The Chomsky Hierarchy

Tim Hunter
Department of Linguistics, UCLA
timhunter@ucla.edu

February 2020

To appear in N. Allott, T. Lohndal & G. Rey (ed.), *Blackwell Companion to Chomsky*

An important cluster of closely-related early Chomsky papers[1] had two major consequences. First, they defined a new branch of mathematics, formal language theory, which has flourished in its own right. But second, and more importantly for our purposes, this new branch of mathematics provided the formal grounding for a new conception of linguistics in which *grammars*, rather than sentences or collections of sentences, were the scientifically central objects: instead being derived from collections of sentences as compact summaries of observed regularities, grammars are seen as (ultimately mental) systems that determine the status of sentences. The "observed regularities" come to be seen as consequences of the structure of the underlying system, the grammar. The classification of grammars that became known as the Chomsky hierarchy was an exploration of what kinds of regularities could arise from grammars that had various conditions imposed on their structure.

Rather than laying out the mathematical theory in complete detail — numerous sources already provide this[2] — my aim in this chapter will be to focus on bringing out some key intuitions that emerge from the theory and try to highlight their applicability to theoretical linguistics. Looking at a completely formal treatment makes it easy to overestimate the degree to which the important concepts are bound to certain idealizations, such as the restriction to strings as the objects being generated and a binary grammaticality/ungrammaticality distinction.[3] While those idealizations *are there* in the theory, I hope to make the case that certain intuitions that emerge from the theory are meaningful and useful in ways that transcend those idealizations.[4] To the extent that I succeed in making this case, the reader will be able to turn to the formal literature with some motivating ideas in mind about the important concepts to watch out for.

One idea which plays a major role is the *intersubstitutability* of subexpressions. This is familiar from the distributional approach to discovering syntactic categories that is sometimes presented in introductory textbooks.[5] We reach the conclusion that *cat* and *dog* belong to the same category, for example, by noting that substituting one for the other does not change a sentence's grammaticality. While we might introduce the term "noun" or the book-keeping symbol N as a label for the class that *cat* and *dog* both belong to, there is nothing to being a noun other than being intersubstitutable with other nouns; the two-place predicate "belongs to the same category as" is more fundamental than the one-place predicate "is a noun". (This

[1] Chomsky (1956); Chomsky and Miller (1958); Chomsky (1959, 1963); Chomsky and Miller (1963); Miller and Chomsky (1963).

[2] For standard presentations from the general perspective of the theory of computation, see e.g. Hopcroft and Ullman (1979), Lewis and Papadimitriou (1981) and Sipser (1997). For more linguistics-oriented presentations, see e.g. Levelt (1974), Partee et al. (1990).

[3] For generalizations beyond the case of strings as the generated objects, see the rich literature on tree grammars (e.g. Thatcher, 1967, 1973; Thatcher and Wright, 1968; Rounds, 1970; Rogers, 1997; Comon et al., 2007). Generalizations beyond binary grammaticality arise via the theory of semirings (e.g. Kuich, 1997; Goodman, 1999; Mohri, 2002).

[4] Notice that the argument here does not concern the usefulness of the traditional notion of weak generative capacity that emerges from the original work on the Chomsky hierarchy, or the viewpoint which equates natural languages with sets of strings and asks where those sets of strings fall on the hierarchy (or extensions of it). The main point I hope to make here is that the usefulness of the Chomsky hierarchy for theoretical linguistics need not be limited to what emerges from those traditional and better-known perspectives.

diverges from the view where a noun, for example, would be defined as a word that describes a person, place or thing.)

Intersubstitutability is closely related to the way different levels on the Chomsky hierarchy correspond to different kinds of memory. A grammar that will give rise to the intersubstitutability of *cat* and *dog* is one that ignores, or *forgets*, all the ways that they differ, collapsing all distinctions between them. Similarly for larger expressions: the distinctions between *wash the clothes* and *go to a bar*, such as the fact that they differ in number of words and the fact that only one of the two contains the word *the*, can be ignored. The flip side of this irrelevant information, that a grammar ignores, is the *relevant* information that a grammar tracks — this remembered, relevant information is essentially the idea of a category. Different kinds of grammars correspond to different kinds of memory in the sense that they differ in how these categories, this remembered information, are used to guide or constrain subsequent generative steps.

Much of the discussion below aims to show that this idea of intersubstitutability gets at the core of how any sort of grammar differs from a mere collection of sentences, and how any sort of grammar might finitely characterize an infinite collection of expressions. A mechanism that never collapsed distinctions between expressions would be forced to specify all combinatorial possibilities explicitly, leaving no room for any sort of productivity; a mechanism that collapsed all distinctions would treat all expressions as intersubstitutable and impose no restrictions on how expressions combine to form others. An "interesting" grammar is one that sits somewhere in between these two extremes, collapsing some but not all distinctions, thereby giving rise to constrained productivity — productivity stems from the distinctions that are ignored, while constraints stem from those that are tracked. The task of designing a grammar to generate some desired pattern amounts to choosing which distinctions to ignore and which distinctions to track.

1 Rewriting grammars

We begin with the general concept of a string-rewriting grammar, which provides the setting in which the Chomsky hierarchy can be formulated.

1.1 Unrestricted rewriting grammars

An *unrestricted rewriting grammar* works with a specified set of *nonterminal symbols*, and specified set of *terminal symbols*. One of the nonterminal symbols is designated as the grammar's *start symbol*. The grammar also specifies a set of *rewrite rules* of the form $\alpha \to \beta$ where α is any non-empty string of nonterminal and terminal symbols, and β is any (possibly empty) string of nonterminal and terminal symbols. A rewrite rule $\alpha \to \beta$ says that from any string ϕ that contains α, we can derive a new string which is like ϕ but has α replaced with β. We say that a rewriting grammar *generates* a string s if s can be derived from the grammar's start symbol via a sequence of steps using the grammar's rewrite rules, and s contains only terminal symbols.

It turns out that grammars of this sort are extremely powerful — in fact, they are general purpose computing devices, capable of carrying out any computation that a Turing machine is capable of.[6] For example, the grammar in Figure 1[7] emulates a machine that begins with the length-one string *a*, and "doubles" this string repeatedly to produce *aa*, then *aaaa*, then *aaaaaaaa*, before stopping at some point with the current string as output. I use uppercase letters for nonterminal symbols and letters in a different font for terminal symbols. So in the grammar in Figure 1, the set of nonterminal symbols is $\{\mathrm{S}, \mathrm{L}, \mathrm{R}, \mathrm{F}, \mathrm{B}, \mathrm{X}\}$, and the set of terminal symbols is $\{a\}$. The empty string is written as ε.

For any string s that we choose, the grammar in Figure 1 generates s if and only if the length of s is a power of two and no symbol other than *a* occurs in s. For convenience we sometimes say, using "generate" in a subtly different sense, that the grammar generates the set of strings that satisfy these conditions. A

[5] See e.g. Carnie (2013, pp.48–50), Fromkin et al. (2000, pp.147–151). Johnson (2019) gives a particularly clear presentation of the fundamental relationship between substitution classes and phrase structure.

[6] See e.g. Chomsky (1963, pp.358–359), Levelt (1974, pp.106–109), Partee et al. (1990, pp.516–517), Hopcroft and Ullman (1979, pp.221–223).

[7] This is based on an example from Hopcroft and Ullman (1979, pp.220–221).

start symbol: S

i.	S → L F *a* R
ii.	F *a* → *a a* F
iii.	F R → B R
iv.	*a* B → B *a*
v.	L B → L F
vi.	L B → L X
vii.	L X → X
viii.	X *a* → *a* X
ix.	X R → ε

(a) An unrestricted rewriting grammar

start:	S	
apply rule i:	L F *a* R	
apply rule ii:	L *a a* F R	
apply rule iii:	L *a a* B R	first doubling cycle
apply rule iv:	L *a* B *a* R	
apply rule iv:	L B *a a* R	
apply rule v:	L F *a a* R	
apply rule ii:	L *a a* F *a* R	
apply rule ii:	L *a a a a* F R	
apply rule iii:	L *a a a a* B R	
apply rule iv:	L *a a a* B *a* R	second doubling cycle
apply rule iv:	L *a a* B *a a* R	
apply rule iv:	L *a* B *a a a* R	
apply rule iv:	L B *a a a a* R	
apply rule vi:	L X *a a a a* R	
apply rule vii:	X *a a a a* R	
apply rule viii:	*a* X *a a a* R	
apply rule viii:	*a a* X *a a* R	cleaning up
apply rule viii:	*a a a* X *a* R	
apply rule viii:	*a a a a* X R	
apply rule ix:	*a a a a*	

(b) A derivation using this grammar

Figure 1: The nonterminal symbols L and R mark the left and right edge of the string. We begin with one occurrence of *a* between these markers; the nonterminal symbols F and B serve as a device that doubles this inner string to form *aa*, then *aaaa*, etc., arbitrarily many times. On each single "doubling cycle" of this device, F moves forward through the string replacing each occurrence of *a* with two (rule ii), and then B moves backward (rule iv) until it reaches the left edge. At the end of any such cycle, B can be replaced (rule vi) by the "cleaning up" symbol X which deletes the left edge marker (rule vii), then moves rightwards through the string (rule viii) to eventually delete the right edge marker (rule ix).

set of strings is sometimes called a "stringset".[8] Thus the grammar in Figure 1 generates the stringset $\{a^n \mid n \text{ is a power of two}\} = \{a, aa, aaaa, aaaaaaaa, \dots\}$.

Note that there is no direct connection between the notion of generation and that of sets: a grammar generates a set only by virtue of generating (all and only) the strings contained in it. While it is convenient to talk of "generating a set of strings", fundamentally it is strings that are generated, not sets — similarly, while it is convenient to talk of "eating a pile of apples", it is apples that are eaten, not piles.

1.2 Restrictions on grammars

There are (at least) two reasons why we might question the usefulness of unrestricted rewriting grammars in our theories of natural language.

An obvious one perhaps is that since they can carry out any computational procedure at all, adopting this class of grammars would not constitute a meaningful hypothesis about what is a possible human language. This is an objection on the grounds of the *generative capacity* of the formalism.

The concern that is more prominent in Chomsky's early discussions, however, involves the notion of a *structural description*. This concern is not about the range of possible grammars made available by the formalism, but about what is said by the fact that a particular grammar generates a particular string. The idea is that if a grammar f generates a sentence x, then we would like to be able to identify "a structural

[8] Much of the technical literature uses the term "language" here, but this creates unnecessary distractions.

Restriction	**Required form of rules**	
Restriction 1, "context-sensitive"	$\phi A \psi \rightarrow \phi \omega \psi$	where ω is a non-empty string
Restriction 2, "context-free"	$A \rightarrow \omega$	where ω is a non-empty string
Restriction 3, "finite-state"	$A \rightarrow x$ or $A \rightarrow xB$	where x is a terminal symbol and B is a nonterminal symbol

Table 1

description of x (with respect to the grammar f) giving certain information which will facilitate and serve as the basis for an account of how x is used and understood by speakers of the language whose grammar is f; i.e., which will indicate whether x is ambiguous, to what other sentences it is structurally similar, etc." (Chomsky, 1959, p.138). Chomsky's chosen way to make this notion precise was to import the existing idea of "immediate constituent analysis" (Harris, 1946, 1954; Wells, 1947) and take trees of the now-familiar sort to be the desired kind of structural descriptions (Chomsky, 1959, p.141).[9] But there is no obvious way to associate a derivation in an unrestricted rewriting grammar with a tree structure that provides an immediate constituent analysis for the generated string. For example, there is no natural way to assign such a tree structure to the string *aaaa* on the basis of the derivation shown in Figure 1. This is roughly because the grammar contains rules that do not simply replace a single symbol with a string (Chomsky and Miller, 1963, p.294).

Another computational device that had attracted much attention in the 1950s was the finite-state automaton (FSA) (e.g. Rabin and Scott, 1959). Chomsky (1956) investigated the generative capacity of FSAs and concluded that it was insufficient for natural language, and showed that a certain sort of phrase structure grammar formalizing the notion of immediate constituent analysis could handle at least some cases that FSAs could not.[10]

The classes of rewriting grammars investigated in Chomsky 1959 are therefore motivated by the interest in "devices with more generative capacity than finite automata but that are more structured (and, presumably, have less generative capacity) than arbitrary Turing machines" (Chomsky, 1963, p.360), the idea being that "the intermediate systems are those that assign a phrase structure description to the resulting sentence" (Chomsky, 1959, p.139). A sequence of three increasingly strict restrictions on rewriting grammars (Chomsky, 1959, p.142) produces a hierarchy of classes of grammars, the broadest of which corresponds to Turing machines and the narrowest of which corresponds to finite-state automata.

These three restrictions are shown in Table 1. In this table, A stands for any nonterminal symbol, and ϕ and ψ stand for any (possibly empty) strings of terminal and nonterminal symbols. Notice that each of these restrictions requires that a rule replaces a single symbol with some non-empty string, in order to allow the possibility of constructing a tree expressing immediate-constituent structure from any derivation.[11]

To be precise, these are restrictions on the form that individual *rules* can take, and a grammar is of a certain type if and only if all of its rules satisfy the corresponding restriction. But notice that rules that satisfy Restriction 2 necessarily also satisfy Restriction 1 (by choosing ϕ and ψ to be the empty string), and similarly rules that satisfy Restriction 3 also satisfy both of the others. So these restrictions on the form of rules give us four classes of grammars:

(1) a. The Type 0 grammars are simply all unrestricted rewriting grammars.
b. The Type 1 grammars are those satisfying Restriction 1.
c. The Type 2 grammars are those satisfying Restriction 2 (and therefore also Restriction 1).

[9] See also Chomsky 1956, §3.1, Chomsky and Miller 1963, pp.288–289.

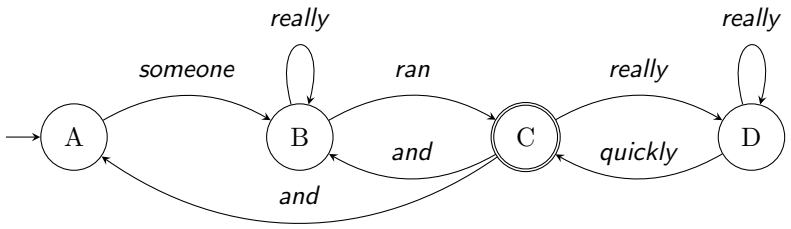
[10] The phrase structure grammars considered in section 3 of Chomsky (1956) do not correspond exactly to any of the classes in (1) that are discussed in Chomsky (1959).

[11] Citing Harris 1951, Chomsky (2006, p.172, fn.15) writes that "The concept of 'phrase structure grammar' was explicitly designed to express the richest system that could reasonably be expected to result from the application of Harris-type procedures to a corpus".

start symbol: A

A	→	*someone* B
B	→	*really* B
B	→	*ran* C
B	→	*ran*
C	→	*and* A
C	→	*and* B
C	→	*really* D
D	→	*really* D
D	→	*quickly* C
D	→	*quickly*

(a) Rewrite-rule representation



(b) Graphical representation

Figure 2: A simple finite-state grammar, represented graphically and as rewrite rules

d. The Type 3 grammars are those satisfying Restriction 3 (and therefore also Restrictions 1 and 2).

The grammar in Figure 1 is a Type 0 grammar and does not belong to any of the more restricted classes, since rules such as '*a* B → B *a*' do not satisfy Restriction 1. It turns out, however, that there is a Type 1 grammar that generates this same powers-of-two stringset.[12] This raises the question of how these classes of grammars relate to generability of stringsets: are there stringsets that can be generated by a Type 0 grammar but not by any Type 1 grammar? Or is the extra rule-writing freedom that comes with Type 0 grammars actually inconsequential from the perspective of generable stringsets?

It turns out that all four classes of grammars are distinct in their generative capacity: at each boundary, there are stringsets that can be generated by a Type n grammar that cannot be generated by any Type $(n+1)$ grammar. So there are, for example, Type 0 grammars that generate stringsets that no Type 1 grammar can generate — even if the Type 0 grammar in Figure 1 is not one of them.

2 Type 3 grammars: finite state grammars

As mentioned above, Type 3 grammars were understood from the outset to be a reformulation of finite-state automata in the setting of rewriting grammars, so I will generally describe them as "finite-state grammars" (FSGs).

Figure 2 shows an example of a FSG in rewrite-rule format, and also the equivalent finite-state automaton in a standard graphical representation. A finite-state automaton works with a specified finite set of *states* (indicated with circles in the diagram) and a specified set of *(terminal) symbols*. One of the states is the designated *start state* (indicated with an unlabeled arrow), and some of the states are designated *accepting states* (indicated with a double circle). The workings of the automaton are specified as a set of *transitions*, each associating an ordered pair of states with a symbol, and represented graphically by an arrow.

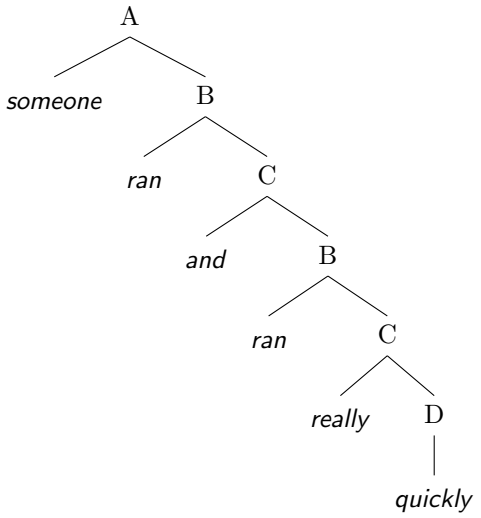
A finite-state automaton generates a string s if and only if there is a connected sequence of transitions leading from the start state to an accepting state, that emit the symbols of s in order. For example, the automaton in Figure 2 generates *someone really ran*, but not *someone ran really* or *someone ran quickly*.

To see the connection between Type 3 rewriting grammars and finite-state automata, we associate states with nonterminal symbols, and in particular associate the start state of an automaton with the starting nonterminal of a grammar. The grammar's rewrite rules correspond to the automaton's transitions. Consider

[12] Hopcroft and Ullman (1979, p.224) show that this stringset can be generated by a grammar consisting of rules $\alpha \rightarrow \beta$ where β is at least as long as α. The stringsets generable by grammars satisfying this "non-contracting" requirement are the same as those generable by Type 1 grammars (Chomsky, 1959, pp.144–145). The non-contracting requirement is sometimes given as an alternative condition defining Type 1 grammars, e.g. Levelt (1974, pp.27–29). Chomsky (1963, pp.360–363), departing from the Chomsky 1959 numbering system that has now become standard, defines Type 1 grammars with the non-contracting requirement, and calls grammars with rules satisfying the $\phi A \psi \rightarrow \phi \omega \psi$ format "Type 2 grammars".

A
someone B
someone ran C
someone ran and B
someone ran and ran C
someone ran and ran really D
someone ran and ran really quickly

(a) Rewriting derivation



(b) Tree diagram

Figure 3: Two equivalent representations of how the grammar in Figure 2 generates the string *someone ran and ran really quickly*

for example the transition from state A to state B emitting *someone*. If we think of each nonterminal X as characterizing all strings that can be used to move the automaton *from* state X to an accepting state, then the rewrite rule 'A $\rightarrow$ *someone* B' says that a string can move the automaton from state A to an accepting state if it's made up of the symbol *someone* followed by a string that moves from state B to an accepting state. For transitions into an accepting state we include an additional rule with no nonterminal on the right hand side, such as 'B $\rightarrow$ *ran*'.

The string-rewriting derivation and the tree structure in Figure 3 both show that *someone ran and ran really quickly* is generated by both the rewriting grammar and the automaton in Figure 2.

The central idea in understanding the capabilities and limitations of FSGs is what I will call a string's *forward set*. Adopting the automaton perspective, the forward set of a string s is the set of states that the automaton could reach from its initial state by taking some sequence of transitions that produce s. For example, if the automaton in Figure 2 has, from its initial state, taken some sequence of transitions that produces *someone ran*, then the only state that it might be in is C; so the forward set of *someone ran* (for this automaton) is $\{\text{C}\}$. Similarly, the forward set of *someone really* is $\{\text{B}\}$, the forward set of *someone ran and* is $\{\text{A}, \text{B}\}$, and the forward set of *someone and* is the empty set. The automaton generates a string if and only if the forward set of the string contains an accepting state.

Importantly, knowing the forward set of some initial part of a string (i.e. a *prefix* of the string, in formal terms) provides a head-start towards calculating the forward set of the entire string. For example, suppose we are told that the forward set (using the grammar in Figure 2) of some particular string s is $\{\text{C}\}$. Then, writing $+\!\!+$ for string concatenation, it is straightforward to see that the forward set of the string $s +\!\!+$ *really* is $\{\text{D}\}$, and that the forward set of $s +\!\!+$ *and* is $\{\text{A}, \text{B}\}$. We can be sure of this without knowing anything about s except its forward set. The forward set captures everything there is to know about how the automaton treats the prefix s; it identifies what we can think of as the category of that subexpression.

This leads us to the crucial idea of *intersubstitutability* mentioned in the introduction. Notice that *someone ran* and *someone really ran* both have the same forward set, namely $\{\text{C}\}$. So for any string t, the forward set of *someone ran* $+\!\!+ t$ must be the same as the forward set of *someone really ran* $+\!\!+ t$ — because in each case, all that matters is how the string t can "move us forward" from whatever states happen to be in the common forward set of *someone ran* and *someone really ran*. Furthermore, the two strings *someone ran* $+\!\!+ t$ and *someone really ran* $+\!\!+ t$ are either both generated by the grammar, or neither is, since these two strings'

Forward set	Example strings from the corresponding equivalence class
{A}	ε
{B}	*someone* *someone really* *someone really really* *someone ran and someone* *someone really ran and someone*
{C}	*someone ran* *someone really ran* *someone ran and ran* *someone ran and someone ran*
{D}	*someone ran really* *someone ran really really* *someone ran and someone ran really*
{A, B}	*someone ran and* *someone really ran and* *someone ran and someone ran and*
∅	*someone and* *someone someone* *and ran*

Table 2: Equivalence classes induced by the grammar in Figure 2

common forward set either does or doesn't contain an accepting state.

Stated generally, what we can conclude when two strings have the same forward set is that the relevant grammar treats them as intersubstitutable prefixes. So forward sets describe the way a grammar partitions prefixes into *equivalence classes*, collections of prefixes that are all intersubstitutable with each other. Table 2 shows the classes into which prefixes are partitioned by the grammar in Figure 2.

These intersubstitutability relationships also underlie the way "loops" in the structure of an FSG allow for the generation of infinitely many strings. A consequence of the loops in Figure 2 is that, for example, *someone really* and *someone really really* have the same forward set (namely {B}). Since these are therefore intersubstitutable, it follows that they are also both interchangeable with *someone really really really* — since we can substitute *someone really really* for the *someone really* part of itself. Any continuation that is compatible with one of these strings (e.g. the continuation *ran*) will be compatible with all others from the infinite class as well.

Figure 4 and Figure 5 show some more abstract examples to sharpen these important points.

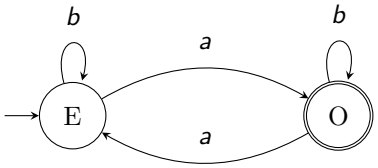
The grammar in Figure 4 requires that *a* occurs an odd number of times. Here, "what matters" about a prefix is just whether *a* occurs an odd or even number of times. The grammar reflects this by grouping all strings with an even number of *a*s together in the equivalence class represented by the forward set {E}; and similarly, for an odd number, the forward set {O}. See Table 3.

The grammar in Figure 5 requires that *a* be the second last symbol in a string. Part of what matters here is whether *a* was the second last symbol; we also need to track whether the *last* symbol was *a*, so that we know where we stand if one more symbol is added. These two yes-no questions create a four-way split, represented by the four forward sets shown in Table 4. The set of all possible strings is partitioned into these four classes, and knowing which of these four classes a string belongs to provides everything an automaton needs to know in order to enforce appropriate requirements on what follows. For example, any two strings with *a* in the last position but not the second last position will both have forward set {P, Q}, and will therefore be treated as intersubstitutable.

start symbol: E

E	→	*a* O
E	→	*a*
O	→	*a* E
E	→	*b* E
O	→	*b* O
O	→	*b*

(a) Rewrite-rule representation



(b) Graphical representation

Figure 4: Two representations of a finite-state grammar requiring an odd number of *a*s

Contains an even number of *a*s	FORWARD SET: {E} EXAMPLE STRINGS: ε, *b*, *bb*, *aa*, *baa*, *aba*, *aab*, *abba*
Contains an odd number of *a*s	FORWARD SET: {O} EXAMPLE STRINGS: *a*, *ab*, *ba*, *bab*, *bba*, *aaa*, *aaba*, *baaa*

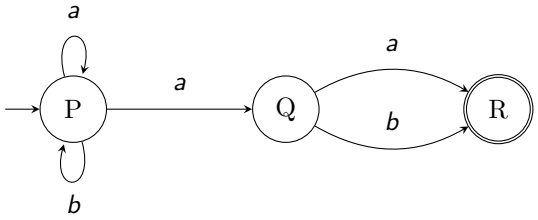
Table 3: Equivalence classes induced by the grammar in Figure 4

These examples bring out an important intuition that is likely familiar to linguists: designing a grammar amounts to choosing "what to remember" about intermediate subexpressions, and what can be ignored. It is exactly this move of ignoring distinctions between subexpressions that allows a finitely-specified grammar to generate unboundedly many expressions. A device that never allowed itself to "forget", or ignore, *some* aspects of an expression's internals, would be one where the applicability of a grammatical rule is dependent on the *entire* surrounding context, and would simply amount to a finite list of expressions. The automaton in Figure 6 is of this uninteresting sort: each state serves to remember an entire prefix, so no two distinct prefixes are "grouped together" by virtue of sharing a forward set, and the automaton simply encodes an arbitrary finite set of strings (namely {*b*, *ab*, *aaa*, *aab*, *aba*, *bba*, *bbb*}). This set exhibits no interesting patterns because the automaton has no interesting structure.

Given the central role of this partitioning of strings into classes, a natural question that arises is what sorts of partitionings are possible. How large can the number of classes induced by a finite-state grammar get? There is no upper limit, but the number of classes *must be finite*.[13] This follows from the fact that the number of states is finite: each equivalence class is identified by a subset of the set of states, and if the set of states is finite then it has only finitely many subsets.

The requirement that there only be finitely many equivalence classes allows us to succinctly identify the limitations of FSGs. Consider for example trying to construct an FSG for $a^n b^n$ (for $n \geq 1$). First note that the prefix *a* is not intersubstitutable with *aa*, since one but not the other will lead to a well-formed string when followed by *bb*. Similarly, neither of these prefixes is intersubstitutable with *aaa*. So the three strings

[13]This is the Myhill-Nerode Theorem (e.g. Hopcroft and Ullman, 1979, p.65).



(a) Graphical representation

start symbol: P

P	→	*a* P
P	→	*b* P
P	→	*a* Q
Q	→	*a*
Q	→	*b*

(b) Rewrite-rule representation

Figure 5: Two representations of a finite-state grammar requiring that the second-last symbol is *a*

	Second-last symbol is not *a*	Second-last symbol is *a*
Last symbol is not *a*	FORWARD SET: $\{\mathrm{P}\}$ EXAMPLE STRINGS: ε, *b*, *bb*, *abb*	FORWARD SET: $\{\mathrm{P}, \mathrm{R}\}$ EXAMPLE STRINGS: *ab*, *aab*, *bab*
Last symbol is *a*	FORWARD SET: $\{\mathrm{P}, \mathrm{Q}\}$ EXAMPLE STRINGS: *a*, *ab*, *aba*, *bba*	FORWARD SET: $\{\mathrm{P}, \mathrm{Q}, \mathrm{R}\}$ EXAMPLE STRINGS: *aa*, *aaa*, *baa*

Table 4: Equivalence classes induced by the grammar in Figure 5

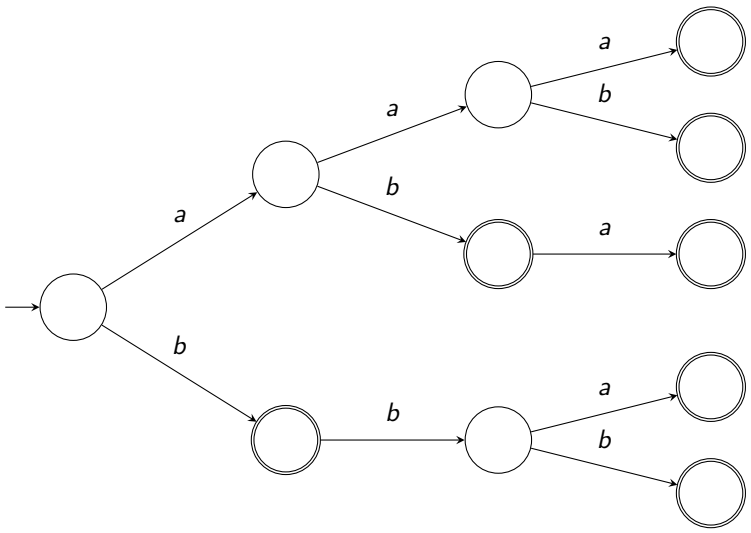


Figure 6: A boring finite state automaton where states stand in a one-to-one correspondence with prefixes

a, *aa* and *aaa* need to be split up into three distinct equivalence classes. In fact, as we go further down the list to *aaaa*, *aaaaa*, *aaaaaa* and so on, we will never find two such strings of *a*s that are intersubstitutable. No finite number of equivalence classes will be enough to track all the relevant distinctions. But for any finite number of states, there are only *finitely many different possible forward sets*; any FSG will, eventually, assign two different strings of *a*s — say, a^{73} and a^{94} — the same forward set, and will therefore incorrectly treat them as intersubstitutable prefixes (e.g. incorrectly generating $a^{73}b^{94}$ and $a^{94}b^{73}$ in addition to $a^{73}b^{73}$ and $a^{94}b^{94}$).

There is something natural about this idea: a grammar (or a mind) can only contain finitely many rules, and each rule specifies some allowable "use" for finitely many classes of expression, picked out by states or nonterminal symbols. Even if we somehow sliced up the space of all possible expressions into infinitely many equivalence classes, any finite grammar would not contain enough rules to define uses for all of those classes.

3 Type 2 grammars: context-free grammars

A Type 2 or context-free grammar (CFG) allows the right-hand side of a rule to be any mixture of nonterminal and terminal symbols. A classic example, which generates the $a^n b^n$ (for $n \geq 1$) stringset, is shown in (2). A derivation using this grammar is in (3).

(2) start symbol: S
S → *a*S*b*
S → *ab*

(3) S
*a*S*b*
*aa*S*bb*
*aaa*S*bbb*
*aaaa*S*bbbb*
aaaaabbbbb

It is helpful to consider exactly how this CFG manages to do what no FSG can. What made this pattern impossible for an FSG is that the endless collection of prefixes *a*, *aa*, *aaa*, etc. cannot all be "kept separate" by a function that maps each prefix to a subset of a finite number of states. But a CFG still has only a finite number of rules, which specify the behaviour of a finite number of nonterminal symbols; one way or another, a CFG must boil down to a collection of rules that specify finitely many ways in which subexpressions of finitely many different classes can be combined. How can this finiteness be reconciled with the fact that a CFG can generate the $a^n b^n$ pattern?

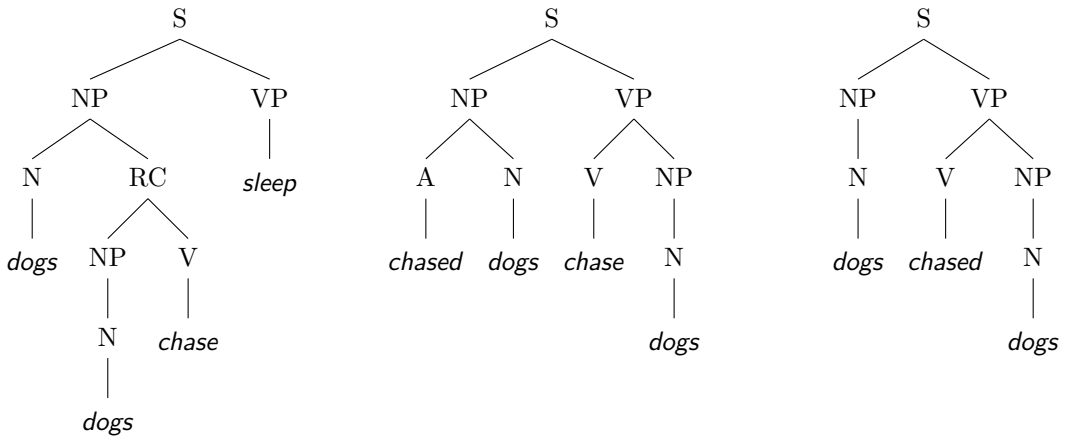
The answer is that while a FSG is limited to classifying strings according to their role as *prefixes* (i.e. according to what can come *after* them), a CFG is able to classify strings according to their role as *infixes* (i.e. according to what can come *around* them). The CFG in (2) does not work by specifying what can come after *aa*, and what can come after *aaa*, etc.; as we have seen, any finite device that adopts this strategy is doomed. Instead, this grammar works by specifying what can appear *around* certain strings, and the crucial point is that what can appear around, say, *aabb*, is the same as what can appear around *aaabbb* — so while the pattern $a^n b^n$ does not create any interesting equivalences between prefixes, it *does* create interesting equivalences between infixes. The grammar in (2) takes advantage of this by making a two-way distinction between (i) strings of the form $a^i b^i$, which can be combined with any surroundings of the form $a^j \_\_ b^j$, and (ii) all other strings. It uses the nonterminal symbol S as a book-keeping symbol to identify strings belonging to the first class, just as FSGs use states.

To see these concepts in a more familiar form, consider the CFG in (4). A few example derivations are illustrated with tree diagrams in (5). The stringset that this grammar generates also cannot be generated by any FSG: among strings of the form *dogs** *chase** *sleep*, only those where there is exactly one occurrence of *chase* for each occurrence of *dogs* after the first one will be generated (i.e. those of the form *dogs dogs*^{n} *chase*^{n} *sleep*), so there are unboundedly many non-equivalent prefixes of the form *dogs*^{i}. The early rejection of FSGs as models of natural language was based on the claim that a grammar of English would need to generate patterns of essentially this sort (Chomsky, 1956, §2.3).

(4) start symbol: S

S	→	NP VP	N	→	*dogs*
NP	→	N	A	→	*chased*
NP	→	A N	A	→	*big*
NP	→	N RC	V	→	*chased*
RC	→	NP V	V	→	*chase*
VP	→	V NP	VP	→	*sleep*

(5)



Subsets of the set $\{\mathrm{S}, \mathrm{NP}, \mathrm{VP}, \mathrm{RC}, \mathrm{N}, \mathrm{A}, \mathrm{V}\}$ of nonterminals characterize classes of strings, grouped according to the ways they can appear as infixes. I will call these sets *inside sets*: for example, the inside set of the string *chase big dogs* is $\{\mathrm{VP}\}$, and the inside set of *chased dogs* is $\{\mathrm{VP}, \mathrm{NP}\}$; see Table 5. This partitioning of strings is the CFG's analog of an FSG's partitioning of strings by forward sets, and corresponds to the familiar notion of categorized constituents. A typical motivation for writing a grammar where a single nonterminal can derive, say, both *sleep* and *chase dogs*, is to express the idea that they are *intersubstitutable infixes*, i.e. for any two strings u and v, if $u + \!\!+$ *sleep* $+ \!\!+ v$ is generated then $u + \!\!+$ *chase dogs* $+ \!\!+ v$ is also generated. This is the sense in which CFGs carry over earlier ideas from immediate constituent analysis (e.g. Harris, 1946, 1954; Wells, 1947). For more recent discussions of this important notion see Clark (2013), Clark and Yoshinaka (2016) and Stabler (2019, §2).

Like forward sets, the inside set of a string (relative to a particular CFG) can be computed recursively from the inside sets of the string's parts. Recall that the forward set of a string $x_1x_2x_3x_4$, for example, can be computed by considering the states in the forward set of $x_1x_2x_3$, and asking which of these states have an outgoing transition emitting x_4. The situation for inside sets is similar. Considering only rules of the form '$A \to B\ C$' for ease of exposition: to compute the inside set of $x_1x_2x_3x_4$, we must consider the inside set of x_1x_2 and ask whether any of its member nonterminals can be combined with any nonterminal in the inside set of x_3x_4; but also consider combinations drawing one nonterminal from the inside set of x_1 and one from that of $x_2x_3x_4$, *plus* combinations drawing one from the inside set of $x_1x_2x_3$ and one from that of x_4. (The inside set of a length-one string x, we can assume, is just the set of nonterminals A for which the grammar contains the rule $A \to x$.)[14] The upshot is that to work out the inside set of $x_1x_2x_3x_4$, it suffices to know the inside sets of all its substrings. This is more complex than the situation for forward sets because there are multiple "splits" to consider (as opposed to only splitting into $x_1x_2x_3$ and x_4); but analogous in the important sense that inside sets tell us everything there is to know about how the grammar treats the relevant subexpression.

This freedom to split strings into pieces in arbitrary ways underlies a CFG's ability to create classes of intersubstitutable *infixes*. For example, consider the first tree in (5): the fact that we can split the *last* word off *dogs dogs chase sleep* but then split the *first* word off *dogs dogs chase* leads to the end result that the overall string is split into an infix *dogs chase* and its surroundings *dogs* __ *sleep*. The nonterminal symbol RC serves as a hinge or pivot which links together these two pieces — *dogs chase*, which can appear "inside" RC, and *dogs* __ *sleep*, which can appear "outside" it — just as the state B in Figure 2, for example, links together *someone ran and* and *ran really quickly*.

In an important sense, what distinguishes CFGs from FSGs is not directly an issue of hierarchical tree structure or constituency — the tree in Figure 3 is hierarchical, expressing part-whole relationships among constituents, in at least one sense — but rather the distinction between being able to make infix-circumfix splits and being restricted to prefix-suffix splits. Specifically, the stringsets that can be generated by a CFG

[14] This logic is the basis of the CKY algorithm, due to Cocke and Schwartz (1970), Kasami (1965) and Younger (1967); see also Hopcroft and Ullman (1979, pp.139–141).

Inside set	Example strings from the corresponding equivalence class
{S}	*dogs sleep* *dogs dogs chase sleep* *chased dogs chase dogs* *dogs chased dogs*
{NP}	*dogs* *big dogs* *dogs dogs chase* *dogs dogs dogs chase chase*
{VP}	*sleep* *chase dogs* *chased dogs dogs chase* *chased big dogs*
{RC}	*dogs chase* *dogs chased* *big dogs chased*
{N}	*dogs*
{A}	*big*
{V}	*chase*
{A, V}	*chased*
{NP, VP}	*chased dogs*
∅	*dogs dogs* *sleep chased* *big dogs chased*

Table 5: Equivalence classes induced by the grammar in (4)

but not by any FSG are those for which every CFG is "self-embedding" (Chomsky, 1959, p.167), i.e. those where some string can be substituted for a proper infix of itself, such as the way *aabb* can be substituted for *ab* in the $a^n b^n$ stringset. The relevant equivalence classes are still classes of strings, and are based on the way substrings fit into larger strings.

What sorts of stringsets *cannot* be generated by the infix-based mechanisms of a CFG? The comparison between the stringsets L_{pal} and L_{repeat} defined in (6) is instructive (Chomsky and Miller, 1963, pp.285–287). Some notation: w^R is the reverse of a string w, and $\{a, b\}^+$ is the set of all non-empty strings using the symbols *a* and *b*. So L_{pal} is the set of even-length palindromes, and L_{repeat} is the set of strings whose second half simply repeats the first half.

(6) $L_{\text{pal}} = \{ww^R \mid w \in \{a, b\}^+\} = \{aa, bb, aaaa, abba, baab, bbbb, aaaaaa, aabbaa, \dots\}$
$L_{\text{repeat}} = \{ww \mid w \in \{a, b\}^+\} = \{aa, bb, aaaa, abab, baba, bbbb, aaaaaa, aabaab, \dots\}$

There is a CFG that generates L_{pal}, shown in (7).

(7) S → *a*S*a*
S → *b*S*b*
S → *aa*
S → *bb*

It may seem at first surprising then that no CFG can generate L_{repeat}, since it is simply L_{pal} without the reversal. But the ww^R notation obscures the way that the CFG in (7) actually works. Importantly, this

CFG generates *abaaba* *not* by combining the prefix *aba* with the suffix *aba*, but rather by combining the infix *baab* with the surroundings *a* __ *a*, using the first rule shown in (7). An infix-based analysis of L_{repeat}, however, is no help.

4 Beyond context-free grammars

From the very outset there were doubts about about whether CFGs could form the basis of a theory of natural language syntax. Chomsky (1956, §4) argued that even if the generative capacity of CFGs (unlike FSGs) turned out to be sufficient for English (a question he left open), the resulting grammars would be unreasonably complex.[15] Relatedly, one motivation for considering Type 1 rules in the first place was the recognition that, in practice, linguists found uses for contextual restrictions on rewrite rules, for example to state selectional restrictions (Chomsky, 1959, p.148; Chomsky and Miller, 1963, p.295; Chomsky, 1963, p.363).

Furthermore, it has more recently been discovered that CFGs might be insufficient, even on the straightforward basis of generative capacity, to describe some natural languages. The best-known case is a construction in Swiss-German (Shieber, 1985) that exhibits *crossing dependencies* of the sort exhibited by L_{repeat} in (6); these contrast with the *nested dependencies* exhibited by L_{pal}, which are neatly handled by CFGs. See e.g. Pullum (1986), Partee et al. (1990, pp.503–505), Frank (2004), Kallmeyer (2010, pp.17–20) and Jäger and Rogers (2012) for useful discussion; ideas closely related to the crucial point about Swiss-German can be traced back to Huybregts (1976, 1984) and Bresnan et al. (1982).

But despite these reasons for looking beyond CFGs, Type 1 or *context-sensitive* grammars (CSGs) have not proven to be a particularly useful tool for linguistics; they have turned out to be "too close" to unrestricted rewriting grammars. While CSGs can generate the crossing-dependency patterns of L_{repeat} and Swiss-German, their generative capacity extends far beyond this. For example, there is even a CSG that generates $\{a^n \mid n \text{ is prime number}\}$.[16] The sense that this stringset seems not at all "language-like" plausibly stems from the property of CSGs that caused Chomsky the most concern initially: contextually-restricted rewrites produced structural descriptions that could not be interpreted along the lines of immediate constituent analysis. Immediately after showing that CSGs could generate stringsets that no CFG could generate, Chomsky (1959, p.148) surmised that "the extra power of grammars that do not meet Restriction 2 appears . . . to be a defect of such grammars, with regard to the intended interpretation". The underlying issue here is the absence of any meaningful kind of *intersubstitutability* at the core of CSGs: what distinguishes a Type 1 grammar from FSGs and CFGs is exactly the fact that the substrings derivable from a symbol A in the context $\phi\_\_\psi$ might *not* be derivable from A in another context.

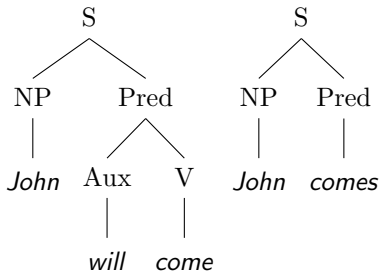
Chomsky's discussion of the undesirable properties of CSGs focuses on their ability to, in effect, reorder constituents. For example, a permuting rule 'CD → DC', which does not itself satisfy Restriction 1 (recall that the Type 0 grammar in Figure 1 contains rules like this), can be mimicked by a sequence of Type 1 rules 'CD → XD → XC → DC' (Chomsky 1959, p.148; Chomsky 1963, p.365). Chomsky considers using this kind of reordering to derive a question form such as *will John come* in a way that relates it to its corresponding declarative *John will come*. The CSG in (8) shows how this would work. The first group of rules shown in (8) generates the declaratives *John will come* and *John comes* as shown in (9); these are all context-free rules, and notice that they correctly capture the intersubstitutability of *will come* with *comes*, via the nonterminal Pred. The second group of rules in (8) serves to turn 'NP Aux' into 'Aux NP'; in particular, the derivation in (10) uses them to derive 'Aux NP *come*', and then eventually *will John come*, from the (canonically ordered, intuitively) 'NP Aux *come*'.

[15] "I do not know whether English is actually a terminal language or whether there are other actual languages that are literally beyond the bounds of phrase structure description. Hence I see no way to disqualify this theory of linguistic structure on the basis of [generative capacity]. When we turn to the question of the complexity of description . . . , however, we find that there are ample grounds for the conclusion that this theory of linguistic structure is fundamentally inadequate." (Chomsky, 1956, §4). See also Chomsky and Miller (1963, p.297).

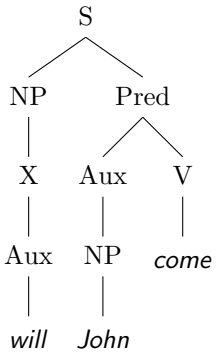
[16] This follows from the relationship between CSGs and linear bounded automata; see e.g. Hopcroft and Ullman (1979, p.225).

(8)
S → NP Pred	NP Aux → X Aux
NP → *John*	X Aux → X NP
Pred → Aux V	X NP → Aux NP
Pred → *comes*	
Aux → *will*	
V → *come*	

(9)



(10)
S
NP Pred
NP Aux V
NP Aux *come*
X Aux *come*
X NP *come*
Aux NP *come*
Aux *John come*
will John come



The fact that each step of the derivation in (10) rewrites only a single nonterminal symbol ensures that we *can* construct a tree structure that indicates which parts of the eventual string were derived from which nonterminal symbols.[17] (This would not be possible for a derivation that implemented the reordering directly with the rule 'NP Aux → Aux NP'; recall Figure 1.) But the resulting tree structure says "that *will* in this sentence is a noun phrase ... and that *John* is a modal auxiliary, contrary to our intention" (Chomsky, 1963, p.365). This result is undesirable because we do not want *will* to be *in general* intersubstitutable with *John*, or the other strings that we would expect to be derivable from the nonterminal symbol NP if this tiny grammar were expanded. So the labeled constituency relationships that can be read off the trees associated with Type 1 derivations are not interpretable as statements about intersubstitutability, as they are in more restricted grammars. In other words, Restriction 1's requirement that each derivational step rewrites only a single nonterminal symbol turned out to be insufficient to capture the important linguistic intuitions regarding categorization and intersubstitutability that underlie immediate constituent analysis.

In the light of more recent developments, the difficulties raised by the issue of reordering can be seen as stemming from the tight connection between intersubstitutability (in the sense that can be captured in rewriting systems of the sort Chomsky was exploring) and *linear contiguity*. Only linearly contiguous strings of symbols have the chance to be placed in an equivalence class. While familiar, there is nothing necessary about this connection: a sub-part of a string might belong to a class of intersubstitutable subexpressions without being contiguous. In this case, the relevant sub-parts will not themselves be strings, but will be tuples of strings. To illustrate it suffices to consider tuples of size two, i.e. pairs of strings that are co-

[17] I am leaving aside here the issues raised by rules like XZ → XYZ, which, because they satisfy Restriction 1 "in two ways", do not let us uniquely identify a tree structure for each derivation; and therefore do not even let us say which strings are derived from which nonterminal symbols in any sense. This can be overcome by simply requiring that such rules be reformulated as either 'X → XY / __ Z' or 'Z → YZ / X __'. For other discussions of the relationship between Type 1 grammars and tree structures, see Partee et al. (1990, pp.448–450) and Levelt (1974, pp.29–31).

dependent, and together constitute an expression belonging to a meaningful grammatical category, but need not be pronounced together. For example:

(11) a. The pair (*will*, *come*) and the pair (*must*, *leave*) are intersubstitutable, in the sense that we can replace the former with the latter in *will the students come* to produce *must the students leave*. (As well as in *John will come* to produce *John must leave*.)

b. The pair (*John*, *to be tall*) and the pair (*the girl*, *to win*) are intersubstitutable, in the sense that we can replace the former with the latter in *John is likely to be tall* to produce *the girl is likely to win*.[18]

c. The pair (*buy*, *which book*) and the pair (*eat*, *what*) are intersubstitutable, in the sense that we can replace the former with the latter in *which book did you buy yesterday* to produce *what did you eat yesterday*.

Chomsky's chosen approach to these phenomena, the notion of a grammatical transformation — extensively elaborated elsewhere (e.g. Chomsky, 1957, 1965) but formally somewhat removed from the work described here — was one way to resolve the tension created by the conflation of intersubstitutability and contiguity.[19] In the transformational approach, the patterns described in (11) are handled by first deriving a structure in which the co-dependent elements (e.g. *will* and *come*, or *John* and *to be tall*) are linearly contiguous, in a *base component* which functions essentially like a CFG and therefore ties contiguity and intersubstitutability together. This correctly prevents generating an expression that contains *will* without an accompanying verb like *come* (see (9)), or contains *which book* without a verb to select it, or contains the predicate *be tall* without a subject — but at the cost of grouping these co-dependent elements together in ways that do not align with their relative linear positions. The *transformational component* resolves the tension created by tying co-occurrence to contiguity, transforming a structure which has such co-dependent elements adjacent into one where they are separated.

But another logically possibility, when we are confronted with the patterns in (11), is to simply break the link between co-dependence and linear contiguity right from the beginning. Multiple context-free grammars (MCFGs) (Seki et al., 1991) provide a canonical instantiation of this option; see e.g. Kallmeyer (2010, ch.6) and Clark (2014) for overviews. Derivations in these grammars are most naturally understood in terms of a "bottom-up" composition process, unlike the "top-down" rewriting grammars that serve as the framework for the Chomsky hierarchy. MCFGs have proven to be a useful reference point for understanding and comparing various *mildly context-sensitive* grammar formalisms (Joshi, 1985; Joshi et al., 1990) which sit between CFGs and CSGs on the scale of generative capacity, including formalisms expressed in terms of transformation-like tree-manipulating operations, such as Minimalist Grammars (Stabler, 1997, 2011) and Tree-Adjoining Grammars (Joshi et al., 1975; Abeillé and Rambow, 2000; Frank, 2002).

5 Conclusion

The notion of intersubstitutability of subexpressions, or categorization of subexpressions into equivalence classes, is tightly related to the very idea of a grammar itself. Grammar formalisms differ in the ways that they compose these subexpressions (e.g. prefix-suffix combinations, infix-circumfix combinations), but this composition is mediated by categorization. Any interesting system of categorization involves isolating out the properties of a subexpression that affect its combinatory potential, and those that don't; those properties that need to be remembered or tracked, and those that can be safely ignored or forgotten. If everything is remembered and nothing is forgotten, a grammar reduces to a list of stored complete expressions (recall Figure 6); at the other extreme, a grammar that remembers nothing treats all subexpressions interchangeably, and therefore generates a set of expressions that exhibits no regularities. An interesting grammar is one that sits in between these two extremes, yielding constrained productivity.

[18] This is closely analogous to the point made by the wrap operation introduced by Bach (1979), modulo the distinction between raising and control.

[19] Chomsky (1956, §4.1) briefly mentions that the move from CFGs to transformational grammars introduces the possibility of "selecting as elements certain discontinuous strings", for example (*has*,*-en*) and (*be*,*-ing*). But this perspective on transformational grammars seems to have not been discussed much otherwise.

The overall perspective that I have offered here is somewhat more optimistic about lasting contributions of the Chomsky hierarchy than linguists have generally been since the 1960s — not more optimistic about the role string-generating grammars can play in linguistic theory, but more optimistic about the role that insights gleaned from the careful study of string-generating grammars can play in an understanding of any kind of grammar.

Acknowledgements

Thanks to Bob Frank, Bruce Hayes, Jeff Heinz, Norbert Hornstein, Kyle Johnson, Paul Pietroski and the editors and reviewers for helpful comments and suggestions on earlier drafts.

References

Abeillé, A. and Rambow, O. (2000). *Tree Adjoining Grammars*. CSLI Publications, Stanford, CA.

Bach, E. (1979). Control in Montague Grammar. *Linguistic Inquiry*, 10(4):515–531.

Bresnan, J., Kaplan, R. M., Peters, S., and Zaenen, A. (1982). Cross-serial Dependencies in Dutch. *Linguistic Inquiry*, 13(4):613–635.

Carnie, A. (2013). *Syntax: A Generative Introduction*. Wiley-Blackwell, Malden, MA, third edition.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2:137–167.

Chomsky, N. (1963). Formal properties of grammars. In Luce, R. D., Bush, R. R., and Galanter, E., editors, *Handbook of Mathematical Psychology*, volume 2, pages 323–418. John Wiley and Sons, Inc., New York and London.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Chomsky, N. (2006). *Language and Mind*. Cambridge University Press, 3rd edition.

Chomsky, N. and Miller, G. (1963). Introduction to the formal analysis of natural languages. In Luce, R. D., Bush, R. R., and Galanter, E., editors, *Handbook of Mathematical Psychology*, volume 2, pages 269–321. John Wiley and Sons, Inc., New York and London.

Chomsky, N. and Miller, G. A. (1958). Finite state languages. *Information and Control*, 1(2):91–112.

Clark, A. (2013). The syntactic concept lattice: Another algebraic theory of the context-free languages? *Journal of Logic and Computation*, 25(4):1203–1229.

Clark, A. (2014). An introduction to multiple context-free grammars for linguists. `https://alexc17.github.io/papers/mcfgsforlinguists.pdf`.

Clark, A. and Yoshinaka, R. (2016). Distributional learning of context-free and multiple context-free grammars. In Heinz, J. and Sempere, J. M., editors, *Topics in Grammatical Inference*, pages 143–172. Springer.

Cocke, J. and Schwartz, J. T. (1970). Programming languages and their compilers. Courant Institute of Mathematical Sciences, New York University.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata: Techniques and applications. Available from: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Frank, R. (2002). *Phrase Structure Composition and Syntactic Dependencies*. MIT Press, Cambridge, MA.

Frank, R. (2004). Restricting grammatical complexity. *Cognitive Science*, 28(5).

Fromkin, V., Curtiss, S., Hayes, B. P., Hyams, N., Keating, P. A., Koopman, H., Munro, P., Sportiche, D., Stabler, E. P., Steriade, D., Stowell, T., and Szabolsci, A. (2000). *Linguistics: An Introduction to Linguistic Theory*. Blackwell.

Goodman, J. T. (1999). Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Harris, Z. S. (1946). From morpheme to utterance. *Language*, 22(3):161–183.

Harris, Z. S. (1951). *Methods in Structural Linguistics*. University of Chicago Press.

Harris, Z. S. (1954). Distributional structure. *Word*, 10(2–3):146–162.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, MA.

Huybregts, R. (1976). Overlapping dependencies in dutch. In *Utrecht Working Papers in Linguistics*, number 1, pages 24–65.

Huybregts, R. (1984). The weak inadequacy of context-free phrase structure grammars. In de Haan, G. J., Trommelen, M., and Zonneveld, W., editors, *Van Periferie Naar Kern*, pages 81–99. Foris, Dordrecht.

Jäger, G. and Rogers, J. (2012). Formal language theory: refining the Chomsky hierarchy. *Philosophical Transaction of the Royal Society B*, 367:1956–1970.

Johnson, K. (2019). Transformational grammar. Unpublished course notes.

Joshi, A. (1985). How much context-sensitivity is necessary for characterizing structural descriptions? In Dowty, D., Karttunen, L., and Zwicky, A., editors, *Natural Language Processing: Theoretical, Computational and Psychological Perspectives*, pages 206–250. Cambridge University Press, New York.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree adjunct grammars. *Journal of Computer and System Sciences*, 10:136–163.

Joshi, A. K., Shanker, K. V., and Weir, D. (1990). The convergence of mildly context-sensitive grammar formalisms. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-01.

Kallmeyer, L. (2010). *Parsing Beyond Context-Free Grammars*. Springer-Verlag, Berlin Heidelberg.

Kasami, T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. AFCRL Technical Report 65-758.

Kuich, W. (1997). Semirings and formal power series: their relevance to formal languages and automata. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 1, pages 609–677. Springer.

Levelt, W. J. M. (1974). *An Introduction to the Theory of Formal Languages and Automata*, volume 1 of *Formal Grammars in Linguistics and Psycholinguistics*. Mouton.

Lewis, H. R. and Papadimitriou, C. H. (1981). *Elements of the Theory of Computation*. Prentice-Hall.

Miller, G. A. and Chomsky, N. (1963). Finitary models of language users. In Luce, R. D., Bush, R. R., and Galanter, E., editors, *Handbook of Mathematical Psychology*, volume 2. Wiley and Sons, New York.

Mohri, M. (2002). Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350.

Partee, B. H., ter Meulen, A., and Wall, R. E. (1990). *Mathematical Methods in Linguistics*. Kluwer, Dordrecht.

Pullum, G. K. (1986). Footloose and context-free. *Natural Language and Linguistic Theory*, 4(3):409–414.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125.

Rogers, J. (1997). Strict LT_2 : Regular :: Local : Recognizable. In Retoré, C., editor, *Logical Aspects of Computational Linguistics: First International Conference, LACL '96 (Selected Papers)*, volume 1328 of *Lectures Notes in Computer Science/Lectures Notes in Artificial Intelligence*, pages 366–385. Springer.

Rounds, W. C. (1970). Mappings and grammars on trees. *Mathematical systems theory*, 4(3):257–287.

Seki, H., Matsumara, T., Fujii, M., and Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343.

Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, MA.

Stabler, E. P. (1997). Derivational minimalism. In Retoré, C., editor, *Logical Aspects of Computational Linguistics*, volume 1328 of *LNCS*, pages 68–95, Berlin Heidelberg. Springer.

Stabler, E. P. (2011). Computational perspectives on minimalism. In Boeckx, C., editor, *The Oxford Handbook of Linguistic Minimalism*. Oxford University Press, Oxford.

Stabler, E. P. (2019). Three mathematical foundations for syntax. *Annual Review of Linguistics*, 5:243–260.

Thatcher, J. W. (1967). Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317–322.

Thatcher, J. W. (1973). Tree automata: An informal survey. In Aho, A. V., editor, *Currents in the Theory of Computing*, pages 143–172. Prentice-Hall.

Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81.

Wells, R. S. (1947). Immediate constituents. *Language*, 23(2):81–117.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.