AD-A050 794 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/2
ARRAY GRAMMARS.(U)
JAN 78 A ROSENFELD AFOSR-77-3271
UNCLASSIFIED TR-629 AFOSR-78-0353 NL

| OF |
AD
A050794

END
DATE
FILMED
4-78
DDC

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(14) TR-629
AFOSR-77-3271

(11) Jan[illegible]978

(6) ARRAY GRAMMARS

(10) Azriel/Rosenfeld
Computer Science Center
University of Maryland
College Park, MD 20742

(9) Interim rept.,

(12) 53 P.

(15) AFOSR-77-3271

(16) 2304 (17) A2

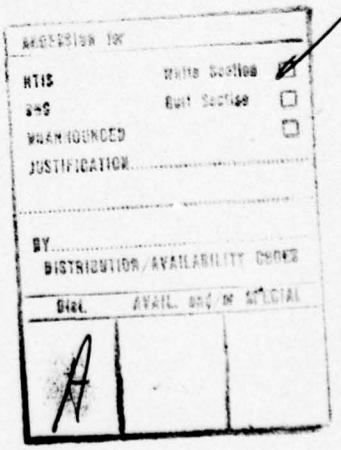
(18) AFOSR (19) 78-0353

ABSTRACT

This report is a draft of Chapter 8 of a forthcoming book on "Picture Languages." [Drafts of Chapter 2, on digital topology, and Chapters 3-5, on sequential and parallel picture acceptors, were issued earlier as TR's 542 and 613, respectively.] This chapter deals with array grammars, with emphasis on their relationship to acceptors; some preliminary material on string grammars is also included. Comments on the choice and treatment of the material are invited.

The support of the Directorate of Mathematical and Information Sciences, U. S. Air Force Office of Scientific Research, under Grant AFOSR-77-3271, is gratefully acknowledged, as is the help of Shelly Rowe and Judy Myrick in preparing this paper.

Table of Contents



Chapter 8

ARRAY GRAMMARS

In the previous chapters we have discussed various types of string and array acceptors. This chapter introduces grammars that can be used to either generate or accept ("parse") sets of strings or arrays. It defines a hierarchy of such grammars, and establishes equivalences between certain classes of grammars and of acceptors.

1. String grammars

We first consider grammars that generate or parse sets of strings. Sections 1.1-2 review the usual definitions of such grammars. Section 1.3 defines "isometric" grammars that rewrite #s, rather than simply extending the given string; this alternative approach will be important when we treat array grammars in Section 3. Section 1.4 discusses various definitions of "parallel" grammars and their relationships to conventional (sequential) grammars.

1.1 Grammars

Informally, a (string) grammar G is a mechanism that generates a set of strings by a process of repeatedly substituting one substring for another, starting with a standard initial string. The (rewriting) rules or productions of G specify which substrings can be replaced by which others. The language of G is the set of strings (usually required to consist entirely of symbols of a special type, called "terminal" symbols) that can be produced in this way. We shall now define these ideas more precisely.

As we did in Chapter 3 for automata, we shall require that grammars satisfy certain finiteness and nontriviality conditions. Specifically, we shall assume that the vocabulary, or set of symbols, the subset of "terminal" symbols, and the set of rules are all finite, nonempty sets. We shall also assume that the initial string consists of a single non-terminal symbol S. Under these assumptions, we can formally define a ("type 0") grammar G to be a 4-tuple (V,V_T,P,S), where

V is the vocabulary (a finite, nonempty set of symbols)

$V_T \subseteq V$ is the terminal vocabulary ($\neq \emptyset$)

$S \in V-V_T$ is the initial symbol

P is a finite, nonempty set of pairs (α,β), where α and β are non-null* strings of elements of V. These pairs

---

*We require β to be non-null to prevent the null string from being generated by a grammar; this corresponds to our requirement, in earlier chapters, that the input strings of acceptors are always non-null.

are called the rules of G, and are usually written in the form $\alpha \to \beta$ (denoting the fact that α can be replaced by β).

In order to define the language of G, we must introduce the notion of a "derivation" in G. We say that the string τ is <u>directly derived</u> from the string σ in G (notation: $\sigma \underset{G}{\Rightarrow} \tau$) if there exists a rule $\alpha \to \beta$ of G such that α is a substring of σ, and τ is the result of replacing some occurrence of α (as a substring of σ) by β. We say that τ is <u>derived</u> from σ in G (notation: $\sigma \overset{*}{\underset{G}{\Rightarrow}} \tau$) if there exists a sequence of strings $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_n = \tau$, where $n \geq 0$, such that σ_i is directly derived from σ_{i-1} in G, $1 \leq i \leq n$. The set of strings on V_T that can be derived in G from the initial string S (consisting of the single symbol S) is called the <u>language</u> of G, and is denoted by L(G). A sequence of strings $S = \sigma_0, \sigma_1, \ldots, \sigma_n = \tau \in L(G)$ for which $\sigma_{i-1} \underset{G}{\Rightarrow} \sigma_i$ is called a <u>derivation</u> of τ in G. (We will omit "in G" from now on unless there is danger of confusion.)

Given any grammar G, there exists a grammar G' in which terminal symbols are never rewritten by any rule, and such that L(G') = L(G). In fact, we can define G' to have nonterminal vocabulary $(V - V_T) \cup V_T'$, where V_T' is a set of primed copies of the symbols in V_T. For each rule $\alpha \to \beta$ of G, we have the rule $\alpha' \to \beta'$ in G', where the primes denote the fact that each symbol in V_T is replaced by the corresponding symbol in V_T'. In addition, G' has the rules $x' \to x$

for all $x \in V_T$. Evidently a string on V_T is derivable in G' if it is derivable in G (since G' can derive the corresponding string on V_T', which can then be changed into a V_T string using the $x' \to x$ rules), and conversely (since the $x' \to x$ rules are the only rules of G' that produce symbols in V_T). [Note that if the $x' \to x$ rules are applied too soon, it may be impossible to complete a derivation, since a rule $\alpha' \to \beta'$ may require the presence of some primed symbol that has already become unprimed.] We shall assume from now on that grammars never rewrite terminal symbols.

A grammar can function as an "acceptor" by performing derivations in reverse. Given a string σ on V_T, we say that G parses σ if there exists a sequence of strings $\sigma = \sigma_n, \sigma_{n-1}, \ldots, \sigma_0 = S$ such that σ_{i-1} is the result of applying a rule $\alpha \to \beta$ of G in reverse (i.e., replacing β by α) to σ_i, $1 \leq i \leq n$. It is clear that this is the same thing as saying that $\sigma_0, \ldots, \sigma_n$ is a derivation; thus σ is parsed by G iff. it is generated by G.

A less trivial equivalence between grammars and acceptors is provided by

Theorem 1.1. The languages generated by grammars are the same as the languages accepted by Turing acceptors.

Proof: Let L be generated by G; then we can define a Turing acceptor A that operates as follows: Given a string σ, A creates an S (separated from σ by a distinctive marker) and then (nondeterministically) applies a sequence of rules of

G, starting with the S (and leaving σ intact), to produce a sequence of strings $\sigma_0 \equiv S, \sigma_1, \sigma_2, \ldots$ At each step of this process, it compares σ_i with σ (this can be done by moving back and forth from σ_i to σ, and marking off corresponding symbols if they are the same). If at any stage A finds that $\sigma_i = \sigma$, it accepts. Evidently this happens iff. $\sigma \in L(G)$.

Conversely, let L be accepted by A. We shall define below a grammar G_A that generates two copies of an arbitrary string, and simulates the operation of A on one copy. If the simulation accepts, G_A erases that copy and converts the other copy to a terminal string. Evidently a terminal string σ is produced by G_A in this way iff. $\sigma \in L(A)$.

The grammar G_A has the following sets of rules:

(1)
$$\left.\begin{array}{l} S \rightarrow S'(x',x) \\ S' \rightarrow S'(x,x) \\ S' \rightarrow ((q_0,N),\ (x'',x)) \\ S \rightarrow ((q_0,N),\ (x^*,x)) \end{array}\right\} \text{for all } x \text{ in the vocabulary of A}$$

These rules generate a string of the form $((q_0,N)(x^*,x))$, or $((q_0,N)(x''_1,x_1))(x_2,x_2),\ldots,(x_{n-1},x_{n-1}),(x'_n,x_n)$ (where $n > 1$). Primed x's are always at the right end of the string; double-primed x's, at the left end; and starred x's are always singletons. Here q_0 represents the initial state of A at the left end of its input string.

(2) For all u,v,w,x,y,z,

(a) $((q_1,d)(w,x)) \rightarrow ((q_2,N)(t,x))$ for all $(q_2,t,N) \in \delta(q_1,w,d)$, where δ is the transition

function of A, and where if w is primed, double-primed, or starred, so is t. These rules simulate the case where A changes state and rewrites a symbol but does not move.

(b) $(u,v)((q_1,d)(w,x)) \rightarrow ((q_2,L)(u,v))(t,x)$ for all $(q_2,t,L) \in \delta(q_1,w,d)$, where if w is primed, so is t. These rules handle the case where A moves left, provided it was not at the left end of the string.

(b') $\left.\begin{array}{l}((q_1,d)(w",x)) \rightarrow ((q_2,L)(\#",\#))(t,x) \\ ((q_1,d)(w*,x)) \rightarrow ((q_2,L)(\#",\#))(t',x)\end{array}\right\}$ for all $(q_2,t,L) \in \delta(q_1,w,d)$

Rules (b') handle the case where A is at the left end of the non-#s and moves left.

(c) $((q_1,d)(w,x))(y,z) \rightarrow (t,x)((q_2,R)(y,z))$ for all $(q_2,t,R) \in \delta(q_1,w,d)$, where if w is double-primed, so is t. These rules handle the case where A moves right and was not at the right end of the string.

(c') $\left.\begin{array}{l}((q_1,d)(w',x)) \rightarrow (t,x)((q_2,R)(\#',\#)) \\ ((q_1,d)(w*,x)) \rightarrow (t",x)((q_2,R)(\#',\#))\end{array}\right\}$ for all $(q_2,t,R) \in \delta(q_1,w,d)$,

Rules (c') handle the case where A is at the right end of the non-#s and moves right.

These rules simulate A on the string, adding pairs of #s at its ends if necessary. Note that the conventions about primes,

double-primes and stars are preserved by these rules. Note also that the simulation affects only the first terms of the pairs; the second terms are unchanged, except that # second terms may be added at the ends. We assume in (2) that q is not an accepting state of A.

(3) For all w,x,y,z, where w may be double-primed and y may be primed, and for all accepting states q_A of A,

$$((q_A,d)(w,x))(y,x) \rightarrow (w,x)((q_A,d)(y,z))$$

When an accepting state is created, it moves rightward.

(4) For all w,x,y,z, where w may be double-primed,

$$((q_A,d)(y',z)) \rightarrow ((y',z)q_A) \text{ if } z \neq \#$$
$$(w,x)((q_A,d)(y',\#)) \rightarrow ((w,x),q_A')$$
$$((q_A,d)(y^*,z)) \rightarrow z$$

When (q_A,d) reaches the right end of the string, it becomes q_A if the second term at the right end is non-#, or erases the # and becomes q_A' if the second term at the right end is #. If the string is a singleton, (q_A,λ) turns it into a singleton terminal (the second term must be non-# in this case, since the original string of second terms was non-null).

(5) For all w,x,y, where w may be double-primed in the second case,

$(w,\#)((y,\#),q_A') \rightarrow ((w,\#),q_A')$

$(w,x)((y,\#),q_A') \rightarrow ((w,x),q_A)$ if $x \neq \#$

The q_A' moves leftward, erasing #s; when it reaches a non-# second term (which must happen eventually, since the original string was non-null), it becomes q_A.

(6) For all w,x,y,z, where y may be primed,

$(w,x)((y,z),q_A) \rightarrow ((w,x),q_A)z$ if $x \neq \#$

$((y",z)q_A) \rightarrow z$

The q_A moves left, turning pairs with non-# second terms into their second terms; if it reaches the left end of the string, it erases itself, leaving only its pair's second term z. Note that z *cannot* be #, by (4-6).

(7) For all w,y,z, where y may be primed,

$(w,\#)((y,z),q_A) \rightarrow ((y,z),q_A)$

$(w",\#)((y,z),q_A) \rightarrow z$

If the q_A reaches # second terms (which must lie to the left of all the non-# second terms), it erases them; when it erases the last one, it also erases itself, leaving only its pair's second term z.

Evidently, this process generates a terminal string (namely, the original string of second terms; the terminal vocabulary of G_A is the same as the vocabulary of A) iff. the simulation

of A accepted that string (of first terms); in other words, G_A generates a terminal string σ iff. $\sigma \in L(A)$.//

1.2. Special types of grammars

A grammar G is called context-sensitive (or "type 1") if for each rule $\alpha \to \beta$ of G there exist strings ξ, η, and τ on V, where τ is non-null, and a symbol $A \in V - V_T$, such that $\alpha = \xi A \eta$ and $\beta = \xi \tau \eta$. Thus any such rule rewrites a single nonterminal A, "in the context of" ξ and η, as a non-null string τ. Note that in such a grammar, terminals are never rewritten.

We shall call a grammar G monotonic if for each rule $\alpha \to \beta$ of G we have $|\alpha| \leq |\beta|$.

Theorem 1.2. The languages generated by monotonic grammars are the same as those generated by context-sensitive grammars.

Proof: Context-sensitive certainly implies monotonic, since τ non-null implies $|\xi A \eta| \leq |\xi \tau \eta|$. To prove the converse, we show that any rule $\alpha \to \beta$ with $|\alpha| \leq |\beta|$ can be replaced by a set of context-sensitive rules without altering the language generated by the grammar. We can assume, as shown earlier, that the given grammar does not rewrite terminals, so that the symbols involved in the rule $\alpha \to \beta$ are all nonterminals (unless the rule is of the form $x' \to x$, which is already context-sensitive).

Let $\alpha = A_1 \ldots A_m$ and $\beta = B_1 \ldots B_n$, where $m \leq n$. Then we can replace $\alpha \to \beta$ by the following sets of context-sensitive rules:

$$A_1 \to A_{11} \tag{1}$$
$$A_{11}A_2 \to A_{11}A_{22}$$
$$\vdots$$
$$A_{m-1,m-1}A_m \to A_{m-1,m-1}A_{mm}$$

These rules successively change the symbols in α to special forms which are understood to be unique to the given rule and to the given position in that rule, so that they can only be applied in the given sequence. If a rule is applied in the wrong place (only the beginning of α is present, or a later part of α is the same as its beginning), the rule sequence (1) cannot go to completion, and there is no way to get rid of the last-created special symbol (unless the rest of α is later created; but if this happens, it cannot depend on the presence of the special symbols, and so could have happened earlier).

$$\begin{aligned} A_{11}A_{22} &\rightarrow B_1A_{22} \\ A_{22}A_{33} &\rightarrow B_2A_{33} \\ &\vdots \\ A_{mm} &\rightarrow B_mB_{m+1}\cdots B_n \end{aligned} \tag{2}$$

These rules change the special A's to B's after the next ones have been created (except in the case of the last one). As already pointed out, this process can go to completion only if all the special A's are created, which can only happen if all of α is present (or could have been present). Thus replacing $\alpha \rightarrow \beta$ by (1-2) cannot change L(G), since the only way (1-2) can create a string free of special symbols is to simulate a complete application of $\alpha \rightarrow \beta$, and analogously for all the other rules of G.//

Theorem 1.3. The languages generated by monotonic grammars are the same as those accepted by tape-bounded acceptors.

Proof: When a terminal string σ is derived from S in a monotonic grammar, the successive strings in the derivation have nondecreasing lengths, so that none of them is longer than σ. Hence in the first part of the proof of Theorem 1.1, A can generate an S by turning some symbol x of σ into a pair (S,x), and then simulating the application of a sequence of rules, creating further pairs as necessary, and shifting the pairs leftward or rightward if it comes to the end of σ. If the string of first terms generated in this way tries to become longer than σ, A cannot possibly be simulating a derivation of σ. Thus in carrying out the proof of the first part of Theorem 1.1 in the case where G is context-sensitive, A need never move off the non-#s.

Conversely, in the second part of the proof, note that the rules of G_A are all monotonic except for some rules in (4-7) which erase #s; but these rules are not needed if A is tape-bounded, since rules (2b') and (2c') are never used, so that pairs with # second terms are never created. (Since the string ends are marked " and ', the simulation of A need not involve #s at all, even when A bounces off them.) Thus if A is tape-bounded, G_A is monotonic.//

A grammar G is called context-free (or "type 2") if in all rules $\alpha \to \beta$ of G, α is a single nonterminal symbol and β is non-null. (Evidently context-free implies context-

sensitive.) G is called finite-state (or "type 3") if in all rules $\alpha \to \beta$ of G, α is a single nonterminal symbol and β either ends in a single nonterminal symbol, possibly preceded by terminal symbols, or else β is a nonnull string of terminal symbols.

Theorem 1.4. The languages generated by finite-state grammars are the same as the languages accepted by finite-state acceptors.

Proof: If given G, we define an FSA,A, that operates on its input string σ as follows: A examines the right end of σ for matches to right-hand rules of G. If $\sigma = \sigma_1\beta_1$, where $B_1 \to \beta_1$ is a rule of G, A memorizes B_1, moves to just past the beginning of β_1 (i.e., to the right end of σ_1), and looks for matches of the right end of $\sigma_1 B_1$ to right-hand sides of rules of G. If it finds such a match, say to $B_2 \to \beta_2 B_1$, where $\sigma_1 = \sigma_2\beta_2$, this process is repeated, with σ becoming shorter at each repetition. If at some stage A finds that all of $\sigma_n\beta_n$ matches the right-hand side of a rule of G whose left-hand side is S, then A accepts σ. Evidently this can happen iff. σ can be derived in G, using the succession of rules $S \to \beta_n B_{n-1}$, $B_{n-1} \to \beta_{n-1}B_{n-2}$, ..., $B_2 \to \beta_2 B_1$, $B_1 \to \beta_1$. Thus A accepts σ iff. $\sigma \in L(G)$.

Conversely, suppose that L is accepted by a one-way FSA, A. We can define a finite-state grammar G that generates L as follows:

1) $S \to (q_0, x)$ for all x

2) $(q,x) \rightarrow x(q',y)$ for all $q' \in \delta(q,x)$ and all y, where δ is A's transition function, and q is a non-accepting state of A.

3) $(q_A,x) \rightarrow x$ for all accepting states q_A of A.

Here the x's (the vocabulary of A) constitute the terminal vocabulary of G. Thus G generates a string and simulates A's behavior on it; it generates an all-terminal string σ iff. A accepts σ.//

Theorems 1.1-4 tell us that the languages of (arbitrary, context-sensitive, finite-state) grammars are the same as the languages of (Turing, tape-bounded, finite-state) acceptors, respectively. [It can be shown (e.g., [1, pp. 74-78]) that the languages of context-free grammars are the same as the languages of one-way, nondeterministic pushdown acceptors, but we will not prove this here.] Thus introducing these types of grammars has not expanded our language hierarchy.

1.3 Isometric grammars

In this section we consider grammars in which the left and right hand sides of any rule $\alpha \to \beta$ have the same length -- i.e., $|\alpha| = |\beta|$. In such "isometric" grammars (called "isotonic" in [2]), strings grow by rewriting #s, and the initial string is $\#^{\infty}S\#^{\infty}$. We shall show that such grammars generate the same set of languages as do ordinary grammars, and that by suitably restricting them, we obtain the context-sensitive and finite-state languages. The restriction that, in any rule $\alpha \to \beta$, α and β must be of the same size will prove to be important when we define array grammars in Section 3.

Formally, an isometric grammar is a 5-tuple $G = (V, V_T, P, S, \#)$, where V, V_T, P and S are defined just as in Section 1.1, except that for all pairs $(\alpha, \beta) \in P$ we require that

a) $|\alpha| = |\beta|$

b) α does not consist entirely of #s

c) Replacing α by β cannot disconnect or eliminate the non-#s*.

The symbol $\# \in V - V_T$ is called the blank symbol. Derivations are defined exactly as in Section 1.1. The language of G is the set of all (non-null) strings σ on V_T such that $\#^{\infty}\sigma\#^{\infty}$ can be derived in G from the initial infinite string $\#^{\infty}S\#^{\infty}$.

---

*Readily, condition (c) is equivalent to the following: The non-#s in β exist and are connected; if α has a non-# at its left(right) end, so does β.

Theorem 1.5. The languages generated by isometric grammars are the same as those generated by ordinary grammars.

Proof: For any grammar G, we can define an isometric grammar G' that generates exactly L(G) as follows:

1) For every rule $\alpha \to \beta$ of G such that $|\alpha| \leq |\beta|$, G' has the rule $\alpha\, \eta^{|\alpha|-|\beta|} \to \beta$, where η is a special nonterminal symbol.
2) For every rule $\alpha \to \beta$ of G such that $|\alpha| \geq |\beta|$, G' has the rule $\alpha \to \beta\, \eta^{|\alpha|-|\beta|}$.
3) In addition, G' has the rules $x\# \to x\eta$, $x\eta \to \eta x$, and $\#\eta\, x \to \#\#x$, for all non-#x. These rules allow #s to be changed into ηs at the right end of the string of non-#s; ηs to be shifted leftward; and ηs to be changed back to #s at the left end of the string.

It is easy to see that G' can generate a terminal string σ (surrounded by #s) iff. just enough ηs are created, and they are shifted into just the right positions, to allow a derivation of σ in G to be simulated using rules in (1-2) (The terminal vocabulary of G' is the same as that of G.) Thus L(G') = L(G).

Conversely, given an isometric grammar G', we can define G such that, for every rule $\alpha \to \beta$ of G', G has the rule $\alpha' \to \beta'$, where α' and β' are the same as α and β but with #s,

if any, omitted. It is not hard to see that G can generate a terminal string σ iff. G' generates $\#^{\infty}\sigma\#^{\infty}$. (We recall that under G' the non-#s remain connected; thus derivations in G differ from those in G' only in that, when G' destroys or creates #s at its ends, G simply grows or shrinks.)//

Theorem 1.6. The languages generated by isometric grammars that never create #s (i.e., for all rules $\alpha\rightarrow\beta$, there can be #s in β only in positions corresponding to #s in α) are the same as those generated by monotonic grammars.

Proof: In the first part of the proof of Theorem 1.5, if G is monotonic there are no rules of type (2); hence we can omit the rules $\#\natural x \rightarrow \#\#x$ and still guarantee that derivations in G can be simulated, by creating and shifting in just enough $\natural$s to allow the rules of type (1) to operate. Thus if G is monotonic, we can define G' so that it never creates #s. In the second part of the proof, if G' never creates #s, then for all rules $\alpha\rightarrow\beta$ of G' we have $|\alpha'| \leq |\beta'|$, so that G is monotonic.//

Theorem 1.7. The languages generated by isometric grammars whose rules are all of the form $\beta\#^{m} \rightarrow \beta$, where $|\beta| = m+1$ and β is a string of terminals ending in at most one (non-#) nonterminal, are the same as the languages generated by finite-state grammars.//

1.4 Parallel grammars

As defined in Sections 1.1-2, a grammar generates (or parses) a string by applying only one rule $\alpha \to \beta$ at a time, and replacing only one instance of α by β when this rule is applied. In this section we briefly discuss several approaches to defining "parallel grammars", in which rules are applied in more than one place at a time.

We first consider an approach [2] in which we still apply only a single rule $\alpha \to \beta$ at a given time, but we apply this rule by replacing every instance of α by β. One problem with this approach is that instances of α can overlap. For example, if we apply the rule AA → BB to the string AAA, in parallel, the result is (presumably) BBBB, since AAA contains two instances of AA; thus application of the isometric rule AA → BB changes the length of the string. (Indeed, if we apply AAA → BB to A^m in parallel, we get $B^{2(m-2)}$, so that applying a length-decreasing rule can increase the string length.) In ordinary ("sequential") rule application, this could not happen; applying the rule AA → BB to AAA yields either BBA or ABB, to which the rule no longer applies.

Another problem with parallel rule application is that parsing and generating are no longer inverses of one another. For example, applying AA → BB to AAA in parallel yields BBBB, but applying BB → AA to BBBB in parallel yields AAAAAA, since BBBB contains three instances of BB. In fact (see [2]), it is not hard to exhibit a grammar G such that the sets of

strings generated and parsed by G "in parallel" are disjoint from one another and from L(G).

The "parallel language" $L_p(G)$ generated by a given grammar G is not the same as G's "sequential language" L(G) in general, but it is the same in certain important cases. For example, consider the finite-state grammars; it is evident that these have the property that, at any stage of a derivation, at most one nonterminal symbol is present. (Grammars with this property are called "linear".) Since the rules of a finite-state grammar all have left-hand sides consisting of a single nonterminal symbol, it is clear that at any stage of a derivation, a given rule can apply in at most one place; thus it makes no difference whether a rule is applied "sequentially" (i.e., in one place) or in parallel. In other words, if G is a finite-state (or more generally, a linear) grammar, we have $L_p(G) = L(G)$.

It can also be shown that the class of languages generated in parallel by grammars is the same as the class of languages generated in the usual way; and similarly for the classes of languages generated sequentially and in parallel by monotonic grammars. To prove that any (sequential) language is a parallel language, we show that for any grammar G, there exists a grammar G* with $L(G^*) = L(G)$, such that at any step of a derivation in G*, no rule applies at more than one place. Specifically, for each rule $A_1 \ldots A_m \rightarrow B_1 \ldots B_n$ of G, G* has the rule $\bar{A}_1^* \bar{A}_2 \ldots \bar{A}_m \rightarrow \bar{B}_1^* \bar{B}_2 \ldots \bar{B}_n$, where the barred symbols are all nonterminals. We also give

G* the rule $S \rightarrow \overline{S}*$ and the set of rules

$$\overline{A}*\overline{B} \rightarrow \overline{AB}* \quad \text{and} \quad \overline{AB}* \rightarrow \overline{A}*\overline{B}$$

for all pairs of nonterminals one of which is starred and the other is not. These rules initially create a * and allow it to shift from symbol to symbol. Since the rules that correspond to those of G all involve the *, it is clear that any rule applies at only one place (evidently, more than one * never exists). Finally, we give G* the rules

$$\overline{a}*\overline{b} \rightarrow a\overline{b}* \; ; \; \overline{a}*\# \rightarrow a\#$$

for all $\overline{a},\overline{b}$ corresponding to terminal symbols a,b of G (or, if we do not want to introduce the # symbol, we design G* so that the rightmost symbol of any string derivable in it is always uniquely marked). These last rules allow the * to shift rightward through symbols that correspond to terminals, turning them into terminals as it goes, until it reaches the right end of the string and vanishes. It is evident that this process can result in a terminal string iff. G* generated a string of symbols corresponding to terminals, by imitating a derivation in G which leads to a string of terminals; thus the resulting terminal string must be in L(G). (If the last rules are used too soon, or are not started with the * at the left end of the string, a string consisting entirely of terminals will not be created.) Since in G*, no rule can apply in more than one place, we have $L_p(G*) = L(G*)$; thus $L(G) = L(G*) = L_p(G*)$, which proves that any language

is a parallel language. Note also that if G is monotonic, so is G*.

Conversely, we can show that for any grammar G there exists a grammar G' that, in effect, applies the rules of G in parallel; thus $L_p(G) = L(G')$, so that any parallel language is a sequential language. Basically, we define G' to simulate an automaton A' (compare the proof of Theorem 1.1). Suppose that, at a given stage in the operation of A', we are ready to apply a rule of G in parallel to the current string σ (initially, this string is S). A' picks a rule $\alpha \rightarrow \beta$, scans σ, and marks every position in σ at which a match to α begins. It then scans σ again and replaces each instance of α by β. (If instances overlap, the part of σ from the start of one instance of α to the start of the next is replaced by β.) When this process is complete, we are ready to apply another rule of G. Readily, if G is monotonic, so is G'. (A detailed description of G' can be found in [2].)

The remarks in the preceding three paragraphs show that any (arbitrary, context-sensitive, finite-state) language is an (arbitrary, context-sensitive, finite-state) parallel language, and conversely, The analogous result about context-free languages is false. For example, consider the context-free grammar whose rules are

$$S \rightarrow SS;\ S \rightarrow a$$

The parallel language of this grammar is readily the set of

strings $\{a^{2^n} | n=0,1,2,...\}$; but it is well-known that this is not a context-free language in the ordinary sense (see, e.g., [1, p. 57, Theorem 4.7]). Conversely, it can be shown that the parenthesis-string language, which is context-free [1, p. 67], is not parallel context-free. On the relationship between context-free and parallel context-free languages see [3, 4].

Parallel rule application has a particularly convenient interpretation for rules that are in the context-sensitive form $\xi A\eta \rightarrow \xi\tau\eta$: For each instance of $\xi A\eta$ in the given string σ, we replace the A by a τ*. In this case, some of the problems mentioned at the beginning of this section do not arise, since the substrings being replaced (i.e., the A's) cannot overlap, even if the $\xi A\eta$'s do overlap. Parsing and generation are still not always inverses of one anohter; e.g., if we apply AA → AB to AAA we obtain ABB, whereas if we apply AB → AA to ABB we obtain AAB. On the other hand, string length can never decrease, since A's are being replaced by nonnull strings (τ's). The results in the preceding paragraphs all continue to hold for this modified concept of parallel rule application.

---

*Note that this is not the same as replacing each instance of $\xi A\eta$ by $\xi\tau\eta$; for the rule AA → AB, applied to the string AAA, replacing AA's by AB's gives ABAB, whereas replacing A's by B's when they have A's on their left gives ABB.

Another approach to defining parallelism for context-sensitive grammars is to apply rules in all possible positions at the same time. In other words, given a string $\sigma = A_1 \ldots A_n$, for each A_i we choose a rule $\xi_i A_i \eta_i \rightarrow \xi_i \tau_i \eta_i$ of G that applies to A_i in σ (i.e., such that ξ_i precedes A_i in σ, and η_i follows it), and replace A_i by τ_i. (If no such rule exists, we leave A_i unchanged.) This is done in parallel for every symbol in σ; the rule used is chosen independently for each A_i. Parallel rule-application systems of this kind are called L-systems [5-7]; if the rules are all context-free, they are called 0L-systems. L-systems have been extensively studied as models of biological growth; they will not be discussed here in detail.

In defining L-systems, it is customary to make no distinction between terminal and nonterminal vocabularies; the language is the set of all strings that can be derived from the initial string. Another possibility is to define the language as the set of all such strings that are stable under application of the rules; such strings correspond, from the biological growth standpoint, to "adult" organisms. It is not hard to show [8] that the "adult languages" of L-systems are just the context-sensitive languages. In fact, given an L-system H and a string σ, we can easily define a tape-bounded acceptor A that accepts iff. it belongs to the adult language of H. [A first checks that σ is stable under H. It then nondeterministically generates a sequence of strings $\sigma_1, \sigma_2, \ldots$ by applying the rules of H to the initial string σ_0,

and checks at each step whether σ_i matches σ. Since we have assumed that the rules of H are context-sensitive (i.e., the τ_i's are non-null), we know that $|\sigma_0| \leq |\sigma_1| \leq |\sigma_2| \leq \ldots$; thus if σ is derivable from σ_0 in H, no σ_i can be longer than σ, so A has room to store it, as in the proof of Theorem 1.3. Thus if σ is derivable in H, A can discover this fact and accept σ.] Conversely, given a context-sensitive grammar G, we can define a modified G* such that at any stage of a derivation in G* there is only one place where any rule applies, as shown earlier in this section. We can thus define an L-system H* that simulates G*. Note that in the definition of G*, as long as the * remains, there is always some rule that applies; but the * disappears only when all the symbols have been turned into terminals*. Thus the only strings that are stable under the rules of H* are the terminal strings of G*, which are the same as the terminal strings of G; hence the adult language of H* is just L(G). If we modify the definition of an L-system to allow erasing rules (i.e., $\xi A \eta \rightarrow \xi \eta$), it can be shown that the adult languages are just the arbitrary (Turing-machine) languages (but allowing the null string to be in a language); the proof is analogous. [L-systems that do not allow erasing rules are called "propagating".]

This requires a slight modification of the definition of G so that the * can only begin creating terminals when it is at the left end of a string; the details are straightforward.

Similarly, it can be shown that the adult languages of 0L-systems are just the context-free languages. For the details of these proofs, see [8].

In analogy with Section 1.3, one can consider "isometric" parallel grammars (or L-systems) in which the rules are all of the form $\xi A\eta \rightarrow \xi B\eta$, so that strings can grow only by rewriting #s at their ends. L-systems of this special type are essentially nondeterministic cellular automata, in which the transition function is defined by the rewriting rules: state A, in the context of states ξ on the left and η on the right, can change into state B. If #s are never rewritten, they become bounded cellular automata.

2. Matrix grammars

In this section we discuss grammars that generate or parse sets of rectangular arrays. Grammars whose languages are sets of connected (not necessarily rectangular) arrays will be considered in Section 3.

A class of "matrix grammars" [9-11] that generate rectangular arrays can be informally defined as follows:

a) A string grammar G generates a string σ which will become the top row of the array.

b) The symbols in σ are initial symbols of a set of finite-state string grammars $G_1,\ldots,G_n$. These grammars operate in parallel (compare Section 1.4) to generate the columns of the array. Their operation must be coordinated so that in every column, at any given time, a rule of the same length if applied, and that the terminating rules are all applied at the same time.*

Formally, a matrix grammar M is a pair $(G,\bar{G})$, where G is a grammar and $\bar{G} = \{G_1,\ldots,G_n\}$ is a set of finite-state grammars, such that the terminal vocabulary of G is the set $\{S_1,\ldots,S_n\}$

---

*We can assume without loss of generality that the rules of each G_i are all of the forms $A\rightarrow aB$ and $A\rightarrow a$, where A,B are nonterminals and a is a terminal. Under this assumption, it suffices to require that the terminating rules are applied in every column at the same time.

of initial symbols of the G_i's. (We assume that the nonterminal vocabularies of the G_i's are disjoint.) M operates by generating a (horizontal) string σ of S_i's using the rules of G, and then generating a rectangular array from the top row σ by applying rules of the G_i's in parallel, and finally terminating simultaneously.

Matrix grammars are of interest because they generate (or parse) rectangular arrays, but unfortunately the classes of array languages that they generate are not the same as the classes accepted by the various types of array acceptors. Specifically, we give the following examples:

a) If G is finite-state, the language L of M is a finite-state array language. [Indeed, we can define a finite-state array acceptor A that accepts L as follows: Given a rectangular array R, A scans the last column C of R, simulates acceptors for the finite-state column languages, and thus determines to which of those languages, if any, C_m belongs. A can thus verify whether a terminating rule of G could have produced any of the initial symbols S_m that began the generation of C_m. A then scans the next-to-last column C_{m-1}, determines which symbols S_{m-1} could have begun its generation, and verifies whether a rule of G could have produced any of these symbols together with a nonterminal that then resulted in S_m. This process is repeated

until all of R has been accounted for.] Conversely, however, there are (deterministic, tape-bounded) finite-state rectangular array languages that cannot be generated by any such M. For example, consider the set L of square arrays of even side length whose rows are all of the form $a^m b^m$. No such M can generate L, since the string language $\{a^m b^m | m=1,2,...\}$ is not finite-state (see Section 3.2 of Chapter 3). On the other hand, a finite-state array acceptor can verify that its input array R is in L as follows: Verify squareness (and even side length) by moving "diagonally"; then move at "$67\frac{1}{2}°$" (alternately two steps upward and one leftward, starting at the lower right corner) to find a middle column; finally, move down that column and verify that, in each row, there are a's on its left and b's on its right. Thus the class of matrix languages for which G is finite-state is a proper subset of the class of finite-state array languages. [Readily, we can require A in the first part of the proof to be deterministic and tape-bounded; thus these matrix languages are a proper subset of the deterministic, tape-bounded finite-state array languages.]

b) If G is context-free, the language of M is not finite-state. For example, the set L of rectangular

arrays of even width whose rows are all of the form $a^m b^m$ can be generated by such an M (G generates $S_1^m S_2^m$; the rules of G_1 and G_2 are $S_1 \rightarrow aS_1, S_1 \rightarrow a$, and those of G_2 and $S_2 \rightarrow bS_2, S_2 \rightarrow b$), but L cannot be accepted by a finite-state array acceptor, since $\{a^m b^m | i=1,2,...\}$ is not a finite-state string language (consider the arrays of height 1!). Thus the matrix languages in general (even for context-free G's) are not a subset of the finite-state array languages. Conversely, there are (deterministic, tape-bounded) finite-state array languages that are not matrix languages, for any choice of G. For example, the set L consisting of all square arrays is not a matrix language, since the termination decision in generating the columns does not depend on the array width. [The step numbers at which column termination can occur depend on the set of S_i's that initiated the columns; there are only finitely many such sets, and they cannot carry enough information to provide for a different termination step for every top row width.] Thus the matrix languages and the finite-state array languages are incomparable.

c) The matrix languages for which G is context-sensitive are a proper subset of the tape-bounded array languages. For example, the arrays whose columns

are all of the form $a^m b^m$ are a tape-bounded array language, but are not a matrix language (since $\{a^m b^m | m=1,2,...\}$ is not a finite-state string language). Conversely, a tape-bounded acceptor can scan the columns of a rectangular array, determine which S_i's could have initiated their generation, record these (sets of) S_i's in the top row, and then simulate a one-dimensional tape-bounded acceptor to verify whether some such set of S_i's could have been generated by G; thus any such matrix language is a tape-bounded array language. On the other hand, there are matrix languages (having non-context-sensitive top rows, and height 1) which are not tape-bounded array languages; thus the tape-bounded array languages and the matrix languages are incomparable. Finally, the matrix languages are a proper subset of the Turing rectangular array languages, by arguments similar to those already given.

It can be shown [9,12] that the languages of matrix grammars for which G is context-sensitive are the same as the languages of a special class of parallel-sequential array acceptors (Section of Chapter 7) -- namely, the class in which the cells (of the one-dimensional cellular acceptor) do not communicate with their neighbors until the acceptor has reached the last row of its input array.

A number of other grammar-like mechanisms can be used to generate rectangular array languages. In [10-11], for example, a grammar generates a string which specifies a sequence of horizontal and vertical concatenations to be performed on rectangular arrays. It is understood that when two arrays are horizontally (vertically) concatenated, they must have the same numbers of rows (columns); hence any such sequence of concatenation operations generates a rectangular array. Another way to insure rectangularity is to require that all border symbols of a given type (north, south, east, or west) be rewritten simultaneously; note that the column generation process in a matrix grammar is a special case of this (rewriting all south border symbols at the same time). Such models have been studied in [13]. We will not consider these approaches further here.

3. Array grammars

This section introduces grammars that generate or parse sets of connected arrays, and investigates their relationship to array acceptors. As we shall see, several problems arise with array grammars that did not arise for string grammars.

An array grammar must operate by replacing subarrays by subarrays, just as a string grammar replaced substrings by substrings. However, if the two subarrays (α and β, say) are not identical in size and shape, it is not clear how to replace α by β. One can presumably shift the rows or columns of the host array relative to one another so as to make room for β; but this may cause changes in symbol adjacencies arbitarily far away from β, since adjacent rows or columns may shift by different amounts. Thus applying a local rewriting rule $\alpha \to \beta$ may cause nonlocal changes in the host array, which seems undesirable. To avoid this problem, we shall require array grammars to be isometric (see Section 1.3); this means that for any rule $\alpha \to \beta$, α and β are geometrically identical, and the array grows (or shrinks) by rewriting (or creating) #s. For a treatment of non-isometric array grammars, see [14].

A second problem with array grammars is how to insure that they preserve connectedness (and nonemptiness) of the set of non-#s. Before formulating conditions for this, we first observe that in any rule $\alpha \to \beta$, α must contain non-#s, but they need not be connected. [In one dimension, as long

as the non-#s remain connected, a rule in which α has non-connected non-#s could never apply; but in two dimensions, a globally connected array can contain locally non-connected parts.] On the other hand, if α has more than one connected component of non-#s, every such component must touch the border of α. Our conditions on β are then as follows:

1) If the non-#s of α do not touch the border of α, then the non-#s of β must be connected (and non-empty).

2) Otherwise,

   a) Every connected component of non-#s in β must contain the border of some component of non-#s in α, and

   b) The border of every component of non-#s in α must be contained in some component of non-#s in β.

Proposition 3.1. If conditions (1-2) hold, applying the rule α→β does not disconnect or eliminate the non-#s.

Proof: We assume that non-#s existed and were connected before α→β was applied. In case (1), the non-#s of α must be the only non-#s that exist (there cannot be any outside α, since they would have to be connected to those in α, which would require that non-#s exist on the border of α); hence when we replace α by β, the non-#s of β are the only ones that exist, and they are connected by (1).

In case (2), we first show that any two non-#s p,q not in α remain connected when α is replaced by β. If there is a path of non-#s from p to q that does not meet α, this is clear. Otherwise, note that each time the path enters and leaves α, it meets only one component of non-#s in α, and by (2b) the border of this component is contained in a component of non-#s in β; hence each such segment of the path can be replaced by a segment in β.

Similarly, if p is not in α and q is in β, then by (2a) the component of non-#s in β that contains q also contains the border of some component of non-#s in α. Now p was connected (outside α) to this component, hence to its border; thus p is connected to q.

Finally, if p and q are both in β, and are not connected, they are both in components of non-#s that (by (2a)) contain borders of components of non-#s in α. But these components were connected outside α, hence so were their borders.//

It is clear that if (1) did not hold, applying $\alpha \to \beta$ would disconnect (or annihilate) the non-#s. If (2a) did not hold, a component of non-#s in β might be created that failed to be connected to the other non-#s; while if (2b) did not hold, some non-#s outside β might be disconnected from the other non-#s.

3.1. Array grammars and languages

Formally, an (isometric) array grammar is a 5-tuple $G = (V,V_T,P,S,\#)$, where all of these are defined just as in Section 1.2, except that P is a set of pairs of connected arrays (α,β), for all of which

a) α and β are geometrically identical

b) α does not consist entirely of #s

c) β satisfies conditions (1-2) of Proposition 3.1.

The language of G, L(G), is the set of all (non-null) connected arrays Σ on V_T such that Σ (embedded in an infinite array of #s) can be derived in G from the initial array, which consists of a single S (embedded in an infinite array of #s). [Derivations are defined exactly as in Section 1, except that they involve replacement of one subarray by another, rather than of one substring by another. Parsing is also defined just as in Section 1.] Note that by Proposition 3.1 and induction, the array resulting from any derivation always has its non-#s connected.

As in Section 1.1, we can (and shall from now on) assume that terminal symbols are never rewritten by any rule of G.

An array grammar is called monotonic if #s are never created by any rule -- in other words, for all rules $\alpha\rightarrow\beta$ of G, there are #s in β only in positions corresponding to #s in α. [We can also define an array grammar G to be "context-free" if, for all rules $\alpha\rightarrow\beta$ of G, α consists of a single nonterminal

symbol and (possibly) of #s; note that such rules do use #s as "context." Similarly, we can call G linear if G is "context-free" and, in addition, any β contains at most one nonterminal symbol; note that at any stage (except the last) of a derivation in such a G, exactly one nonterminal exists. As we shall see later, there does not seem to be a good analog of "finite-state" for array grammars.]

In some of the examples to be given later, it will be convenient to introduce some notation that allows us to compactly describe sets of array rewriting rules that differ by 90° rotations. We shall denote by $\alpha \rightarrow \beta$ (ROT) the set of four rules in which α and β are both rotated by 0°, 90°, 180°, and 270°. For example,

$$AB \rightarrow UV \text{ (ROT)}$$

denotes the four rules

$$AB \rightarrow UV \qquad \begin{matrix} B \\ A \end{matrix} \rightarrow \begin{matrix} V \\ U \end{matrix}$$

$$BA \rightarrow VU \qquad \begin{matrix} A \\ B \end{matrix} \rightarrow \begin{matrix} U \\ V \end{matrix}$$

Alternatively, we may denote by A(d)B the pair of adjacent symbols such that B is in direction d (=U,D,L, or R) from A; thus $A(R)B \equiv AB$, $A(L)B \equiv BA$, $A(U)B \equiv \begin{matrix} B \\ A \end{matrix}$, and $A(D)B \equiv \begin{matrix} A \\ B \end{matrix}$. In this notation, we may write

$$A(d)B \rightarrow U(d)V \qquad \text{for all } d \in \Delta \equiv \{U,D,L,R\}$$

instead of $AB \rightarrow UV$ (ROT).

3.2. Equivalence to array acceptors

Our goal in this section is to prove that (arbitrary, monotonic) array grammars generate the same classes of languares as (Turing, tape-bounded) array acceptors. The proofs are based on [15]; see also [16], which proves an analog of Theorem 3.3 using a somewhat different definition of an array grammer. We also discuss the relationship between finite-state array grammars and array acceptors; this discussion is based on [17].

Theorem 3.2. The languages generated by array grammars are the same as the languages accepted by Turing array acceptors.

Proof: Given an array grammar G, we define a TAA, A, that accepts exactly L(G) as follows: Given an array Σ of non-#s in the terminal vocabulary V_T of G, embedded in an infinite array of #s, A moves around nondeterministically, starting at some point of Σ, and rewrites each symbol x (possibly #) that it comes to as (x,#).* Finally, at some step, A rewrites some x as (x,S); this can only happen once. After it happens, A begins to simulate rules of G on the second terms of pairs; pairs with no second term are regarded as having second term #.

---

*This is done in order to insure that A is always on a connected set of non-#s; if A moved onto the #s without rewriting them as pairs, it might be unable to find Σ again.

Suppose that at some step A decides to apply the rule $\alpha \to \beta$ of G. It then searches (nondeterministically) for one of the non-# symbols in α (as a second term), marks it, checks (deterministically) for the other symbols as second terms (these are in known positions relative to the given symbol), marks them, and when they have all been found, rewrites these second terms as the corresponding symbols in β; after this is done, A is ready to simulate another rule. (While searching, A continues to rewrite x's as (x,#)'s.) At any stage after applying a rule of G, A can systematically scan the array of non-#s (Proposition 4.1 of Chapter 4) and check whether (a) every non-# is a pair; (b) the second term of every pair is the same as its first term. [A can also check, during some such scan, that the non-# first terms are in fact all in V_T.] If this is found to be true, A accepts Σ; otherwise, A can resume the simulation of G. Evidently A accepts Σ iff. a derivation in G exists that generates a copy of Σ; thus A accepts exactly L(G) (from some starting point).

Conversely, given A, we define a grammar G that generates exactly L(A) as follows: Starting with an S on a background of #s, G begins by generating an arbitrary array Σ of triples of the form (x,x,0) or (x,x,1), where x is any non-# symbol in the vocabulary V of A, and where there is always exactly one 1. This can be done using the rules

1a) $S \rightarrow (x,x,1)$

1b) $(x,x,1)\# \rightarrow (x,x,0)(y,y,1)$ (ROT)

1c) $(x,x,1)(y,y,0) \rightarrow (x,x,0)(y,y,1)$ (ROT)

1d) $(x,x,1) \rightarrow (x,x,(q_0,N))$

for all $x, y \neq \#$ in V.

When (1d) is used, the 1 turns into a pair (q_0,N) representing the initial state and (fictitious) "previous move direction" of A.

Next, G simulates the operation of A on the array of second terms of the triples; the first terms remain unaffected. This is done using the rules (for all x,u,v in V).

2a) $(x,y,(q,d)) \rightarrow (x,z,(q',N))$ for all $(q',z,N) \in \delta(q,y,d)$

2b) $(x,y,(q,d))(d')(u,v,0) \rightarrow (x,z,0)(d')(u,v,(q',d'))$
for all $(q',z,d') \in \delta(q,y,d)$

2c) $(x,y,(q,d))(d')\# \rightarrow (x,z,0)(d')(\#,\#,(q',d'))$ (similarly)

where δ is the transition function of A, and provided that q is not an accepting state of A.

Finally, if the simulation of A enters an accepting state, the triples are turned into their first terms using the rules

3a) $(x,y,(q,d)) \rightarrow x$ for all $x,y, \in V$ and all $q \in Q_A$

3b) $(u,v,0)x \rightarrow ux$ (ROT) for all $u,v,x \in V$.

Since V-{#} is the terminal vocabulary of G, this means that G generates a terminal array of non-#s (the first terms of Σ) iff. A accepts this array (from some starting point).//

Theorem 3.3. The languages generated by monotonic array grammars are the same as the languages accepted by tape-bounded array acceptors.

Proof: In the first part of the proof of Theorem 3.2, note that at any step of a derivation of Σ in the monotonic array grammar G, the non-#s are always a subset of Σ, since #s can never be created by G. Thus, we can simulate G using a tape-bounded A that always remains on Σ. [A problem arises with this simulation if a rule $\alpha \rightarrow \beta$ of G uses #s as context (i.e., there are #s in α which are also in the corresponding positions in β); these #s may be outside Σ, but A must verify their presence in order to apply the rule. This can be done using the array scanning process described just prior to Proposition 4.6 of Chapter 4. Suppose that A is located at position (i,j) and wants to verify that there is a # in position (i',j'), where the differences i-i' and j-j' are known to A. To do this, A marks (i,j), finds the outer border of Σ, and scans Σ row by row. When A reaches the row with the marked point, it returns to the outer border and follows it, using the positions of markers to keep track of its net up and down moves. If the outer border never reaches row i', Σ cannot intersect that row, so that position (i',j') is certainly #. Otherwise, A marks the points of the outer border that are on row i', and scans these rows of Σ, marking their points. Analogously, using a column by

column scan, A finds the marked point and checks whether the outer border reaches column j'. If not, (i',j') is #; if so, A marks the points of Σ that are on column j'. If these marks ever encounter the marks on row i', (i',j') is not #; otherwise, it is #, and A can rescan Σ, erase the marks on row i' and column j', rescan Σ and return to the marked point (i,j). In this way, A can verify that #s exist in given positions relative to some non-# point of α, and thus check that all of α is present, without having to leave Σ. Note that a similar process may be necessary for the rewriting of α, since the non-#s of α may be separated by #s, and A may have to compute the positions of these non-#s in order to locate them and rewrite them.] After any step of the simulation, A can scan Σ and check that the first and second terms are all the same (and all in V_T). If so, A accepts; otherwise, it continues the simulation. Thus A accepts exactly L(G), as in the proof of Theorem 3.2.

In the second part of the proof, note that the only rules of G that can create #s are the terminating rules 3(a-b); and these create #s only if triples with # first terms were created by the simulation rule (2c). But if A is tape-bounded, we can modify (2c) to have the form

$$(x,y,(q,d))(d')\# \rightarrow (x,z,(q'',d'^{-1}))(d')\#$$

where q" is a possible state of A after it has bounced back off the #. Thus when A is tape-bounded, G need never create #s, so that G can be monotonic.//

It does not seem to be easy to define a class of array grammars whose languages are the same as those of the finite-state array acceptors. (A generator model that generates exactly the FSA languages is described in [18]. However, in this model, every time a given symbol is rewritten, it must be rewritten in the same way, even though the generation process cannot tell whether or not this is happening; thus this model requires some means, outside the generator itself, to reject arrays in which a rewriting inconsistency occurs.) Of course, one can always define a class of array grammars that are capable only of simulating FSA's [19]; but this is artificial.

In one dimension, the linear grammars (in which, at any step of a derivation except the last step, exactly one nonterminal symbol exists) generate more than the finite-state languages (e.g., $\{a^m b^m | m=1,2,...\}$ is a linear string language, having the grammar whose rules are $S \to aSb, S \to ab$); whereas the "right linear" grammars, in which (until the last step) the sole nonterminal symbol is always at the right end of the string, generate exactly the finite-state languages (Theorem 1.4). In two dimensions, there is no natural analog of "right linear." Moreover, we shall now show that the languages of linear array grammars are incomparable with the FSA languages.

We first exhibit an FSA language that cannot be generated by a linear array grammar. Consider the set of thin,

upright T-shaped arrays of x's on a background of #s. This set is accepted by a deterministic, tape-bounded FSA which operates as follows: It moves up until it hits a #, then moves left until it hits a # again. It then verifies that it is on a row of x's that have #'s above and below them, with exactly one exception that has an x below it and that is not at either end of the row. Finally, it moves back to the exceptional x and moves downward, verifying that it is on a column of x's that have #s on their left and right. When it reaches the bottom of this column, if all these verifications have been carried out successfully, it accepts.

We now show that the set of T's cannot be generated by a linear array grammar. Suppose G were such a grammar; let the greatest diameter of any right hand side of a rule of G be k. Since the T's can be arbitrarily large, it is clear that G must generate the ends of the arms of a large T at different steps in the derivation of that T. Let these arm ends be a, b, and c. One of them, say a, can be generated before the point P at the T-junction is generated, but b and c must be generated afterwards. After P is generated, if the nonterminal moves sufficiently far (>k) down the b arm, it can never generate the c arm, since it cannot return down the b arm (which now consists of terminals) or cross the #s (which, once rewritten, would destroy the shape of the desired thin T).

Conversely, we can exhibit a language that is generated by a linear array grammar but is not accepted by a tape-bounded FSA. Let G be the array grammar whose rules are

$$S\# \to xS \quad ; \quad S\# \to xT$$

$$\begin{matrix} T \\ \# \end{matrix} \to \begin{matrix} x \\ T \end{matrix} \quad ; \quad \begin{matrix} T \\ \# \end{matrix} \to \begin{matrix} x \\ U \end{matrix}$$

$$\#U \to Ux \quad ; \quad \#U \to Vx$$

$$\begin{matrix} \# \\ V \end{matrix} \to \begin{matrix} V \\ x \end{matrix} \quad ; \quad V \to x$$

Informally, the S moves to the right, leaving a trail of x's, until it changes to a T; the T moves down, trailing x's, until it changes to a U; the U moves left, leaving x's behind it, until it changes to a V; and the V moves up, trailing x's, until it changes to an x. Thus the language of this grammar consists of four-sided upright rectangular arcs, composed of x's, which may touch themselves, but cannot cross themselves. In particular, all the hollow, thin upright rectangles of x's are in this language. But as we have seen (in the proof of Theorem 4.3 of Chapter 4), any tape-bounded FSA that accepts all such rectangles must also accept various sufficiently large rectangular spirals (having many more than four sides). Hence the language of G is not a tape bounded FSA language.

Another possibility, discussed in [17,19], is to allow a linear grammar to rewrite terminal symbols, but not to sense what symbols are present; for example, in a rule

$\alpha \rightarrow \beta$, α would consist of a single nonterminal A together with a set of "don't-care" symbols, and in applying the rule we would rewrite whatever symbols lie in those positions relative to A. [If we do allow the grammar to sense what symbols are present, it is at least as strong as an ordinary linear array grammar, hence can generate a class of patterns that is not an FSA language, as in the preceding paragraph.] If we do this, it becomes possible to generate thin T-shaped patterns (possibly including degenerate T's with arms missing); we omit the details here. On the other hand, such a "blind" grammar could not generate the set of arcs (composed of x's) that do not cross or touch themselves, since the nonterminal leaving the trail of x's could not know when it was about to cross its own path; but this set is evidently an FSA language (the FSA verifies that all but two of the non-#s have exactly two non-# neighbors). Conversely, a "blind" grammar can generate the set of all connected arrays of x's; but this is not an FSA language. [Proof: It contains the set of all thin, hollow rectangles of x's; but if A accepts such a rectangle, it also accepts a sufficiently large rectangular spiral without visiting its ends, so that it cannot tell whether or not there are symbols other than x at the ends.]

3.3 Parallel array grammars

In Section 1.4, we introduced the idea of parallel rule application (rewrite all instances of α as β's, rather than just a single instance), and showed that the classes of languages generated in parallel by (arbitrary, context-sensitive) grammars are the same as the classes of languages generated by those types of grammars in the ordinary way. The proofs apply also to isometric grammars; in fact, in the proof that any sequential language is a parallel language, if G is isometric, so is G*, while in the proof that any parallel language is a sequential language, we know that an isometric G' can simulate an automaton of the desired type that applies the rules of G.

Analogous results can be established for (isometric) array grammars; the details are straightforward [20]. One can also define (isometric) array L-systems (the details will be omitted here); as pointed out in Section 1.4, they are essentially nondeterministic cellular automata.

References

1. J. E. Hopcroft and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, MA, 1969.

2. A. Rosenfeld, Iostonic grammars, parallel grammars, and picture grammars, in B. Meltzer and D. Michie, eds., Machine Intelligence 6, Edinburgh University Press, 1971, 281-294.

3. R. Siromoney and K. Krithivasan, Parallel context-fee languages, Information and Control 24, 1974, 155-162.

4. S. Skyum, Parallel context-free languages, ibid. 26, 1974, 280-285.

5. G. T. Herman and G. Rozenberg, Developmental Systems and Languages, North-Holland, Amsterdam, 1974.

6. G. Rozenberg and A. Salomaa, L-Systems, Springer, Berlin, 1974.

7. A. Lindenmayer and G. Rozenberg, eds., Automata, Languages, Development, North-Holland, Amsterdam, 1976.

8. A. Walker, Dynamically stable strings in developmental systems, Journal of Computer and System Sciences 11, 1975, 252-261.

9. G. Siromoney, R. Siromoney, and K. Krithivasan, Abstract families of matrices and picture languages, Computer Graphes and Image Processing 1, 1972, 284-307.

10. G. Siromoney, R. Siromoney, and K. Krithivasan, Picture languages with array rewriting rules, Information and Control 22, 1973, 447-470.

11. G. Siromoney, R. Siromoney, and K. Krithivasan, Array grammars and Kolam, Computer Graphics and Image Processing 3, 1974, 63-82.

12. E. D. Feldman, Matrix grammars and parallel-sequential array automata, Technical Report 250, Computer Science Center, University of Maryland, College Park, MD, 1973.

13. R. Siromoney and G. Siromoney, Extended controlled table L-arrays, Information and Control 35, 1977, 119-138.

14. P. A. Ota, Mosaic grammars, Pattern Recognition 7, 1975, 61-65.

15. D. L. Milgram and A. Rosenfeld, Array automata and array grammars, Information Processing 71, North-Holland, Amsterdam, 1972, 69-74.

16. E. Yodokawa and N. Honda, On 2-dimensional pattern-generating grammars, Systems/Computers/Control 1, 1970, 6-13.

17. A. Rosenfeld, Some notes on finite-state picture languages, Information and Control 31, 1976, 177-184.

18. J. Mylopoulos, On the application of formal language and automata theory to pattern recognition, Pattern Recognition 4, 1972, 37-51.

19. A. Rosenfeld and D. L. Milgram, Array automata and array grammars, 2, TR-171, Computer Science Center, University of Maryland, College Park, Md., 1971.

20. A. Rosenfeld, Array grammar normal forms, Information and Control 23, 1973, 173-182.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 78-0353	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ARRAY GRAMMARS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER TR-629
7. AUTHOR(s) Azriel Rosenfeld		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3271
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Center University of Maryland College Park, Md. 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB Washington, D. C. 20332		12. REPORT DATE January 1978
		13. NUMBER OF PAGES 52
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Formal languages
Picture languages
Array grammars

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is a draft of Chapter 8 of a forthcoming book on Picture Languages. [Drafts of Chapter 2, on digital topology, and Chapters 3-5, on sequential and parallel picture acceptors, were issued earlier as TR's 542 and 613, respectively.] This chapter deals with array grammars, with emphasis on their relationship to acceptors; some preliminary material on string grammars is also included. Comments on the choice and treatment of the material are invited.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED