CONTEXT-FREE LANGUAGES REVISITED YET AGAIN

Ian C. Stirk

Introduction

At the beginning of an interesting paper, Gazdar (1982) points out some misconceptions about the formal properties of grammars which have been current in the linguistics community. One of these is that context-free grammars may never have rules which rewrite non-terminal symbols as zero. Another is that it makes no significant difference whether the rules of grammars are used to generate derivations, or used as node admissibility conditions for trees.

The first of these misconceptions can be cleared up without any very complex argument. The language $a^n b^n$ (that is, the language whose grammatical strings consist of a number of a's followed by the same number of b's) can be generated by a context-free grammar:

$$\begin{aligned} A_0 &\rightarrow A_1\ A_3 \\ A_3 &\rightarrow A_0\ A_2 \\ A_0 &\rightarrow A_1\ A_2 \\ A_1 &\rightarrow a \\ A_2 &\rightarrow b \end{aligned}$$

Some of the symbols in this grammar are redundant. A_1 and A_2, for instance, can only ever be rewritten as a and b respectively, so they might as well be replaced by those terminal symbols wherever they occur:

$$\begin{aligned} A_0 &\rightarrow aA_3 \\ A_3 &\rightarrow A_0b \\ A_0 &\rightarrow ab \end{aligned}$$

Now we notice that A_3 never dominates more than A_0b, so it might as well be replaced by A_0b itself, leaving:

Grammar 1 $$\begin{aligned} A_0 &\rightarrow aA_0b \\ A_0 &\rightarrow ab \end{aligned}$$

Since this grammar will be a useful example later, too, it has been given the label "grammar 1". Getting to that simple grammar 1 illustrates an important principle about context-free grammars, namely that the only non-terminal symbols they need to have are those directly involved in recursion, like A in this one, provided that we are only interested in their weak generative capacity. Other symbols can just be replaced by whatever it is they are rewritten as in the rules. This principle is, I believe, due to Bach in earlier versions of his (1974). It applies only to context-free grammars, of course, since in context-sensitive ones certain non-terminal symbols may be needed as context. The principle may be immediately applied to the problem of rewriting symbols as zero. If such a non-terminal symbol is rewritten only as zero, then of course it can be replaced by zero everywhere: deleted, in other words. If it may also be rewritten as something more substantial, it can be replaced by that other thing. This will certainly take care of non-terminal symbols not involved in recursion, but suppose a recursive symbol is rewritten as zero by some rule? Notice first that grammar 1 could also be expressed in this way:

$$\begin{aligned} A_0 &\rightarrow aA_0b \\ A_0 &\rightarrow \lambda \end{aligned}$$

The symbol λ is frequently employed in mathematical linguistics to represent the null string, and I will use it so. It is apparent that if X is a recursive symbol, the grammar must contain a rule of the form $X \rightarrow \alpha X \beta$. If it also contains $X \rightarrow \lambda$, this latter rule can just be replaced by $X \rightarrow \alpha\beta$, without making any difference to generative capacity. This argument takes care of the first misconception: languages still remain context-free even if their grammars contain rules rewriting symbols as zero.

The same principle also provides simple informal proofs of such facts as that the language $a^n b^n c^n$ cannot be context-free. If it were, then there would be a grammar for it whose only non-terminal symbols were involved in recursion. Consider some derivation generated by this grammar. Its penultimate line must contain just one non-terminal symbol which is rewritten to form the last line $a^n b^n c^n$, for some n. Since the non-terminal symbol which is replaced is involved in recursion, then that penultimate line could also be a line in other derivations where the symbol is used recursively. All of these would have to end in grammatical strings, too, of course. Thus in that penultimate line, the non-terminal symbol could not be among the a's, b's or c's because any recursion would upset the numbers of them, which must stay the same. If it was between the a's and the b's, then the number of a's and b's could be kept the same, but the c's would be left out. Similarly if it lay between the b's and the c's. Thus $a^n b^n c^n$ cannot be context-free. It is clear too that if the strings of a language contain some point of symmetry, such as the one between the a's and the b's in $a^n b^n$, then the derivations of a grammar all of whose non-terminal symbols were recursive would have to contain non-terminal symbols at those points of symmetry. Otherwise recursion would upset the symmetry on one side or the other. Grammar 1 illustrates this.

One of the misconceptions can thus be cleared up without invoking any mathematically sophisticated argument. The other one, about node admissibility conditions, is not so easy to dispose of. Gazdar (1982) refers to Joshi and Levy (1977) as proof of his contention. This paper contains a long and complex mathematical argument, and is therefore guaranteed to miss an important audience of linguists. Of course mathematical proofs may be taken for granted by those who do not wish to delve into them, but in that case some of their important implications may be missed. After a great deal of trial and error, I came up with a fairly straightforward informal proof, which is presented in the next section. Also I hope to show that it could be formalized to provide an alternative proof to Joshi and Levy's: after all, informality must not mean loss of rigour.

It also seems to me that the method used in this proof has some interesting linguistic applications. I have tried to say what they are in a purely speculative conclusion.

Some Theorems

For reasons which will become clear below, it is often convenient to express context-free grammars in what is known as *Chomsky normal form*. One requirement of this is that terminal symbols be introduced only by rules of the form $X \rightarrow c$. Grammar 1 above can easily be made to fulfil this requirement by introducing two new non-terminal symbols A and A, thus:

Grammar 2
$$\begin{array}{lcl} A_0 & \rightarrow & A_1A_0A_2 \\ A_0 & \rightarrow & A_1A_2 \\ A_1 & \rightarrow & a \\ A_2 & \rightarrow & b \end{array}$$

The other requirement is that the right-hand side of any rule, if not a single terminal symbol, should consist of two non-terminal symbols. The rule $A_0 \rightarrow A_1A_0A_2$ obviously violates this, but it is simple enough to introduce a further symbol A_3 and break this rule into two, $A_0 \rightarrow A_1A_3$ and $A_3 \rightarrow A_0A_2$, which obviously have the same weak generating capacity as the single original rule. In fact, a rule with n symbols on its right-hand side, such as $X \rightarrow Y_1Y_2....Y_n$, could be replaced by (n−1) rules as follows:

$$X \rightarrow Y_1Z_1$$

$$Z_1 \rightarrow Y_2Z_2$$
$$\vdots$$
$$Z_{n-1} \rightarrow Y_{n-1}Y_n$$

where $Z_1 \ldots Z_{n-1}$ are new non-terminal symbols. This means that any context-free grammar can be put into Chomsky normal form. The simple example of grammars 1 and 2 will finally become

Grammar 3
$$\begin{aligned} A_0 &\rightarrow A_1A_3 \\ A_3 &\rightarrow A_0A_2 \\ A_0 &\rightarrow A_1A_2 \\ A_1 &\rightarrow a \\ A_2 &\rightarrow b \end{aligned}$$

This is familiar from the introduction! A typical labelled bracketing derived from this grammar would look like this:

String 1 $[_{A_0} [_{A_1} a]_{A_1} [_{A_3} [_{A_0} [_{A_1} a]_{A_1} [_{A_2} b]_{A_2}]_{A_0} [_{A_2} b]_{A_2}]_{A_3}]_{A_0}$

The labelled bracketing, of course, gives the same information as a tree structure, but seeing it in linear form might lead one to wonder whether a finite state grammar could generate similar strings: a grammar containing rules like $X \rightarrow [_2 Y$, for instance. The problem is, of course, that a finite state grammar cannot incorporate an indefinitely large "memory" for previous parts of a string as it is generated, whereas every left-hand bracket in something like string 1 above must be correctly paired with a right-hand one, no matter how many other symbols intervene. Notice that correctly paired strings of brackets, such as the familiar ( ), [ ], { }, may be generated by this particularly simple context-free grammar:

Grammar 4 $X \rightarrow XX$, $X \rightarrow (X)$, $X \rightarrow [X]$, $X \rightarrow \{X\}$, $X \rightarrow \lambda$

All such bracketings as (([ ]){( )}) are possible terminal strings, which is fine so long as the "contents" of brackets are ignored, as well as any relations between the different kinds. Grammars like the above are known as Dyck grammars, and in general have the form:

Grammar 5 $X \to XX$, $X \to cXc'$, $X \to \lambda$

where $X \to cXc'$ is a rule schema: one such rule is needed for each pair of terminal symbols c, c′.

These considerations lead one to wonder what could be done with the *intersection* of a Dyck language and a finite state language. Languages are regarded as sets of sentences, so the term intersection can be used in its usual set-theoretical sense: in this case; to indicate the set of sentences common to both languages. The Dyck grammar would ensure that all the sentences in the set were correctly bracketed, while the finite state grammar would take care of linear dependencies. In the example we have been using, the fact that each $]_{A_3}$ is followed by a $]_{A_0}$ would be an example of one such dependency. Unfortunately, the finite state grammars required for this turn out to be intolerably complex, if not impossible to write.

But is it really necessary to generate a complete labelled bracketing? Maybe a partial bracketing would be enough, provided it contained enough information for reconstructing the complete one. In fact provided we know that a bracketing was generated by a grammar in Chomsky normal form, a knowledge of the left-hand brackets alone is sufficient, as may be illustrated by first deleting all right-hand ones from String 1, and then showing that there is only one way they could be reinserted:

String 2 $[_{A_0} [_{A_1} a \quad [_{A_3} [_{A_0} [_{A_1} a \quad [_{A_2} b \qquad [_{A_2} b$

Right-hand brackets must obviously be placed after each terminal symbol, so we immediately obtain:

String 3 $[_{A_0} [_{A_1} a]_{A_1} [_{A_3} [_{A_0} [_{A_1} a]_{A_1} [_{A_2} b]_{A_2} \qquad [_{A_2} b]_{A_2}$

The $[_{A_1}$ a $]_{A_1}$ and the $[_{A_2}$ b $]_{A_2}$ immediately after the $[_{A_0}$ bracket form two constituents, the only number allowed by the Chomsky normal form. This means that a $]_{A_0}$ must be placed immediately after the $[_{A_2}$ b $]_{A_2}$:

String 4 $\quad [_{A_0} [_{A_1} a]_{A_1} [_{A_3} [_{A_0} [_{A_1} a]_{A_1} [_{A_2} b]_{A_2}]_{A_0} [_{A_2} b]_{A_2}$

In a similar way, two constituents are now discernible after the $[_{A_3}$ bracket, so $]_{A_3}$ must be placed:

String 5 $\quad [_{A_0} [_{A_1} a]_{A_1} [_{A_3} [_{A_0} [_{A_1} a]_{A_1} [_{A_2} b]_{A_2}]_{A_0} [_{A_2} b]_{A_2}]_{A_3}$

This leaves only the final $]_{A_0}$ to go in, and string 1 is restored:

String 1 $\quad [_{A_0} [_{A_1} a]_{A_1} [_{A_3} [_{A_0} [_{A_1} a]_{A_1} [_{A_2} b]_{A_2}]_{A_0} [_{A_2} b]_{A_2}]_{A_3}]_{A_0}$

Obviously a quite automatic process. Going back now just to the lefthand brackets of string 2,

String 2 $\quad [_{A_0} [_{A_1} a \quad [_{A_3} [_{A_0} [_{A_1} a \quad [_{A_2} b \quad\quad [_{A_2} b$

we see that these themselves form pairs. Because of the rule of the context–free grammar $A_3 \rightarrow A_0\, A_2$, a $[_{A_0}$ bracket must be succeeded later by a $[_{A_2}$, and so on. If the symbols x_2 and x_2' are chosen instead of $[_{A_0}$ and $[_{A_2}$, with similar substitutions for the other brackets paired by the rules of grammar 3, the above string might appear as:

String 6 $\quad x_1$ a $x_1'\, x_2 x_3$ a x_3' b x_2' b

Now here the various x_i and x_i' are paired in a manner suitable for the terminal string of a Dyck grammar, but the symbols a and b come singly. This can easily be remedied by a trick: each "a" is simply replaced by "aa′ " and each "b" by "bb′ ". Naturally a′ and b′ are to be new terminal symbols. String 6 now becomes

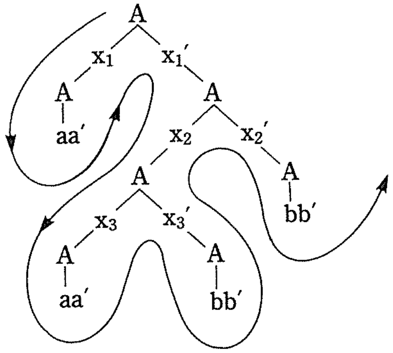
String 7 $x_1\ aa'\ x_1'\ x_2x_3\ aa'\ x_3'\ bb'\ x_2'\ bb'$

which is of course a terminal string of the following Dyck grammar

Grammar 6 $X \rightarrow \lambda,\ \ X \rightarrow XX,\ \ X \rightarrow aXa',\ \ X \rightarrow bXb',\ \ X \rightarrow x_1Xx_1',$
$X \rightarrow x_2Xx_2',\ \ X \rightarrow x_3Xx_3'$

The problem now is to find a finite–state grammar which will interact suitably with grammar 6, so that the intersection of their languages will be the language generated by the original context–free grammar 3. It is helpful here to express string 7 in tree form:

Tree 1



This tree structure enables us to see in rather an insightful way the relation between strings like string 7 and the context–free grammar 3. It will be seen that here the x_i and x_i', which derived originally from left–hand brackets, have been used to label certain branches of the tree, in fact those branches which come from a "fork" in the tree. The order of elements in string 7 follows a path through the tree, indicated by the arrowed curve. The initial "x_1aa'" takes us from the root of the tree down the left most branch to the leaf "aa′". After this we retrace our steps to the fork of the tree, and set off down the next branch, labelled "$x_1'\ x_2\ x_3$...." in the string. By the time we come to the end of the string, every part of the tree has been visited. This makes it clear how the two dimensional tree structure is effectively "linearised" in string 7.

With this in mind, we can devise a suitable finite–state grammar. It is convenient to use the same labels for non–terminal symbols as are used in the context–free grammar, for a glance at tree 1 shows that for each context–free rule of the form "$A_0 \rightarrow A_l A_m$" there should be a finite–state one "$A_0 \rightarrow x_i A_l$", in order to start us off correctly down the leftmost branch of any tree. In fact for the left–hand branch of any fork, there must be a rule "$A_r \rightarrow x_i A_s$" to correspond to the context–free "$A_r \rightarrow A_s A_t$". A glance at grammar 3 again shows that the finite–state grammar should have these rules in it, in that case:

$$A_0 \rightarrow x_1 A_1, \quad A_3 \rightarrow x_2 A_0, \quad A_0 \rightarrow x_3 A_1$$

In order to get the leaves of the trees right, these rules will also obviously be necessary:

$$A_1 \rightarrow aa', \quad A_2 \rightarrow bb'$$

It's not so easy to see what needs to be done next. In tree 1, we see that in tracing the path through, it is necessary to jump from each leaf to the fork above, before continuing down a right–hand branch. This applies until the rightmost leaf, when the path is complete. This consideration suggests that these rules will find a place in the grammar:

$$A_1 \rightarrow aa' x_1' A_3, \quad A_1 \rightarrow aa' x_2' A_2, \quad A_1 \rightarrow aa' x_3' A_2,$$
$$A_2 \rightarrow bb' x_1' A_3, \quad A_2 \rightarrow bb' x_2' A_2, \quad A_2 \rightarrow bb' x_3' A_2$$

The finite–state grammar is now complete, and we can set it out as:

Grammar 7
$$\begin{array}{lll} A_0 & \rightarrow & x_1 A_1 \\ A_3 & \rightarrow & x_2 A_0 \\ A_0 & \rightarrow & x_3 A_1 \\ A_1 & \rightarrow & aa' x_1' A_3 \\ A_1 & \rightarrow & aa' x_2' A_2 \end{array}$$

$$\begin{aligned}
A_1 &\rightarrow aa'x_3'A_2\\
A_2 &\rightarrow bb'x_1'A_3\\
A_2 &\rightarrow bb'x_2'A_2\\
A_2 &\rightarrow bb'x_3'A_2\\
A_1 &\rightarrow aa'\\
A_2 &\rightarrow bb'
\end{aligned}$$

Experiment should convince the reader that strings that can be generated both by this grammar and grammar 6 all contain, among other things, a number of occurrences of "a" followed by the same number of occurrences of "b". The "other things", of course, are the x_i and x_i', as well as a′ and b′. If these could just be struck out, we would end up with just the same strings as the context-free grammar 3 generates. The striking out may be accomplished by a mechanism known, for group-theoretical reasons, as a "homomorphism". This may be regarded as a function from strings into other strings. Calling the function "h", it is to have the property that, when "α" and "β" are strings,

$$h(\alpha\beta) = h(\alpha)\, h(\beta)$$

The particular function we require is defined by:

$h(c)=c$, where "c" is either "a" or "b"
$h(c)=\lambda$, otherwise

Obviously, $h(x_1aa'\ x_1'\ x_2x_3aa'\ x_3'\ bb'\ x_2bb') = aabb$, and this homomorphism will work in general to reduce strings generated by grammars 6 and 7 to terminal strings of grammar 3.

It should be clear by now that the method we have been developing can be generalized to apply to any context-free grammar in Chomsky normal form. The recipe for determining the finite-state grammar is:

(1) Include a rule $X_i \rightarrow x_iY_i$ for every rule $X_i \rightarrow Y_iZ_i$ of the context-free grammar, where x_i is a new terminal symbol.
(2) Include rules of the form $X \rightarrow cc'\ x_i'\ Z_i$ for every rule of the context-free grammar of the form $X \rightarrow c$, and for every rule of the form $X_i \rightarrow Y_iZ_i$.

(3) Include a rule $X \rightarrow cc'$ for each rule of the form $X \rightarrow c$ in the context-free grammar.

The homomorphism can also be easily extended:

$h(c) = c$, where "c" is a terminal symbol of the context-free grammar;

$h(c) = \lambda$, otherwise.

These final results may be compared with what Salomaa (1973, p69) labels Theorem 7.4, and the proof which follows it. For convenience, I repeat Salomaa's statement of the theorem, but call it Theorem 1.

Theorem 1:

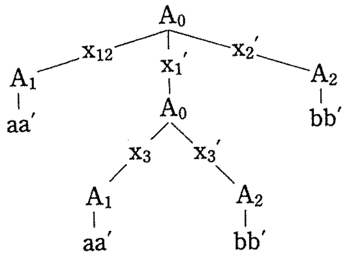
Every context-free language equals a homomorphic image of the intersection of a regular language and a Dyck language.

"Regular language" is synonymous with "finite-state language".

Theorem 1 has some importance in mathematical linguistics, and has been used by Salomaa (1971) to give a mathematically far more elegant proof of the Peters-Ritchie result on the generative power of transformational grammars than that in Peters and Ritchie (1973) .

The method of constructing appropriate finite-state languages given above depends of course on the context-free grammar being given in Chomsky normal form. It is possible to dispense with this condition, if only something a little more complex than a Dyck language is used as the set to be intersected. Grammar 2 will serve quite well as an illustration of this. A typical tree structure derived from this grammar would be:

Tree 2



Various branches are already labelled with x_i and x_i' in anticipation. The tree divides into three at the root, and it will be seen that the leftmost branch is labelled x_{12}, to suggest that it has *two* "companions", x_1' and x_2'. Rules for the finite-state grammar are not hard to devise, following a similar recipe to the one given above:

Grammar 8	A_0	$\rightarrow$	$x_{12}A_1$
	A_0	$\rightarrow$	x_3A_1
	A_1	$\rightarrow$	$aa'x_3'A_2$
	A_1	$\rightarrow$	$aa'x_1'A_0$
	A_1	$\rightarrow$	$aa'x_2'A_2$
	A_2	$\rightarrow$	$bb'x_3'A_2$
	A_2	$\rightarrow$	$bb'x_1'A_0$
	A_2	$\rightarrow$	$bb'x_2'A_2$
	A_1	$\rightarrow$	aa'
	A_2	$\rightarrow$	bb'

Clearly one derivation from these rules would be the string:

String 8 $\quad x_{12}aa'\ x_1'\ x_3aa'\ x_3'\ bb'\ x_2'bb'$

If the rule $X \rightarrow x_{12}Xx_1'\ Xx_2'$ were added to a Dyck grammar, to give:

Grammar 9 $\quad X \rightarrow \lambda,\ X \rightarrow XX,\ X \rightarrow aXa',\ X \rightarrow bXb',\ X \rightarrow x_3Xx_3',\ X \rightarrow x_{12}Xx_1'Xx_2'$

then strings like string 8 could be generated by it. It is not hard to see that the intersection of the languages generated by grammars 8 and 9 is just the context-free language we want. The method we have used can clearly be generalized. For any context-free rule of the form $X_k \rightarrow W_kY_kZ_k$, then a finite-state rule $X_k \rightarrow x_{ij}W_k$would be needed, together with rules of the form $X \rightarrow cc'\ x_i'\ Y_k$ and $X \rightarrow cc'\ x_j'\ Z_k$ for every context-free rule of the form $X \rightarrow c$. Furthermore, the Dyck grammar needs supplementing with a rule $X \rightarrow x_{ij}$

$Xx_i'\,Xx_j'$. In future, such grammars will be called "extended Dyck grammars". The process of generalization need not stop here: no matter how many symbols there are on the right-hand side of some context-free rule, we may accommodate them with extra terminal symbols of the form $x_{ij....k}$ and its "companions" x_i', x_j' etc.

Occasionally rules of the form $X \rightarrow Y$, with only one non-terminal symbol on the right-hand side may be found in context-free grammars, for the purpose of establishing grammatical classes. To deal with this in our system, we would need an x_i with no "companion" at all, just so that a branch labelled "x_i" would appear in the reconstructed tree-structure. This is possible too, if we just include rules like $X \rightarrow x_i$ in the extended Dyck grammar. And in that case, we could drop such terminals as "aa′ " in favour of a rule $X \rightarrow a$, and so on. For instance, grammar 8 above could be altered by removing each occurrence of "a′ " and "b′ ", while grammar 9 would become:

Grammar 10 $X \rightarrow XX$, $X \rightarrow a$, $X \rightarrow b$, $X \rightarrow x_3Xx_3'$, $X \rightarrow x_{12}Xx_1'\,Xx_2'$

Notice that the rule $X \rightarrow \lambda$ become superfluous: there will no longer be "superfluous" X's that need removal.

How far could Dyck grammars be extended, by adding different sorts of rules? It is not difficult to prove a theorem which goes some way towards answering this question:

Theorem 2 :

The intersection of a context-free language and a finite-state language is context-free.

This theorem is essentially the one to be found in Salomaa (1973, p59) as Theorem 6.7. I will give an informal presentation of Salomaa's proof.

Consider some string belonging to the intersection of the two languages, of this form:

$$a_1a_2 \ldots\ldots a_n$$

where the a_i are terminal symbols (not necessarily distinct). Since this typical string is generated by a finite-state grammar as well as a context-free one, we could think of one of those familiar state diagrams:

$$\begin{tikzcd}
s_0 \arrow[r, "a_1"] & s_1 \arrow[r, "a_2"] & s_2 \arrow[r, dotted] & s_n
\end{tikzcd}$$

The underlying idea of the proof is actually to construct a context-free grammar that will generate only the intersection of the two languages. To be sure that every terminal string could be generated by the finite-state grammar too, state diagrams are actually built into the form of the rules of the constructed grammar. Non-terminal symbols will be thought of as ordered triples of the form (v, s_i, s_j) where v is a member of the vocabulary, terminal or non-terminal, of the original finite-state grammar. The constructed grammar will have one set of rules of this sort:

(A) For each symbol c in the terminal vocabulary, and rule $s_i \rightarrow cs_j$ in the finite-state grammar, there will be a rule $(c, s_i, s_j) \rightarrow c$.

This means that the final twigs and leaves of a tree derived from the constructed grammar will look like this, for the typical terminal string:

Tree 3

$$\begin{tikzcd}
(a_1, s_0, s_1) \arrow[d, dash] & (a_2, s_1, s_2) \arrow[d, dash] & (a_3, s_2, s_3) \arrow[d, dash] & \cdots\cdots\cdots & (a_n, s_{n-1}, s_n) \arrow[d, dash] \\
a_1 & a_2 & a_3 & & a_n
\end{tikzcd}$$

It will be seen how the line of triples "imitates" the state diagram: the first bracket shows a transition from the initial state to s_1, the second a transition from s_1 to s_2 and so on. Now in the original context-free grammar, we can suppose that terminal symbols come from rules of the form $X \rightarrow c$. These rules will be taken over into the constructed grammar by arranging:

(B) There will be a rule of the form $(X, s_i, s_j) \rightarrow (c, s_i, s_j)$ for each rule $X \rightarrow c$ of the original context-free grammar and every pair of non-terminal symbols s_i, s_j of the finite-state grammar.

This may seem rather extravagant: after all, if there are, say, k states in the finite-state grammar, then for each rule $X \rightarrow c$ there will be k^2 rules of the above type. Furthermore, not every (c, s_i, s_j) on the right-hand side can be rewritten. According to (A) above, (c, s_i, s_j) can only be rewritten if the finite-state grammar has a rule $s_i \rightarrow cs_j$. But it follows from the usual formal definition (see for example, Salomaa, 1973 p9) that only strings of *terminal* symbols which can be generated by a grammar are to count as part of the language. We can use this as a useful filter device: those derivations which end up with any "wrong" triples (c, s_i, s_j) can just count as "discarded". With that problem out of the way, it only remains to arrange that the leaves of each tree are dominated by triples containing a correct sequence of states from s_0 to s_n.

This is not so hard to do if we specify that the grammar of the original context-free language be given in Chomsky normal form. The constructed grammar then includes rules like this:

(C) There will be a rule of the from $(X, s_i, s_k) \rightarrow (Y, s_i, s_j)(Z, s_j, s_k)$ for each rule of the original grammar $X \rightarrow YZ$ and each triple of non-terminal symbols of the finite-state grammar s_i, s_j, s_k.

Again an extravagant number of rules. But it will be noticed that these rules ensure that at each bifurcation of the tree, the second state number in each bracket will be the same as the first state number in the bracket to the right. Thus there will be a proper sequence of states in triples as we look from left to right in any tree.

It still remains to ensure that the leftmost state will be s_0 and the rightmost one s_n. This of course is decided right at the root of the tree, before any bifurcation takes place at all. All we need to do, then, is to specify that the initial symbol of the constructed grammar be (X_0, s_0, s_n) where X_0 is the initial symbol of the original context-free grammar.

Now imagine a typical tree structure derived from this constructed grammar. Because of the way the initial symbol has been chosen, and the nature of (B) and (C) above, it will be seen that if each triple was deleted except for its first member, a typical tree of the original context–free grammar would remain. This means that the constructed grammar can only generate strings which could have been generated by the original one. Each tree will end in the way pictured in tree 3 above. Because of (A), there can only be a complete terminal string if the sequence of states and symbols illustrated in tree 3 corresponds to a state diagram of the finite–state grammar. Thus any terminal string belongs to the intersection of the two languages, as required, while the constructed grammar is certainly context–free.

Readers who are dissatisfied with introducing a filter device into the constructed grammar, in view of the strictures against them in, say, Gazdar (1982) should note that they are really harmless in the case of context–free grammars. In any particular case, rules containing non–terminal symbols which do not get rewritten may be eliminated from the grammar. The method is similar to the way in which superfluous non–terminal symbols were removed in the introduction, and I will leave the details to those dissatisfied readers.

That completes the proof of theorem 2. The following theorem is rather easier to prove:

Theorem 3:

The language consisting of the images according to a deletion homomorphism of the sentences of a context–free language is also context–free.

By "deletion homomorphism" I mean something like the one considered before, where certain symbols in a string are deleted while others are left unchanged.

Consider some context–free grammar G and an arbitrary deletion homomorphism. A new grammar is formed just like G except that every rule of the form $X \rightarrow c$, where c is a terminal symbol deleted by the homomorphism, is replaced by a rule $X \rightarrow \lambda$. Deletion rules are quite proper in a context–free grammar, as we noted before, so the new grammar is certainly context–free. But it generates precisely the homomorphic image of the original language, as required.

Actually theorem 3 holds for arbitrary homomorphisms, not just for deletion ones, but the proof becomes more complicated, and the generalisation is unnecessary for our purposes. Details may be found in chapter 1, section 3 of Salomaa (1973).

Theorems 2 and 3 are readily combined to give theorem 4:

Theorem 4:

The language consisting of the images according to a deletion homomorphism of the strings in the intersection of a context-free language and a finite-state language is also context-free.

What this means, returning to previous discussion, is that as long as the extended Dyck grammar remains context-free, the homomorphism of the intersection of its language with the finite-state language will also remain context-free.

At first sight, this may not seem a very exciting result. Dyck languages exhibit a very special form of context-freeness, and to express context-free languages in general in terms of them, as theorem 2 does, is something unexpected and interesting. This point of interest just seems to have been discarded.

If we know that some string belongs to a Dyck language, and scan it from left to right, we can be sure, on coming across some x_i, that an x_i' will turn up later to match it. In a way, we can look at this as some kind of promise and fulfilment: the x_i promises the later occurrence of x_i'.

This is rather reminiscent of the nature of context-sensitive rules. An example will help:

$$T \rightarrow UV/\text{–}W$$
$$T \rightarrow YZ$$

Imagine tracing out a path round a tree, of the kind illustrated in tree 1 above. If the context–sensitive rules are being observed, then if you come across a node T followed closely by nodes U and V, you can be certain that at some later stage a W will follow. In that case, why not try to express this kind of context–sensitivity with some form of extended Dyck grammar? If we use the symbol “y” to mean something like, “a context–sensitive rule is being applied; context to come”, while “y′ ” signifies “here is the context”, might not a Dyck–like grammar help to produce just the right sentences? Let's try to work out some details.

Firstly, the finite–state grammar. In the ordinary way of things, this would need to contain a rule of the form $T \rightarrow x_iU$, to correspond to $T \rightarrow UV$. Let us alter this slightly to $T \rightarrow yx_iU$. Elsewhere in the context–sensitive grammar, of course, there must be a rule which expands W. In the finite–state grammar too there would have to be a rule of the form W → ...., but in its place we are going to introduce two rules:

W → ...., the rule as before, and also
W → y′ .... where the right–hand side is preceded by y′ .

No matter what the details of the extended Dyck grammar are, it will certainly require that every string containing a “y” must include a “y′ ” companion. In the finite–state grammar that we are developing, a “y” must appear if T is developed as UV, and in that case the option for W containing “y′ ” must be chosen if the string is to be grammatical. If T is expanded as YZ, however, there is no “y”, and it is immaterial whether or not W is selected later. This is just the situation required by the context–sensitive rules. Rules are unordered, of course, so T may freely be expanded as YZ; on the other hand, if it is expanded as UV, there must be a “W” later in the tree. Notice, in passing, that this method of dealing with context–sensitivity easily solves the problem of the formal nature of “elsewhere” in rules such as:

T → UV/–W
T → YZ, elsewhere

The use of "elsewhere" seems to be an underhand way of bringing in rule ordering. If in the finite-state grammar W is *only* expanded as "y′ ....", however, the elsewhere condition is simply satisfied, since an occurrence of "y′ " without "y" would be ungrammatical.

Now back to the main problem. I spoke above of situations where there must be a "W" "later in the tree". What does this phrase actually mean? The possible positions of "y" and "y′ " in a string are governed by the extended Dyck grammar, and we must turn now to that. Suppose that in addition to the usual rules of a Dyck grammar (given in grammar 5 above), we have these:

$$\begin{aligned} X &\rightarrow X_1X_2 \\ X_1 &\rightarrow y \\ X_2 &\rightarrow y' \end{aligned}$$

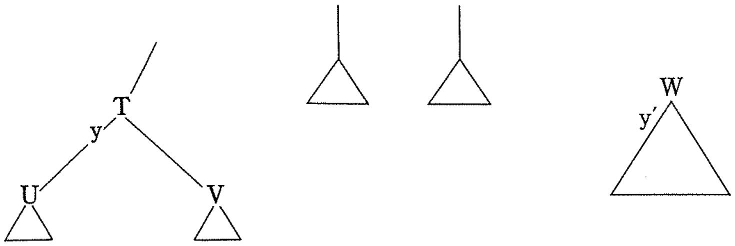
Both "y" and "y′ " are now introduced, but without further rules which might move X_1 and X_2 apart, they could only occur next to each other. Of course we can introduce what rules we like, as long as they remain context-free and each X_1 has an X_2 paired with it. The following seem to be the most liberal we could allow, under the conditions:

y, y′ rules 1
$$\begin{aligned} X_1 &\rightarrow XX_1X \\ X_2 &\rightarrow XX_2X \\ X_1 &\rightarrow zX_1z' \\ X_2 &\rightarrow zX_2z' \end{aligned}$$

The third and fourth of these rules are in fact rule schemata. The symbols z, z′ are to range over all the pairs x_i, x_i' required in the particular case. Remember that the c, c′ used in grammar 5 were to range over *all* terminal symbols, not just the x_i, x_i' pairs.

Now we can investigate just what things might intervene between "y" and "y′ " in the tree structure. The simple answer seems to be, anything! The following schematic tree will illustrate this:

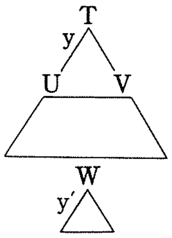
Tree 4



Imagine tracing a path round this tree structure, a path which will of course be represented by a terminal string of the extended Dyck language including y and y′. Before we reach T, and y itself, there will be some x_i for the branch or branches on which T lies. The rules of the form $X_j \rightarrow zX_jz'$ will allow for these x_i. Later we come to the vague triangles beyond U and V. Whatever is in them, though, can only be "complete" structures with x_i and associated x_i'. These are taken care of by the possibilities for "X" in the rule $X_1 \rightarrow XX_1X$. After the triangle below V, we must pass up into upper reaches of the tree to the right of T. Here there must be some x_i' to correspond with the x_i that came before T. But the rules $X_1 \rightarrow zX_1z'$ have already taken care of these. There are still possibilities for more complete "X" structures before we reach W: these are shown by the central triangles. Finally, as we come down branches towards W itself, there will be some x_i not to be "redeemed" by x_i' until beyond the W triangle. These may be dealt with by the rules $X_2 \rightarrow zX_2z'$.

As if that wasn't enough, there is another possibility not well illustrated by tree 4:

Tree 5

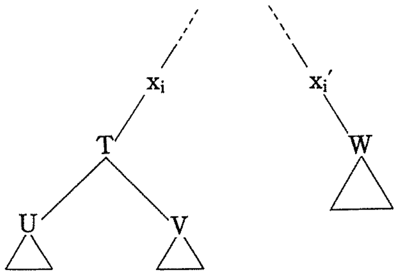


W may actually occur somewhere in the tree structure below U or V. In this case the original context-sensitive rule T → UV/–W is being interpreted as a "domination predicate", and should be written T → UV/δ (—W) according to the notation used by Gazdar (1982). Joshi and Levy (1977) use a similar notation. The meaning of this particular domination predicate is that T may be rewritten as UV as long as there would then be a path in the branches of the tree leading from T to W. Looking at tree 5 we see that there must be such a path, passing through either U or V on the way. Section 3 of Gazdar (1982) contains a more full description of such predicates.

There certainly seems no great problem in establishing context-sensitive rules in this framework. The particularly interesting thing, of course, is that since we are still dealing with the intersections of context-free and finite-state languages, the resulting languages are still context-free, in spite of the context-sensitivity!

There still may be a problem in constraining the rules sufficiently. After all, the intention may be that a rule like T → UV/–W is only supposed to take effect when W is in an adjacent branch of the tree to the right of T, like this:

Tree 6



If T and W are adjacent, then the linking branch is the one labelled x_i', somewhere above T and W. As we go round the tree, moving round from T towards the x_i branch, we should not come across any other x_j' branch whose companion x_j occurs *before* T. Moreover, as we go on from x_i' to W, we should come across x_j type branches only, with no x_j' intervening. All these conditions can be met if the liberal y, y′ rules 1 above are replaced by these:

y, y′ rules 2

$$\begin{aligned} X &\rightarrow x_i X_1 x_i' X_2 \\ X_1 &\rightarrow X\, X_1 X \\ X_2 &\rightarrow z\, X_2 X z' \end{aligned}$$

It seems that the rules can be suitably constrained in general. As a further example, consider the domination predicate T → UV/δ(–W), illustrated in tree 5. Here it is essential that W be dominated by T. Tracing the usual path round the tree, then as we go up the right hand side of the quadrilateral, past V, and then past T, we must eventually come to a fork and a branch labelled with some x_i'. W will certainly be dominated by T if it comes between the companion x_i and x_i' on this path. Thus we need rules of this kind:

y, y′ rules 3 $X \rightarrow x_i X_1 X_2 x_i'$

$X_1 \rightarrow X X_1 X$

$X_2 \rightarrow X X_2 X$

$X_1 \rightarrow z\, X_1\, z'$

$X_2 \rightarrow z\, X_2\, z'$

Clearly all kinds of possibilities can be allowed for by suitably juggling with the rules of the extended Dyck grammar. It would need some improved notation, at least, to go into them further, and I will not do so here. Some general points should be noted, though.

For one thing, I have tacitly been assuming context-free grammars in Chomsky normal form, since each extension of the Dyck grammar has been added to the rules of grammar 5. There is no necessity for this however: there is no reason why the y, y′ rules should not mix with those of the form $X \rightarrow X_{12} X x_1' \; X x_2'$. Only the complexity of the final extended Dyck grammar would increase, unless a really elegant notation could be found, at least.

In the examples above, only context of the form "–W" was mentioned. This is easily amended. For context left and right, for instance, then rather than just "y" and "y′" we could do with, say, "y_1", "y_2", and "y_3", with appropriate rules to introduce them. There is no need to stop at y_3: with extra y_i we could introduce far more complex kinds of context if required, as in these examples:

T → UV/A–B and δ(C__D)

T → UV/A–B or δ(C__D)

It has already been mentioned that "elsewhere" rules can be accommodated. Of course another way to express, say,

$$\begin{array}{l} T \rightarrow UV/\_W \\ T \rightarrow YZ \quad \text{elsewhere} \end{array}$$

is to write

$$\begin{array}{l} T \rightarrow UV/\_W \\ T \rightarrow YZ/\text{not } (\_W) \end{array}$$

With "and", "or" and "not" available, then any Boolean combination of contexts can be dealt with.

In view of all this, then what kind of contexts cannot be allowed? There must be something, since context-sensitive grammars are provably more powerful than context-free ones. The answer is already hidden in diagrams like tree 4. Thinking again of the string which describes a path taken round the tree, we know that it will be grammatical as long as y and y′ occur in it in accordance with the rules of the extended Dyck grammar. What may be hidden in the triangles below U and V and W does not make any difference. This means that the context-sensitive rules are being used as *node admissibility conditions* , to use the terminology of McCawley (1968), and not as conditions applicable just in a particular line of a "derivation" in Chomsky's sense.

These remarkable results are the ones summarized by Joshi and Levy (1977) as their theorem 3.1 (p276). Our quite informal proof of it above is quite different from theirs, which is based on the theory of finite automata. I hope it is clear that the informal proof could be formalized, with no more than purely technical difficulties. If that is so, we may be entitled to reproduce Joshi and Levy's theorem here, with the label "theorem 5".

Theorem 5 :

Let G be a finite set of local transformations and $\tau(G)$, the set of trees analyzable by G. Then the string language L (τ(G)) is context-free.

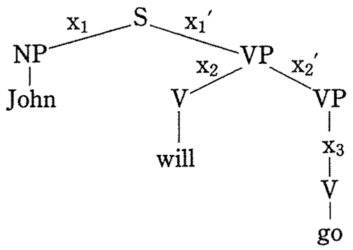
A "set of local transformations" is a context-sensitive grammar in which Boolean combinations of proper analyses and domination predicates are allowed. "Analyzable by G" means that the rules of G are interpreted as node admissibility conditions.

Pure Speculation

It seems to me that this method of language intersection affords not only an alternative proof of Joshi and Levy's theorem, but also may have some promise in the investigation of natural language. I want to mention certain possibilities now as a conclusion to this article, even though they remain as yet purely speculative.

Although the possibility of context-sensitive rules which do not affect the context-freeness of a language is pointed out with great clarity in his (1982), Gazdar makes virtually no use of them in the remainder of that paper. They take no place either in the methods employed in Gazdar, Klein, Pullum and Sag (1985). Instead, a great deal of space is devoted to ways in which features may be "passed down" from node to node in the branches of tree structures, thus effectively linking various parts of a terminal string. The method of achieving certain kinds of context-sensitivity with the y, y′ symbols developed above provides an alternative way of dealing with similar problems, and it should be interesting to work out the details. In the language intersection method, the order of elements is handled by the finite-state grammar, while the extended Dyck grammar can link elements in widely separate parts of a string. It may be that certain features of natural languages could be explained more easily by such a division of labour, rather than by investigating the overall context-free grammar.

There may also be interesting links between these methods and those of dependency grammars. The x_i, x_i' and y, y′ symbols might be regarded as a means of displaying the ends of dependency arrows of the kind used by, for example, Hudson (1984). The relations between phrase–structure grammars and certain kinds of dependency grammars are explored in Gaifman (1965). The present method may allow certain generalizations of those findings to be easily made. Hudson points out certain deficiencies of constituent structure as compared with dependency structure (1984 p94 et seq.). Since the language intersection method explored here has affinities with both kinds of structure, it would be interesting to use it in such investigations. As a preliminary example, consider the following unremarkable tree structure:



This could be associated with the following terminal string of an extended Dyck grammar:

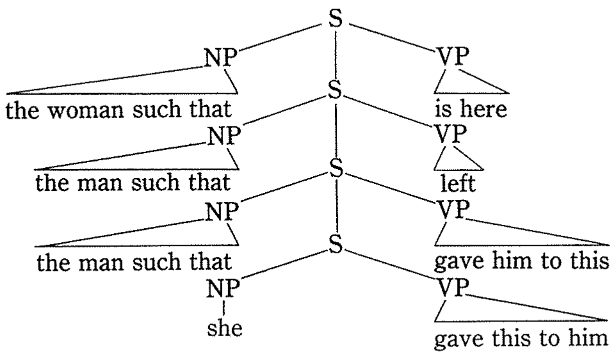
x_1John x_1' x_2 will x_2' x_3 go

If the string is regarded as a representation of dependencies, then the x_i and x_i', could be taken as features of the lexical items which follow them, indicating the relationships between these items. This has been illustrated by the spacing. The string for the question form, "Will John go?" preserves these relationships:

x_1' x_2 will x_1John x_2' x_3 go

It is not possible to reconstruct a tree from this, even if an x_i' was allowed to label the left–hand branch from some node. I am not suggesting this as the basis of any serious analysis. What it does imply is that there is no essential difference between the x_i, x_i' and the y, y′ symbols that we have been using. In certain configurations, it may be possible to interpret certain of the symbols as encoding constituent structure, while others show context-sensitive relations. On the other hand, we might give up the idea of constituent structure altogether and just think about the dependency relations between lexical items. In that case, *apparent* constituent structures would just be artefacts of the essential context-freeness of natural language. It is always *possible* to establish constituent structure, because of this context-freeness, but the structure need not have any real significance.

Are natural languages context-free? This question has been answered in the negative by Higginbotham (1984). I will take his argument as an example, since it applies to English, although I believe there are more cogent arguments for other languages. As far as I am aware, however, the formal structure of all these arguments is similar. Higginbotham gives the following tree as an illustration (1984, p.227):



The sentences have the following form:

the woman such that (the man such that)^n she (gave $\begin{Bmatrix}\text{this}\\\text{him}\end{Bmatrix}$ to $\begin{Bmatrix}\text{him}\\\text{this}\end{Bmatrix}$)^n left is here

The curly brackets indicate a choice, but the choice here is not quite free, if only grammatical sentences are to be arrived at. The tree diagram illustrates the situation clearly. Each subordinate clause introduced by "the man such that" must contain an occurrence of "him" referring to "the man". This is the case in the diagram. The lowest VP in this tree could be replaced by "gave him to him", and the next lowest by "gave this to this". The resulting sentence would still be grammatical, for each clause would still contain the required "him". On the other hand, we could not replace the lowest VP by "gave this to this", since that would leave the lowest S without a "him". This would still of course be the case even if the next lowest VP was replaced by "gave him to him", to preserve the correct number of occurrences of "him".

Let us express these conditions in symbolic terms, to save space at least. If we wanted to write an extended Dyck grammar for Higginbotham's sentences, we would probably think of attaching "y" to each occurrence of "the man" and "y′ " to each occurrence of "him", to express the dependency. In outline, sentences would have this sort of structure:

String 9 $xyxyxyxyx'y'y'x'y'y'x'x'$

The x, x′ pairs may be regarded as representing S–NP and S–VP branches of the tree structure. All other items have been suppressed, but it is not hard to see that this does not affect the overall argument. In any grammatical string, the numbers of occurrences of x, x′, y and y′ must all be the same. Furthermore, there is a condition on the occurrences of y′, corresponding of course to the condition on occurrences of "him" in the English version. One simple way to state this condition is to say that, reckoning from left to right along the string, the number of occurrences of x′ must never exceed the number of occurrences of y′ by more than 1. The following string would be ungrammatical:

String 10 $xyxyxyxyx'x'y'y'x'y'y'x'$

because there are two occurrences of x′ before any one of y′. The condition can also be expressed by noting that every occurrence of x′ y′ y′ will be "balanced" later in the string by some occurrence of x′ not followed by y′. This means that each string will have one or more points of symmetry, that is points between equal numbers of balanced items. String 9, for instance, has two points of symmetry, represented by asterisks below:

String 9   xyxyxyxy * x′ y′ y′ x′ y′ y′ * x′ x′

The first lies between 4 occurrences of x and y, and 4 occurrences of x′ and y′; the second between 2 occurrences of x′ y′ y′ and 2 of "solitary" occurrences of x′. The next string has only one point of symmetry:

String 11   xyxyxyxy * x′ y′ x′ y′ x′ y′ x′ y′

The following has three:

String 12   xyxyxyxy * x′ y′ y′ * x′ x′ y′ y′ * x′

The set of strings meeting these conditions is provably not context-free, meaning that English is not a context-free language either, if Higginbotham's sentences are accepted as English. A formal proof is quite difficult, but a simpler notional one is available using the idea of points of symmetry that we have developed.

It was pointed out in the introduction that conditions of symmetry in a context-free grammar can only be met by derivations which grow from the centre of symmetry. Together with the fact that a context-free language must be able to be generated by a grammar whose non-terminal symbols are all involved in recursion, it was easy to show that the language $a^n b^n c^n$ could not be context-free, essentially because grammatical strings have two points of symmetry. But even though sentences of Higginbotham's language may have two or more points of symmetry, it is still possible to increase their length indefinitely from the leftmost such point, by adding any number of occurrences of xy to the left, and the same number of occurrences of x′ y′ to the right. The proof cannot be quite so simple in this case.

We need to use also the fact that any non-terminal symbol in the last line of a continuing derivation may be selected for rewriting to obtain the next line. Any non-terminal symbol could thus be "saved" until the last for rewriting. There must be a non-terminal symbol at each of the points of symmetry in some line in the derivation of sentences in Higginbotham's language. It must be possible to replace the leftmost of these by terminal symbols before those further to the right. Any on those remaining must of course be able to dominate an infinite number of different terminal strings, but the resulting sentences would inevitably be ungrammatical. A glance at string 12 confirms this: any additions at the second or third points of symmetry would disturb the already fixed numbers of x′ and y′.

Thus it may be that natural languages are not context-free. It would be interesting to investigate just what sort of extended Dyck languages would be needed to generate them in that case. We have been regarding the finite-state part of the intersection as contributing the order of elements, while the extended Dyck language checks their dependency relationships. From this viewpoint, it is odd that a language like $a^nb^nc^n$ should be inherently more "difficult" than just a^nb^n. In checking strings from left to right, then after ensuring that there as many occurrences of b as a, surely the initial segment of a string could be "forgotten" while it is checked that there are as many occurrences of c as b. There are in fact families of languages intermediate between context-free and context-sensitive: that is, they include all context-free languages but not all context-sensitive ones. Some are described in chapter 5 of Salomaa (1973), and may have some linguistic interest. But that speculation takes us outside the range of context-free languages, and is an appropriate point at which to conclude.

Bibliography

E. Bach (1974) "Syntactic Theory" (Holt, Rinehart, Winston)

H. Gaifman (1965) "Dependency Systems and Phrase-Structure Systems" (Information and Control, 8, 304-337)

G. Gazdar (1982) "Phrase Structure Grammar" in P. Jacobson and G.K. Pullum (eds.) "The Nature of Syntactic Representation" (Reidel)

G. Gazdar, E. Klein, G. Pullum and *I. Sag* (1985) "Generalized Phrase Structure Grammar" (Blackwell)

J. Higginbotham (1984) "English is not a Context-Free Language" (Linguistic Inquiry, 15, 225-234)

R. Hudson (1984) "World Grammar" (Blackwell)

A. K. Joshi and *L. S. Levy* (1977) "Constraints on Structural Descriptions: Local Transformations" (Siam Journal of Computing, 6, 272-284)

J. D. McCawley (1968) "Concerning the base component of a transformational grammar" (Foundations of Language, 4, 243-269)

P. S. Peters and *R. W. Ritchie* (1973) "On the Generative Power of Transformational Grammars" (Information Sciences 6, 49-83)

Arto Salomaa (1971) "The Generative Capacity of Transformational Grammars of Ginsburg and Partee" (Information and Control, 18, 227-232)

Arto Salomaa (1973) "Formal Languages" (Academic Press)