way as to maintain the argument-preserving property. Corollary 4.13 asserts that no empty arguments can appear in the sentential forms of such a grammar, ruling out the second kind of length-shortening. We then show in lemma 4.15 that in addition, length-shortening rules of the third type above can be eliminated while maintaining equivalence and the argument-preserving and e-free properties. We conclude that no step of the resulting grammar can decrease the number of terminal and function symbols in a sentential form.

Definition 4.12. An IO grammar is e-free if in each rule

(i) the right part is non-empty, and

(ii) the empty string is not the argument to any function symbol appearing in the right part.

Corollary 4.13. Every sentential form of an e-free grammar is an e-free term.

Lemma 4.14 (e-free). Given any IO grammar $G$, we can find an argument-preserving, e-free, IO grammar $G'$ such that $L(G') = L(G) - \{e\}$.

Proof. We may assume that $G$ is argument-preserving by theorem 4.7. Applying theorem 4.8 to the congruence relation $\equiv$ induced by the partition

$$\{\{e\}, \Sigma^+ (=\Sigma^* - \{e\})\},$$

we get a factored grammar $G_1$ such that $L(G_1) = L(G) \cap \Sigma^+ = L(G) - \{e\}$. $G_1$ will still be argument-preserving (by the construction used to prove theorem 4.8).

Let $G_2$ be an admissible subgrammar of $G_1$. We now eliminate from $G_2$ those argument positions which belong to $[e]_{\equiv}$ (the equivalence class of e), for they always will contain terms which must generate e (by admissibility and the fact that e is the only member of $[e]_{\equiv}$). Also drop from $G_2$ those function symbols (and any rules involving them) whose result will be e.

The grammar $G_3$ thus formed is argument-preserving and e-free, and $L(G_3) = L(G) - \{e\}$. Hence, take $G' = G_3$.

Lemma 4.15 (Length Non-Decreasing). Given an e-free, argument-preserving IO grammar $G$, we can find an equivalent e-free, argument-preserving IO grammar $G'$ such that in every rule, the number of terminal, function, and argument symbols on the right side is at least as large as the number on the left.

Theorem 4.16 (Containment in Context-Sensitive). The class of IO languages is properly contained in the class of context-sensitive languages.

Proof. Given an IO grammar $G$, we first find an e-free, argument-preserving grammar $G'$ such that $L(G') = L(G) - \{e\}$ (lemma 4.14). From $G'$ we construct an equivalent grammar $G''$ in which the number of symbols (not including parentheses and commas) in the right part of each rule is at least as great as the number on the left, and $G''$ is also argument-preserving and e-free (lemma 4.15).

No derivation step of $G''$ can decrease the number of symbols in the sentential form (note how the argument-preserving and e-free conditions are necessary here); hence, a linear bounded automaton can simulate a derivation of $G''$ and thereby recognize $L(G'')$, so $L(G)$ is context-sensitive. The containment is proper since the IO languages are closed under erasing homomophism while the context-sensitive are not.

An OI Language which is not IO

We will prove next that an explicitly stated, intuitively simple language cannot be generated by an IO grammar. The language we present is the inverse homomorphic image of a

language which IO, giving us immediately that the IO languages are not closed under inverse homomorphism. It is also OI, so the class of IO languages does not contain the class of OI languages.

Definition 4.17. Let $L_1 = \{1^{2^n} \mid n \geq 1\}$. Let $f: (0, 1)^* \rightarrow 1^*$ be a homomorphism such that $f(0) = e$ and $f(1) = 1$. Let $L_1' = f^{-1}(L_1)$ (i.e., $L_1' = \{w \in (0,1)^* \mid f(w) \in L_1\} =$ $\{w \in (0,1)^* \mid$ the number of 1's is w is a positive power of 2$\}$).

Theorem 4.18. $L_1'$ is not an IO language.

Proof. Intuitively, two things give an IO grammar more power than a context-free grammar: First, its ability to simultaneously build on several strings, and second, its ability to copy or duplicate terminal strings using rules such as $F(x) \rightarrow xx$.

The main idea of the proof is to show that the ability to copy is of limited use in the generation of $L_1'$, so assuming that there is an IO grammar for $L_1'$, we show there exists a context-free grammar generating $L_1$, a contradiction. An interesting property of the proof is that it is not constructive, that is, we do not show how to find the context-free grammar given the IO grammar but only that such a grammar exists.

We observe first that in an argument-preserving grammar (which we will assume), no rule drops or discards any of its arguments, so if a terminal string appears in the argument position of some function symbol in a sentential form, then it must also appear in any string derived from that sentential form. Clearly, if a duplicating rule is applied to such a function symbol, then two or more copies of that terminal string will appear in any derived string.

Certain strings in $L_1'$, for example, those in $L_1'' = \{101001000 \ldots 10^{m-1}10^m \mid m = 2^n$ for some $n \geq 1\}$, have the property that any substring which appears in the string more than once contains at most a single occurrence of the symbol 1. Thus, in the derivation of such a string, no argument containing two or more occurrences of 1 could be copied, since if it were, both copies would appear in the final string. Hence, in deriving a string in $L_1''$, copying rules can be used only to copy those arguments which happen to contain at most one 1.

We thus wish to keep track of whether or not an argument contains two or more 1's so that we can tell whether a copying rule can be used in the derivation of a string in $L_1''$. The factor theorem (theorem 4.8) allows us to construct an equivalent grammar factored relative to the partition $\{0^*, 0^*10^*, 0^*10^* 1(0,1)^*\}$, of which the third class contains all and only those strings containing two or more 1's. In a factored grammar, a given argument position of a given function symbol always belongs to the same equivalence class in all sentential forms. Thus, a rule which copies an argument belonging to the third class can never be used in the derivation of any string in $L_1''$, for at the time that the rule is applied, its argument will be some string belonging to the third class, that is, containing two or more 1's. Hence, if we drop from the grammar all such rules, we can still generate all of $L_1''$. Call this grammar $\underset{\sim}{G}_1$.

Now, if we erase all the occurrences of the terminal symbol 0 in the grammar $\underset{\sim}{G}_1$, we get a grammar $\underset{\sim}{G}_2$ which generates exactly $L_1$. In addition, the new grammar is factored relative to the new partition $\{e, 1, 111^*\}$, and no rule copies an argument belonging to the third class (from the way we defined $\underset{\sim}{G}_1$).

Thus $\underset{\sim}{G}_2$ can only copy arguments which belong to the first or second classes, that is, arguments which are always e or always 1. Since they are constant, the language generated is not changed if we simply eliminate them from the grammar by substituting for them the constants e and 1, respectively, giving us a grammar for $L_1$ which contains no copying rules.

This new grammar then neither copies nor drops any arguments, so any string which appears in an argument position of a function symbol in a sentential form must eventually appear exactly once in any derived string. Since $L_1 \subseteq 1^*$, the order of the pieces of the string is immaterial, and the same language is generated if we simply eliminate all the argument symbols, parentheses, and commas from the grammar, leaving us with a context-free grammar for $L_1$. But this is impossible since $L_1$ is not a context-free language. Hence, $L_1'$ is not IO.

Theorem 4.19. The class of IO languages does not contain the class of OI languages.

Proof. By theorem 4.18, the language $L_1'$ is not IO. However, an OI grammar for $L_1'$ is exhibited in example 2.4.

Theorem 4.20 (Non-Closure under Inverse Homomorphism). The class of IO languages is not closed under the operation of inverse homomorphism.

Proof. An IO grammar for $L_1$ is given in example 2.4. Theorem 4.18 shows that $L_1' = f^{-1}(L_1)$ is not an IO language.

## 5. OI Grammars

### Equivalence with Indexed Grammars

Our first main result in this section is the equivalence between outside-in languages and the indexed languages presented by A.V. Aho last year at this conference [1]. Since the equivalence between the two types of grammars is effective, most of his results are immediately applicable to OI grammars. Similarly, our proof later that $L_2$ is not OI gives a simple example of a language which cannot be generated by an indexed grammar.

The proofs that each class contains the other involve showing first that any grammar of the one class can be put into a special form without changing the language generated and then that any grammar of that form can be simulated by a grammar of the other class.

We will not define indexed grammars here, nor will we give any of the constructions needed for the proof except to mention the standard form for OI grammars which is used in the proof that every OI language is also an indexed language.

Definition 5.1. An OI grammar is in OI standard form if every one of its rules is in one of the two forms:

(i) $F(x_1, \ldots, x_n) \rightarrow G(H_1(x_1, \ldots, x_n), \ldots, H_m(x_1, \ldots, x_n))$, $m,n \geq 0$, or

(ii) $F(x_1, \ldots, x_n) \rightarrow \eta$, where $n \geq 0$ and $\eta$ contains no function symbols.

Theorem 5.2 (OI Standard Form). Given any OI grammar, we can find an equivalent grammar in OI standard form.

We note that every sentential form of an OI standard form grammar is the concatenation of terminal symbols and terms of a special shape, called symmetric terms. A term $\sigma$ is a symmetric term if $\sigma = G(H_1(\xi_1, \ldots, \xi_n), \ldots, H_m(\xi_1, \ldots, \xi_n))$ where $m,n \geq 0$, $G, H_1, \ldots, H_m$ are function symbols, $\xi_1, \ldots, \xi_n$ are terms, and for all $i$, $1 \leq i \leq m$, $H_i(\xi_1, \ldots, \xi_n)$ is a symmetric term. The indexed grammar which simulates an OI standard form grammar takes advantage of the fact that the argument list to each of the H's is the same and represents it only once.

Theorem 5.3 (OI ≡ Indexed). The class of OI languages is identical to the class of indexed languages. Furthermore, given an OI grammar, we can effectively find an equivalent indexed grammar, and conversely, given an indexed grammar, we can effectively find an equivalent OI grammar.

We get as corollaries that many of the theorems about indexed grammars are also true of OI grammars. In particular, the following corollaries are immediate from the corresponding theorems proved by Aho about indexed grammars [1].

Corollary 5.4 (Decidable Emptiness). The emptiness problem for OI grammars is decidable, that is, given an arbitrary OI grammar $\underset{\sim}{G}$, it is decidable whether or not $L(\underset{\sim}{G}) = \emptyset$.

Corollary 5.5 (Containment in Context-Sensitive). The class of OI languages is properly contained in the class of context-sensitive languages, and given an OI grammar, we can effectively find an equivalent context-sensitive grammar.

Corollary 5.6 (Derivation Problem). It is decidable given an arbitrary OI grammar $\underset{\sim}{G}$ and a terminal string w whether or not $w \in L(\underset{\sim}{G})$.

Corollary 5.7 (Recursiveness). Every OI language is a recursive set.

Corollary 5.8 (Closure Properties). The class of OI languages is closed under the operations of union, concatenation, Kleene closure (star), reversal, arbitrary homomorphism, inverse homomorphism, and intersection with a regular set.

From this corollary, we get immediately the result:

Corollary 5.9 (Full AFL). The class of OI languages is a full abstract family of languages (full AFL). [5,8,9]

The theorems about AFL's give us in addition closure under generalized sequential machine (gms) mappings, inverse gsm mappings, quotients with a regular set, etc. (Aho showed these properties independently for indexed languages.)

Corollary 5.10 (Nested Stack Machine). The class of OI languages is exactly the class of languages accepted by one-way, non-deterministic nested stack machines.*

*A stack machine [6,7] is a generalization of a pushdown store machine [3] in which the machine at any time is allowed to scan down into the stack but cannot write in the stack when in this scanning mode. A nested stack machine is a further generalization due to Aho in which the machine may write in a restricted way while scanning within the stack. [1] It may not erase any of the old stack but it may create new stacks within the old one. However, it must destroy the new stacks before moving back up to the top of the old stack.

An IO Language which is not OI

We now give a language which is IO but not OI, showing in conjunction with theorem 4.19 that the OI and the IO languages are incomparable, that is, neither class contains the other.

The proof of the theorem requires that we be able to eliminate dead symbols from a standard form OI grammar. A function symbol is said to be dead if it cannot generate any terminal string when given terminal strings as arguments. We note that for IO grammars, theorem 4.4 asserts that every IO grammar has an admissible subgrammar and hence a subgrammar with no dead symbols. For OI grammars, this is not so, for a rule that produces a dead symbol can nevertheless be used in the derivation of a terminal string, e.g., the grammar

$$S \rightarrow F(D)$$
$$F(x) \rightarrow a$$

generates the string a and yet the dead symbol D appears in the first rule. Thus, the construction to eliminate the dead symbols is necessarily somewhat elaborate.

Lemma 5.11. Given an OI grammar $\mathcal{G}$, we can effectively find an equivalent OI grammar $\mathcal{G}''$ which contains no dead symbols.

Proof. We first eliminate from $\mathcal{G}$ any rule with a dead symbol on the top level of the right part. This clearly does not change the language generated. However, it is still possible to have a dead symbol appearing as an argument to another symbol, as in the above example.

For each function symbol F of $\mathcal{G}$, $\mathcal{G}'$ will have $2^{\rho(F)}$ function symbols representing each possible subset of F. Whenever $\mathcal{G}'$ generates a new function symbol, it "guesses" that certain of the symbol's arguments will eventually be discarded. It refrains from actually producing these arguments and remembers instead that they are missing so that it can insure that the guess will eventually be correct. $\mathcal{G}'$ now has the property that every sentence of $L(\mathcal{G}') = L(\mathcal{G})$ can be generated with a derivation which contains no dead symbols. Thus, if we form $\mathcal{G}''$ by eliminating from $\mathcal{G}'$ all those rules containing dead symbols, the language generated is unchanged. $\mathcal{G}''$ is then the desired grammar.

The above construction becomes effective upon verifying that it is possible to test if a symbol is dead.

We may now proceed to the main result that an explicitly stated IO language is not OI.

Definition 5.12 Let
$L_2 = \{1^m(c1^m)^{n-1} \mid m \geq 1 \text{ and } n = 2^m\}$.

Theorem 5.13. $L_2$ is an IO language.

Proof. A grammar for $L_2$ is given in example 2.7.

Theorem 5.14. $L_2$ is not an OI language.

Proof. The proof comprises two parts. We first show that given an OI grammar $\mathcal{G}$ for $L_2$, we can find a linear basic grammar $\mathcal{G}'$ which also generates $L_2$. A linear basic grammar is an IO or OI grammar (it makes no difference which) in which the right side of each rule contains at most one function symbol. In the second part of the proof we show that no linear basic grammar can generate $L_2$.

$L_2$ has the special property that given any string in $L_2$ one can determine the whole string by looking at any small substring, so long as the substring contains at least two c's (or either end segment). Thus, the strings of $L_2$ are highly redundant; every subpart between c's must be the same length, and even the number of subparts is a function of the length of the subpart. Intuitively, this means that to generate $L_2$, the whole string must be generated in some coordinated way. For example, at no stage of the derivation could we have the sentential form AB where both A and B can generate more than one string, for at least one does not generate all the c's in the string, so the string it may generate is uniquely determined by the string which is generated by the other. Assume $A \overset{*}{\Longrightarrow} 11c1$. Then the only string w such that $11c1w \in L$ is $w = 1c11c11$, and hence that is the only string which B can generate.

This same idea is used to show that an OI grammar for $L_2$ also cannot use nesting to any advantage. Suppose, for example, that F(G) is a sentential form of an OI grammar for $L_2$ which has no dead symbols. There are two cases: either G can generate exactly one terminal string or it can generate more than one.

If G can generate only one terminal string, say w, then there was no point in producing the G in the first place. A new grammar can "guess" that such is the case and just generate F(w) instead of F(G).

On the other hand, if G can generate more than one terminal string, then F(x) cannot generate any string containing more than one occurrence of x, for if it could, then F(G) could generate a string containing more than one occurrence of G, and by the reasoning we used to show that AB could not be a sentential form, we see that F(G) also could not be a sentential form. Hence, F(x) can give rise to strings containing at most one occurrence of x. Moreover, again

by the same line of reasoning, F(x) can generate at most a single string containing one x, say $F(x) \xRightarrow{*} rxs$, for if $F(x) \xRightarrow{*} r'xs'$, then $F(G) \xRightarrow{*} rGs$ and $F(G) \xRightarrow{*} f'Gs'$, so r=r' and s=s' since we are still assuming that G can generate more than one terminal string. Hence, a new grammar could "guess" that such is the case and generate rGs directly instead of F(G).

Finally, if F(x) can also generate a string v containing no occurrences of x, then the new grammar can "guess" that this will occur and generate instead F(w), where w is any string generable from G. (We want to insure that no new strings can be generated.)

These ideas can be extended to function symbols of more than one argument, thus allowing us to construct a linear basic grammar equivalent to any OI grammar for $L_2$. The complete construction and proof may be found in [4].

We now show that no linear basic grammar can generate $L_2$. Assume $\underset{\sim}{G}$ is a linear basic grammar for $L_2$. Let $\underset{\sim}{G}'$ be the result of putting $\underset{\sim}{G}$ into argument-preserving, e-free form (possible since $\underset{\sim}{G}$ is a special case of an IO grammar). We can in fact do this in such a way that $\underset{\sim}{G}'$ is also linear basic, and we will assume that this has been done.

For each $M \geq 1$, we consider the derivation in $\underset{\sim}{G}'$ of the string $w_M = 1^M(c1^M)^{2^M-1} \in L_2$. Every sentential form of the derivation (except for the last, which is w) is of the form $rF(v_1, \ldots, v_{\rho(F)})s$ for some function symbol F and terminal strings r, s, $v_1, \ldots, v_{\rho(F)}$. We call such a sentential form c-bound if one of the strings r, s, $v_1, \ldots, v_{\rho(F)}$ contains two or more c's. Since $\underset{\sim}{G}'$ is argument-preserving, every sentential form derivable from a c-bound form is also c-bound.

Given any fixed k, we can insure, for sufficiently large M, that in the derivation of $w_M$, there are more than k length-increasing steps beginning from the first c-bound form.

If we choose k to be the number of function symbols in the grammar, then, for sufficiently large M, there must be some function symbol F such that $rF(v_1, \ldots, v_{\rho(F)})s$ is a c-bound sentential form,

$$rF(v_1, \ldots, v_{\rho(F)})\, s \xRightarrow{*} r'F(v'_1, \ldots, v'_{\rho(F)})s' \xRightarrow{*} w_M$$

and the length of $r'F(v'_1, \ldots, v'_{\rho(F)})s'$ is greater than the length of $rF(v_1, \ldots, v_{\rho(F)})s$. By repeating the sequence of rules used in the derivation $rF(v_1, \ldots, v_{\rho(F)})s \xRightarrow{*} r'F(v'_1, \ldots, v'_{\rho(F)})s'$, we can derive arbitrarily long terminal strings from $rF(v_1, \ldots, v_{\rho(F)})s$. But this is impossible since each such string must contain the substring $c1^Mc$ and hence they cannot all be in $L_2$. Hence, no linear basic grammar can generate $L_2$.

Theorem 5.15. The class of OI languages does not contain the class of IO languages.

Proof. Immediate from theorems 5.13 and 5.14.

## 6. Conclusion

We have investigated two natural generalizations of context-free grammars which still maintain many of the "nice" properties of context-free languages. It is somewhat surprising, in view of the Church-Rosser theorem of lambda calculus,[2] to find that the order of derivation does indeed make a difference in the class of languages defined. The difference is accounted for by the non-determinicity of a generative grammar.

It is also somewhat surprising that neither convention on the order of evaluation is strictly more powerful than the other. This suggests that a method for controlling the order of expansion might yield a still larger class, containing both the IO and the OI while maintaining the decidability of the emptiness problem. Quoted grammars[4] achieve this; however, it is not known whether the class of quoted languages is contained in the class of context-sensitive languages, nor whether it is closed under inverse homomorphism or intersection with a regular set.


## Acknowledgement

I am indebted to Mr. John S. James for the original idea which led to my doctoral dissertation, upon which this paper is based. I am grateful to my thesis advisor, Prof. Sheila A. Greibach and to the other members of my committee, Prof. Anthony G. Oettinger and Mr. Thomas E. Cheatham, Jr., for their valuable guidance, suggestions, and criticism of the research.

This research was done while I held a National Science Foundation Graduate Fellowship.

References


1. Aho, A.V. "Indexed Grammars - An Extension of Context-Free Grammars", Doctoral dissertation, Princeton University (1967). Excerpts in IEEE Conference Record of 1967 Eighth Annual Symposium on Switching and Automata Theory (October, 1967) pp. 21-31.

2. Church, A. The Calculi of Lambda-Conversation, Princeton University Press, Princeton (1941).

3. Evey, R.J. "The Theory and Application of Pushdown Store Machines", Mathematical Linguistics and Automatic Translation, Harvard University Computation Laboratory Report NSF-10 (May, 1963).

4. Fischer, M.J. "Grammars with Macro-like Productions", Doctoral dissertation, Harvard University (1968). Reprinted in Mathematical Linguistics and Automatic Translation, Harvard University Computation Laboratory Report NSF-22 (May 1968).

5. Ginsburg, S. and Greibach, S.A. "Abstract Framiles of Languages", System Development Corporation Report TM-738/031/00, April 17, 1967. To appear in a Memoir of the American Mathematical Society.

6. Ginsburg, S., Greibach, S., and Harrison, M. "Stack Automata and Compiling", Journal of the Association for Computing Machinery, vol. 14, no. 1 (January 1967), pp. 172-201.

7. Ginsburg, S., Greibach, S., and Harrison, M. "One-Way Stack Automata", Journal of the Association for Computing Machinery, vol. 14, no. 2 (April, 1967), pp. 389-418.

8. Ginsburg, S., Greibach, S., and Hopcroft, J. "Pre-AFL", System Development Corporation Report TM-738/037/00, August 14, 1967. To appear in a Memoir of the American Mathematical Society.

9. Greibach, S. and Hopcroft, J. "Independence of AFL Operations", System Development Corporation Report TM-738/034/00, July 12, 1967. To appear in a Memoir of the American Mathematical Society.

10. James, J.S. "A Formal Macro Assembler - Version IV", unpublished paper, National Institute of Mental Health, 1965.

11. Kuroda, S.Y. "Classes of Languages and Linear-Bounded Automata", Information and Control, vol. 7 (1964), pp. 207-223.

12. Landweber, P.S. "Three Theorems on Phrase-Structure Grammars of Type 1", Information and Control, vol. 6 (1963), pp. 131-136.

13. McIlroy, M.D. "Macro Instruction Extension of Compiler Languages", Communications of the Association for Computing Machinery, vol. 3 (April, 1960), pp. 214-220.

14. Mooers, C.N. "TRAC, A Procedure-Describing Language for the Reactive Typewriter", Communications of the Association for Computing Machinery, vol. 9 (March, 1966), pp. 215-219.

15. Myhill, J. "Finite Automata and the Representation of Events", Wright Air Development Command Technical Report 57-624 (1957), pp. 112-137.

16. Rabin, M.O. and Scott, D. "Finite Automata and Their Decision Problems", IBM Journal of Research and Development, vol. 3 (1959), pp. 114-125. Reprinted in Moore, E.F. (ed.), Sequential Machines, Addison-Wesley Publishing Company, Inc., Reading, Mass. (1964).

17. Strachey, C. "A General Purpose Macrogenerator", Computer Journal, vol. 8 (1965), pp. 225-241.