PART III

Context-Free Languages

Part III
Context-Free Languages

14 Phrase-Structure Grammars

We have explored, now, two kinds of finite definitions of potentially infinite languages: regular expressions—which describe how to build languages inductively: by starting with a basis of simple languages and then building larger languages from them via some family of operations—and Finite State Automata—which describe a procedure for scanning strings from left to right categorizing them via information gathered while scanning. We now will look at a third approach—*Phrase-Structure Grammars*. These are a kind of *rewriting system*, a set of rules for forming strings by replacing one substring with another.

In general, a phrase-structure grammar consists of a finite set of *variables* (or non-terminals), which must be rewritten, a finite set of *terminals*, which may not be, a single starting symbol (a non-terminal), and a finite set of *productions* or rewrite rules.

Definition 50 *A* phrase-structure grammar G *is a four-tuple* $\langle V, \Sigma, S, P\rangle$, *where:*

$$\begin{array}{rl} V & \textit{is a finite set of } \text{variables,} \\ \Sigma & \textit{is a } \text{terminal alphabet,} \\ S \in V & \textit{is the } \text{start symbol,} \\ P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^* & \textit{is a finite set of } \text{productions.} \end{array}$$

The productions specify a rewriting relation between strings of variables and terminals which is usually expressed with an infix arrow, for example,

Example:

$$\begin{array}{rcl} S & \longrightarrow & aXa \\ S & \longrightarrow & bXb \\ aX & \longrightarrow & abXb \\ bX & \longrightarrow & baXa \\ X & \longrightarrow & c \end{array}$$

Variables are typically upper-case letters, terminals, lower-case. The start symbol is usually 'S'. Under these conventions, grammars can often be unambiguously specified by giving just the set of productions. (Here, V is $\{S, X\}$, Σ is $\{a, b, c\}$, and the start symbol is 'S'.) Often, when there are alternative ways to rewrite a given string they will be written as a single rule with the alternatives separated by '$|$'. The S rules, for example, could be expressed:

$$S \longrightarrow aXa \mid bXb$$

Note that while the left hand side of a production must be non-empty (we do not allow rewriting of nothing) the right hand side is under no such restriction. A production in which the right hand side is empty is called an ε-*production* (or a *null production*). In essence, an ε-production allows the string on the left hand side to simply be erased. Grammars with no ε-productions are referred to as *positive* grammars.

Grammars are applied in a process that begins with the start symbol and successively applies productions until only non-terminals remain:

$$\begin{gathered}
\underbrace{S}_{S\longrightarrow aXa}\\
\overbrace{\underbrace{aX}_{aX\longrightarrow abXb}\,a}\\
\overbrace{a\underbrace{bX}_{bX\longrightarrow baXa}b}\;a\\
a\,\overbrace{ba\underbrace{X}_{X\longrightarrow c}a}\,ba\\
aba\,\overbrace{c}\,aba
\end{gathered}$$

The strings of terminals that can be obtained in this way are said to be *derived* by the grammar and the sequence of strings of variables and terminals witnessing it is a *derivation.* The strings of variables and terminals occurring in a derivation (from S) are called *sentential forms.* The *language generated* by a grammar G is the set of stings of terminals that can be derived by it.

To formalize these we start by extending the rewriting relation of the productions to relations on arbitrary strings of variables and terminals:

Definition 51 *If $G = \langle V, \Sigma, S, P\rangle$ is a phrase structure grammar and α and γ are strings in $(V \cup \Sigma)^*$, then α* directly derives γ in G *($\alpha \underset{G}{\Longrightarrow} \gamma$) iff there is some production $\beta \longrightarrow \beta' \in P$ such that $\alpha = \alpha_l\beta\alpha_r$ and $\gamma = \alpha_l\beta'\alpha_r$.*

A derivation in G from α *is a sequence* $\langle \alpha_1, \ldots, \alpha_m \rangle$ *of strings over* $(V \cup \Sigma)^*$ *in which* $\alpha = \alpha_1$ *and for all* $1 < i \leq m$, $\alpha_{i-1} \underset{G}{\Longrightarrow} \alpha_i$.

A string α derives γ in G *(*$\alpha \overset{*}{\underset{G}{\Longrightarrow}} \gamma$*) iff there is a derivation* $\langle \alpha_1, \ldots, \alpha_m \rangle$ *(where* $m \geq 1$*) for which* $\alpha = \alpha_1$ *and* $\gamma = \alpha_m$.

When it is clear from the context, we will usually drop the G from the notations for derives and directly derives.

73. Given this formal definition of derivation, what would be the effect of extending P to $(V \cup \Sigma)^* \times (V \cup \Sigma)^*$, of allowing ε on the left-hand side of productions?

Definition 52 *If* $G = \langle V, \Sigma, S, P \rangle$ *is a phrase structure grammar then the* language generated by G *is*

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

15 Context-Free Grammars

The defining power of phrase-structure grammars can be restricted by restricting the form of the productions. We are interested, in particular, in productions which rewrite single non-terminals, i.e., in which the left hand side of every production is a single variable.

Definition 53 *A* Context-Free Grammar (CFG) *is a phrase-structure grammar* $\langle V, \Sigma, S, P \rangle$ *in which* $P \subseteq V \times (V \cup \Sigma)^*$.

A language is a Context-Free Language (CFL) *iff it is* $L(G)$ *for some CFG* G.

We will refer to the class of all context-free languages as CFL.

This restriction limits each derivation step to rewrite a single variable, but more importantly, there can be no restriction on when that variable may be rewritten—a production for X can be applied whenever X appears in a sentential form. The grammar of Example 14 actually rewrites single variables—X rewrites as either 'bXb' or 'aXa', for instance—but it limits rewriting as 'bXb' to 'X's that occur with an 'a' immediately to the left and rewriting as 'aXa' to contexts with 'b' immediately to the left. The term Context-Free comes from the fact that these grammars cannot restrict the context in which a production applies.

Because rewriting in CFGs expands single non-terminals to strings of terminals and non-terminals it is possible to represent a derivation as a tree in which the children of a node labeled with a non-terminal are labeled with the string to which it is rewritten. Since every non-terminal must be rewritten during the course of a (complete) derivation, the leaves of a (complete) derivation tree will necessarily be labeled with terminals or, in the case of ε-productions, with 'ε'. We can formalize this as follows:

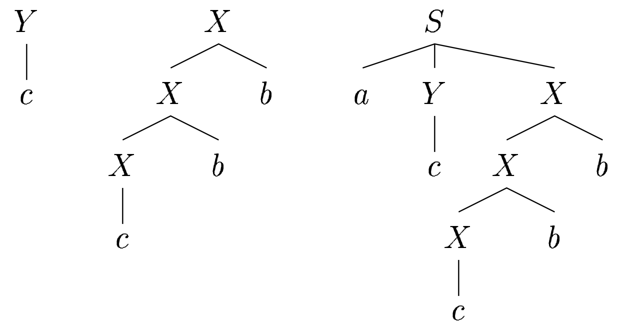
Definition 54 (Derivation tree in G) *If $G = \langle V, \Sigma, S, P\rangle$ is a CFG then:*

- *If $X \longrightarrow w \in P$ for $w \in \Sigma^*$ then the depth one tree with root labeled X and yield w is a* derivation tree for w from X in G.
- *If $X \longrightarrow w_0 Y_1 w_1 \cdots w_{n-1} Y_n w_n \in P$, for $w_0, \ldots, w_n \in \Sigma^*$, $Y_1, \ldots, Y_n \in V$, and if $\Sigma_1, \ldots, \Sigma_n$ are derivation trees from $Y_1, \ldots, Y_n$ in G, respectively, (yielding $v_1, \ldots, v_n$, respectively) then the tree obtained from the depth one tree with root labeled X and yield $w_0 Y_1 w_1 \cdots w_{n-1} Y_n w_n$ by attaching $\Sigma_1, \ldots, \Sigma_n$ as subtrees at the nodes labeled $Y_1, \ldots, Y_n$, respectively, is a* derivation tree for $w_0 v_1 w_1 \cdots w_{n-1} v_n w_n$ from X in G.
- *Nothing else is a derivation tree in G.*

Example: *Let $G = \langle V, \Sigma, S, P\rangle$, where P is:*

$$\begin{aligned} S &\longrightarrow aYX \mid YXb \\ X &\longrightarrow Xb \mid c \\ Y &\longrightarrow aY \mid c \end{aligned}$$

Then



are derivation trees for c from Y in G, for cbb from X in G and for accbb from S in G, respectively.

Lemma 19 *If $G = \langle V, \Sigma, S, P\rangle$ is a CFG then for all $X \in V$ and $w \in \Sigma^*$, there is a derivation of w from X in G iff there is a derivation tree for w from X in G.*

Proof: To show that if there is a derivation of w from X in G then there is a derivation tree for w from X in G, by induction on the length of the derivation:

(Basis:)

If $X \overset{1}{\underset{G}{\Longrightarrow}} w$ then $X \longrightarrow w \in P$ and the depth one tree with root labeled X and yield w is a derivation tree for w from X in G.

(Ind:)

Suppose X derives w in G by a derivation of $n > 1$ steps and for every derivation of any string from any non-terminal in V that is of length less than n there is a corresponding derivation tree. Since the length of the derivation of w from X is greater than one it must have the form

$$X \overset{1}{\underset{G}{\Longrightarrow}} \alpha \overset{n-1}{\underset{G}{\Longrightarrow}} w$$

for some $X \longrightarrow \alpha \in P$. Let $\alpha = w_0 Y_1 w_1 \cdots Y_n w_n$ for $w_i \in \Sigma^*$ and $Y_i \in V$. The remaining $n-1$ steps of the derivation must rewrite each of the Y_i to a (possibly empty) string of terminals and must do so in $n-1$ or fewer steps. Thus for each Y_i there is some $v_i \in \Sigma^*$ such that $Y_i \overset{<n}{\underset{G}{\Longrightarrow}} v_i$. Moreover, whatever is derived from Y_i must appear in w to the right of w_{i-1} and to the left of w_i. Consequently, $w = w_0 v_1 w_1 \cdots v_n w_n$. By the induction hypothesis there is a derivation tree for each v_i from Y_i in G. It follows, then, from the second clause of the definition of derivation trees, that there is a derivation tree for w from X in G.

To show the converse, by induction on the structure of the tree:

(Basis:)

Suppose t is a derivation tree for w from X in G as defined in the first clause of the definition of derivation trees. Then there is a production $X \longrightarrow w \in P$ and, consequently, $X \overset{1}{\underset{G}{\Longrightarrow}} w$.

(Ind:)

Suppose t is formed by the second clause of the definition and that for all simpler derivation trees there is a corresponding derivation. Then there is a production $X \longrightarrow w_0Y_1w_1\cdots Y_nw_n \in P$ and derivation trees for v_i from Y_i in G such that $w = w_0v_1w_1\cdots v_nw_n$. Since the derivation trees for the v_i are subtrees of t there are, by the IH, derivations of each of the v_i from Y_i in G. Then there is a derivation

$$X \overset{1}{\underset{G}{\Longrightarrow}} w_0Y_1w_1\cdots Y_nw_n \overset{*}{\underset{G}{\Longrightarrow}} w_0v_1w_1\cdots Y_nw_n \overset{*}{\underset{G}{\Longrightarrow}} w_0v_1w_1\cdots v_nw_n = w$$

in which each of the Y_i in turn are rewritten using the same sequence of productions as the individual derivations. ⊣

Note that it is possible to read derivations off of derivation trees by taking progressively deeper "cuts" across the tree, e.g.,

$$S \Longrightarrow aYX \Longrightarrow acX \Longrightarrow acXb \Longrightarrow acXbb \Longrightarrow accbb.$$

Each derivation tree, however, may encode several distinct derivations. Here, for example, we can also get

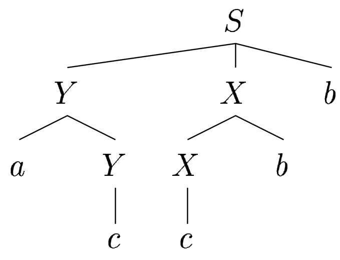
$$S \Longrightarrow aYX \Longrightarrow aYXb \Longrightarrow aYXbb \Longrightarrow aYcbb \Longrightarrow accbb$$

as well as two others. These are equivalent, though, in the sense that they all rewrite a given instance of a non-terminal in exactly the same way, the only difference between them is the order in which the productions are applied.

Definition 55 *A* left-most *derivation in G is a derivation in which, at each step, the left-most non-terminal is rewritten. A* right-most *derivation in G is one in which, at each step, the right-most non-terminal is rewritten.*

The first of the two derivations we obtained above is a left-most derivation, the second a right-most.

In contrast, if there is more than one derivation tree for a given string then these will differ in a substantial way—they must differ in the way that at least one non-terminal is rewritten. For example,



is also a derivation tree for *accbb* from S in G. When a grammar can derive a single string in two substantially distinct ways like this it is said to be *ambiguous.*

Definition 56 *A grammar G is* ambiguous *iff there is some string $w \in L(G)$ for which there are at least two distinct derivation trees for w from S in G.*

Note that it is the grammar that is ambiguous, not the language. Many languages that are generated by ambiguous grammars are also generated by unambiguous grammars. There are, however, some languages that are not generated by any unambiguous grammar. These are referred to a *inherently ambiguous languages.*

Lemma 20 *Suppose $G = \langle V, \Sigma, S, P\rangle$ is a CFG. Then for all $X \in V$ and $w \in \Sigma^*$, if t is a derivation tree for w from X in G there is both a left-most derivation of w from X in G and a right-most derivation of w from X in G corresponding to t and these derivations are unique.*

74. Prove the lemma.

Corollary 6 *A grammar G is* ambiguous *iff there is some string $w \in L(G)$ for which there are at least two distinct left-most derivations of w from S in G. (Similarly iff there are at least two distinct right-most derivations.)*

15.1 Determining the Language Generated by CFG

Consider the following CFG

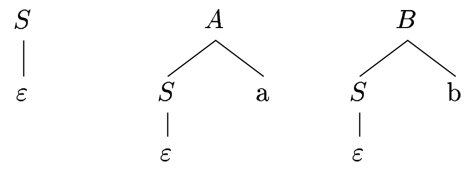
$$\begin{array}{rcl} S & \longrightarrow & aB \mid bA \mid SS \mid \varepsilon \\ A & \longrightarrow & Sa \\ B & \longrightarrow & Sb \end{array}$$

We are interested in two things:

- to describe the language this grammar generates.
- to prove that it generates exactly that language.

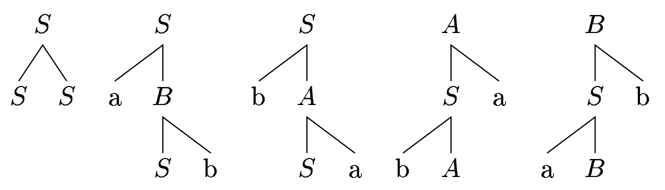
The most general approach to a problem like this is to explore a a variety of derivations in the grammar and try to get a sense of the way in which it builds strings. The key is to have a systematic approach to these explorations. CFGs are a sort of inductive definition. For any CFG there will be a class of simplest derivations—those in which there is no recursion, in which no non-terminal occurs more than once along any path from the root of the derivation tree to a leaf—and there will be a class of (minimal) recursive derivations—those in which the non-terminal at the root also occurs at a leaf. (We will not make these notions more precise here. The object is to give a way of organizing your search. The analysis is *not* part of the proof; rather it is a way of arriving at the invariants that form the foundation of the proof.)

Here the non-recursive derivations are:



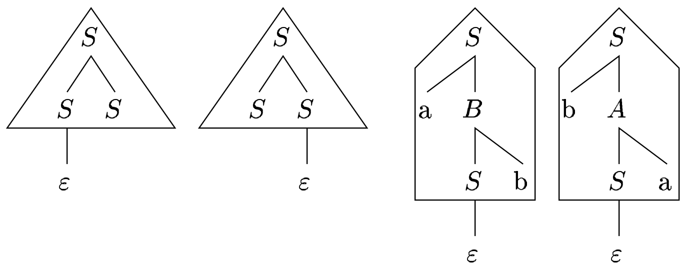
We include derivations from each of the nonterminals since we are going to develop invariants describing the languages generated from each non-terminal. Note that the basic strings derived from S, A, and B are ε, a, and b, respectively. With this particular grammar we can already note that A will derive strings of the form wa where w is some string in the language derived from S. Similarly, B derives strings of the form wb. Things are often not this simple.

One can construct any derivation tree in the grammar by starting with one of these non-recursive derivations and inserting any number of the recursive trees of the grammar:

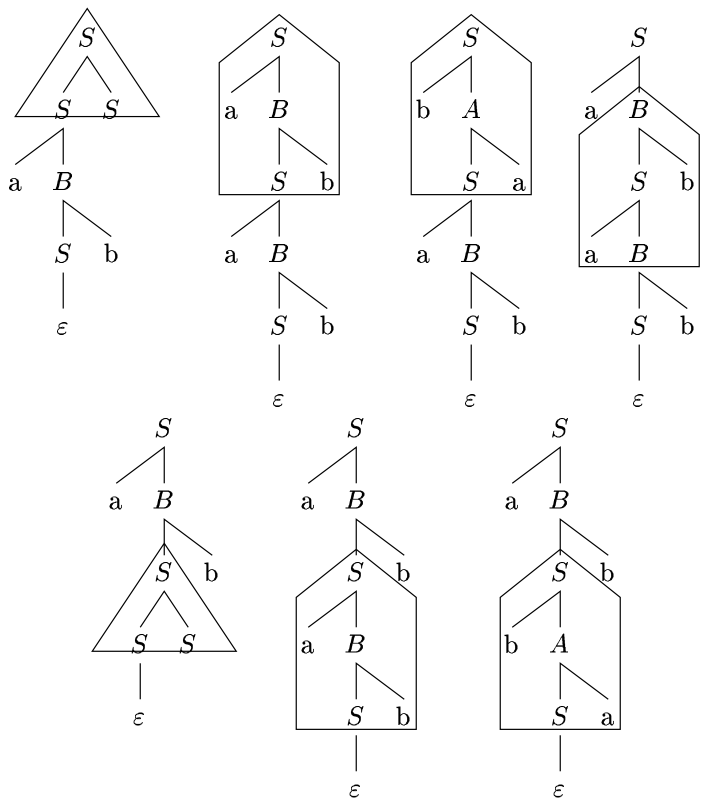


The first recursive S tree is obvious. Note, though, that there are also recursive S trees that go through expansions of B and of A. These are actually more productive, in a sense, since the first S tree just allows the iteration of strings derived from S; its effect is that of a positive Kleene closure, albeit one with complex consequences since the S may be embedded in the midst of a derivation. The other two are responsible for embedding strings in the midst of other strings.

Carrying out a few of these insertions, we can get a sense of how the strings generated from S grow. (Again, in general one needs to carry this out for all the non-terminals, not just for S.)



Carrying on for just the tree containing B:



(We have not carried out all variations of each insertion.)

One of the things that is immediately obvious from the structure of the recursive trees (but not immediately from the rules themselves) is that 'a's and 'b's are inserted in pairs. So one property we can be sure of is that their numbers will be the same in all strings derived from S. Putting this together with our understanding of the relationship between the strings derivable from S and those derivable from A and B this leads to an initial hypothesis:

Claim 5

- $S \overset{*}{\Longrightarrow} w \in \{a,b\}^* \Rightarrow |w|_a = |w|_b$.
- $A \overset{*}{\Longrightarrow} w \in \{a,b\}^* \Rightarrow w = w'a \textit{ where } |w'|_a = |w'|_b$.
- $B \overset{*}{\Longrightarrow} w \in \{a,b\}^* \Rightarrow w = w'b \textit{ where } |w'|_a = |w'|_b$.

This much we are certain is true. To get the converse, though, it may need to be strengthened; there may be more structure to the strings in these languages than we have claimed. However, if we consider the trees obtained by inserting recursive trees into the A tree as well as those obtained above by inserting into the B tree it becomes clear that we can obtain *all* strings of length up to four in which the numbers of 'a's and 'b's are equal. Thus, it appears that the converse of the claim may be true as well. In proving this we will have proven half of the characterization of the language generated by the grammar.

Claim 6

- $w \in \{a,b\}^* \textit{ and } |w|_a = |w|_b \Rightarrow S \overset{*}{\Longrightarrow} w$.
- $w \in \{a,b\}^* \textit{ and } |w|_a = |w|_b \Rightarrow A \overset{*}{\Longrightarrow} wa$.
- $w \in \{a,b\}^* \textit{ and } |w|_a = |w|_b \Rightarrow B \overset{*}{\Longrightarrow} wb$.

Proof: (By induction on $|w|$.) Suppose $w = \varepsilon$. Then

$$S \Longrightarrow \varepsilon \qquad A \Longrightarrow Sa \Longrightarrow \varepsilon a = a \qquad B \Longrightarrow Sb \Longrightarrow \varepsilon b = b.$$

Suppose $w \in \{a,b\}^*$, $|w|_a = |w|_b$ and $|w| = n$. Suppose, further, that for all such w of length less than n the claim is true. Break w into w_1w_2 where w_1 is the shortest non-empty prefix of w in which the number of 'a's and 'b's are equal. Note that this implies that the 'a's and 'b's are equinumerous in w_2 as well.
Suppose that w_2 is not empty. Then the lengths of w_1 and w_2 are both strictly less than n and they both have equal numbers of 'a's and 'b's. From the induction hypothesis, then,

$$S \Longrightarrow SS \overset{*}{\Longrightarrow} w_1S \overset{*}{\Longrightarrow} w_1w_2 = w$$

and

$$A \Longrightarrow Sa \overset{*}{\Longrightarrow} wa \qquad B \Longrightarrow Sb \overset{*}{\Longrightarrow} wb.$$

Suppose, on the other hand, w_2 is empty. Suppose $w = aw'$. (A similar analysis applies if w starts with 'b'.) Since no prefix of w has equal numbers of 'a's and 'b's, w' must be of the form $w''b$, where $|w''|_a = |w''|_b$. Then, by the IH again,

$$S \Longrightarrow aB \Longrightarrow aSb \overset{*}{\Longrightarrow} aw''b = w$$

and

$$A \Longrightarrow Sa \overset{*}{\Longrightarrow} wa \qquad B \Longrightarrow Sb \overset{*}{\Longrightarrow} wb.$$

⊣

We now need only to complete the proof of Claim 5.

Proof (Claim 5)**:** (By induction on the length of the derivation.)

(Basis:)

The base cases are just the non-recursive derivations.

$$\begin{aligned} S &\Longrightarrow \varepsilon & |\varepsilon|_a &= |\varepsilon|_b, \\ A \Longrightarrow Sa &\Longrightarrow a &= \ &\varepsilon a, \\ B \Longrightarrow Sb &\Longrightarrow b &= \ &\varepsilon b. \end{aligned}$$

(Induction:)

Suppose the claim is true for all derivations of length less than n. Then (letting '$\overset{n}{\Longrightarrow}$' denote "derives in n steps" and '$\overset{<n}{\Longrightarrow}$' "derives in less than n

steps"):

$$\begin{aligned}
S \overset{n}{\Longrightarrow} w \quad &\Rightarrow \quad S \Longrightarrow aB \overset{n-1}{\Longrightarrow} aw', \text{ or} \\
& \quad S \Longrightarrow bA \overset{n-1}{\Longrightarrow} bw', \text{ or} \\
& \quad S \Longrightarrow SS \overset{<n}{\Longrightarrow} w'S \overset{<n}{\Longrightarrow} w'w'', \\
A \overset{n}{\Longrightarrow} w \quad &\Rightarrow \quad A \Longrightarrow Sa \overset{n-1}{\Longrightarrow} w'a, \\
B \overset{n}{\Longrightarrow} w \quad &\Rightarrow \quad A \Longrightarrow Sb \overset{n-1}{\Longrightarrow} w'b.
\end{aligned}$$

In the first S case, by the induction hypothesis, $w' = w''b$ where $|w''|_a = |w''|_b$. Thus $w = aw''b$ and $|w|_a = |w|_b$.
The second case is similar, $w' = w''a$ where $|w''|_a = |w''|_b$. Thus $w = bw''a$ and $|w|_a = |w|_b$.
In the third S case, we have that both w' and w'' have equal numbers of 'a's and 'b's and, consequently, so does w.
Finally, in both the A and B case the claim follows immediately from the induction hypothesis. ⊣
It is worth noting that the way to carry out the induction is to 'grow' the derivations from the beginning. It is easy to determine the set of productions that may be the initial production of the derivation and trivially easy to determine the sentential form to which they apply. The same cannot be said for the last production of a derivation.

15.2 Non-inductive Proof that $L \subseteq L(G)$

For many simple grammars it is not necessary to actually carry out an induction to prove that every string in a given language is derivable in the grammar. For instance consider the grammar G_{ab}:

$$S \longrightarrow aSb \mid \varepsilon.$$

To show that $w \in \{a^i b^i \mid i \geq 0\} \Rightarrow S \underset{G_{ab}}{\overset{*}{\Longrightarrow}} w$ it suffices to give a derivation in G_{ab} which yields w. Since w depends on the parameter i the derivation will as well. (We will use $\overset{i}{\Longrightarrow}$ to denote "derives in i steps".)

$$\begin{aligned}
S \quad &\overset{i}{\Longrightarrow} \quad a^i S b^i \quad (S \longrightarrow aSb, \quad i \text{ times}) \\
&\Longrightarrow \quad a^i b^i \quad\;\; (S \longrightarrow \varepsilon).
\end{aligned}$$

75. Prove that $L(G_{ab}) = \{a^ib^i \mid i \geq 0\}$ by proving the remaining direction—that $L(G_{ab}) \subseteq \{a^ib^i \mid i \geq 0\}$.

Corollary 7 *The class of context-free languages is not a subset of the regular languages (*CFL $\not\subseteq$ Regular*).*

We will employ this language ($\{a^ib^i \mid i \geq 0\}$) as a *canonical* context-free language—a standard example of a language that is in CFL but not regular.

16 Some Closure Properties of the class CFL

Suppose $G_1 = \langle V_1, \Sigma, S_1, P_1 \rangle$ and $G_2 = \langle V_2, \Sigma, S_2, P_2 \rangle$ are CFGs generating L_1 and L_2, respectively, and that V_1 and V_2 are *disjoint*—they have no variables in common. (Of course, if they were not disjoint we could simply rename the variables in one set or the other, so we can assume this without loss of generality.) Consider the language $L_1 \cup L_2$. A string will be in this language iff it can be derived either from S_1 in G_1 or from S_2 in G_2. It should not be hard to see that we can obtain a CFG for $L_1 \cup L_2$ by putting G_1 and G_2 together along with a new start symbol S and productions rewriting S as either S_1 or S_2. Let

$$G = \langle V_1 \cup V_2 \cup \{S\}, \Sigma, S, P_1 \cup P_1 \cup \{S \longrightarrow S_1, S \longrightarrow S_1\} \rangle$$

We claim that $w \in L(G) \Leftrightarrow w \in L(G_1) \cup L(G_2)$.

76. Why do we need to assume that the sets of non-terminals are disjoint?

77. Prove the claim.

It follows, then, that whenever L_1 and L_2 are CFLs then their union will be a CFL as well.

Lemma 21 *The class of context-free languages is closed under union.*

Proof: Suppose L_1 and L_2 are CFLs then there must be CFGs G_1 and G_2 (with disjoint sets of non-terminals) that generate them and, as we have just established, we can combine these to form a CFG G that generates their union. Thus their union is a CFL, as it is generated by a CFG. $\dashv$

Note that the proof is *constructive*. Given any two CFLs (represented by their CFGs) we can effectively build a CFG representing their union. We will make use of this construction shortly.

Consider, now, the concatenation of L_1 and L_2, $L_1 \cdot L_2$. A string w is in this language iff it can be split into two parts $w = w_1 w_2$ where $w_1 \in L_1$ and $w_2 \in L_2$.

78. Give a construction that, from $G_1 = \langle V_1, \Sigma, S_1, P_1\rangle$ and $G_2 = \langle V_2, \Sigma, S_2, P_2\rangle$, builds a CFG G that generates $L(G_1) \cdot L(G_2)$. Argue that your construction is correct, that $w \in L(G)$ iff $w \in L(G_1) \cdot L(G_2)$.

Lemma 22 *The class of context-free languages is closed under concatenation.*

What about Kleene closure; is L_1^* a CFL? Here we will need a CFG that generates every string in L_1^i for all $i \geq 0$. We can get $L_1^0 (= \{\varepsilon\})$ simply by including the production $S \longrightarrow \varepsilon$, moreover, if we have that we can derive L_1^i from S we can derive $L_1^{i+1} (= L_1^i \cdot L_1)$ by including the production $S \longrightarrow SS_1$. This is all we need.

Claim 7 *Let* $G_1 = \langle V_1, \Sigma, S, P_1\rangle$ *and*

$$G = \langle V_1 \cup \{S\}, \Sigma, S, P_1 \cup \{S \longrightarrow \varepsilon, S \longrightarrow SS_1\}\rangle,$$

where $S \notin V_1$. *Then* $L(G) = (L(G_1))^*$.

Proof: To show that $(L(G_1))^* \subseteq L(G)$, it suffices to show that if $w \in (L(G_1))^i$, for any i, then $S \underset{G}{\overset{*}{\Longrightarrow}} w$. This we do by induction on i.

(Basis:)

$w \in (L(G_1))^0 \Rightarrow w = \varepsilon$ and $S \underset{G}{\Longrightarrow} \varepsilon$.

(Ind:)

Suppose that if $w \in (L(G_1))^i$ then $S \underset{G}{\overset{*}{\Longrightarrow}} w$. To show that if $w \in (L(G_1))^{i+1}$ then $S \underset{G}{\overset{*}{\Longrightarrow}} w$: if $w \in (L(G_1))^{i+1}$ then $w = w_1 w_2$ for some $w_1 \in (L(G_1))^i$ and $w_2 \in L(G_1)$. Then $S \underset{G}{\overset{*}{\Longrightarrow}} w_1$, by the induction hypothesis, and $S_1 \underset{G_1}{\overset{*}{\Longrightarrow}} w_2$, by the definition of $L(G_1)$, which, since G includes every production in P_1, implies $S_1 \underset{G}{\overset{*}{\Longrightarrow}} w_2$ as well. Then

$$S \underset{G}{\Longrightarrow} SS_1 \underset{G}{\overset{*}{\Longrightarrow}} w_1 S_1 \underset{G}{\overset{*}{\Longrightarrow}} w_1 w_2 = w.$$

To show that $L(G) \subseteq (L(G_1))^*$ it suffices to show that if $S \overset{*}{\underset{G}{\Longrightarrow}} w$, then $w \in (L(G_1))^i$ for some $i \geq 0$. Since we are proving a property of the strings derived in G we proceed by induction on the length of the derivation of the string.

(Basis:)

The minimal derivation from S in G is $S \underset{G}{\Longrightarrow} \varepsilon$ and $\varepsilon \in (L(G_1)^0$.

(Ind:)

Suppose $S \overset{n}{\underset{G}{\Longrightarrow}} w$, $n > 1$, and for all strings v derivable from S in G in fewer than n steps $v \in (L(G_1))^i$ for some $i \geq 0$. Since $n > 1$ the first step of the derivation of w must be $S \longrightarrow SS_1$. It follows that $w = w_1w_2$ where $S \overset{<n}{\underset{G}{\Longrightarrow}} w_1$ and $S_1 \overset{<n}{\underset{G}{\Longrightarrow}} w_2$. Since the only productions of G that are not in P_1 are the S productions and since no production introduces S except $S \longrightarrow SS_1$, the derivation of w_2 from S_1 in G must involve only productions from P_1. It follows that $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} w_2$ as well. Thus $w_1 \in (L(G_1))^i$ for some i and $w_2 \in L(G)$. It follows that $w = w_1w_2 \in (L(G_1))^{i+1}$ $\dashv$

Lemma 23 *The class of context-free languages is closed under Kleene closure.*

It should be immediately obvious that the singleton language $\{\varepsilon\}$ and the singleton languages containing just the strings of length 1 over Σ are CFLs. By definition, every regular language is obtainable from these languages by a finite sequence of unions, concatenations and Kleene closures. But we have just shown that the result of these operations when applied to CFLs is also a CFL. Consequently, every regular language must be context-free. Putting this together with Corollary 7 we get:

Corollary 8 *The class of regular languages is properly contained in the class of CFLs (*Regular $\subsetneq$ CFL*).*

16.1 Closure of CFL under substitution

A *substitution* f for an alphabet Σ is a mapping of each symbol σ of Σ to some language (not necessarily over the same alphabet), which we will denote $f(\sigma)$. For any string $w \in \Sigma$, the *image of w under f* is the set of all strings obtained by substituting for each symbol σ in w any string in $f(\sigma)$. The substitution may be non-uniform—the string replacing each occurrence of a symbol is chosen independently of the strings replacing any other occurrences of that symbol. For any language $L \in \Sigma^*$, the *image of L under f* is the union of the images of the strings in L under f, i.e.,

$$f(L) = \{f(w) \mid w \in L\}.$$

Example: Let $\Sigma = \{a, b\}$ and $f = \{a \mapsto L(a^*c), b \mapsto L(a^*b^*)\}$. Then $f(aba) = L(a^*ca^*b^*a^*c)$ and if $L = L((ab)^*)$ then $f(L) = L((a^*ca^*b^*)^*)$.

Lemma 24 *The class of context-free languages is closed under substitution by context-free languages.*

Proof (Sketch)**:** If $L \in CFL$ then $L = L(G)$ for some CFG $G = \langle V, \Sigma, S, P\rangle$. Similarly, if f is a substitution into CFL then for each $\sigma \in \Sigma$ the language $f(\sigma)$ is $L(G_\sigma)$ for some CFG $G_\sigma = \langle V_\sigma, \Sigma_\sigma, S_\sigma, P_\sigma\rangle$. Without loss of generality, the sets of non-terminals of the grammars are pairwise disjoint. The idea is to build a grammar for $f(L)$ by replacing every occurrence of a terminal σ in the productions in P with the start symbol S_σ of its corresponding grammar. This is then combined with the union of P_σ. It is not hard to see that a tree is a derivation tree of the combined grammar iff it is a derivation tree of G with each leaf labeled σ (independently) replaced with a derivation tree of G_σ. $\dashv$

79. Show that closure of CFL under substitution into CFL implies closure of CFL under union, concatenation and Kleene closure.
[Hint: Show first that $L(a + b)$, $L(ab)$ and $L(a^*)$ are context-free languages. Then for any CFLs L_a and L_b show how to obtain $L_a \cup L_b$. L_aL_b and L_a^* from these by substitution.]

16.2 Constructing Grammars for CFLs

These closure properties can often be used to simplify the task of developing a grammar for a CFL. The idea is to decompose the language into simpler languages in such a way that the decomposition can be reversed using operations that preserve context-freeness. In many cases the decomposition can reduce the language in question into trivially CF languages—either regular languages or canonical CFLs.

For example, let

$$\begin{aligned} L = \{w \in L(1^*a^*b^*c^*) \quad \mid \quad & |w|_1 \text{ odd} \Rightarrow |w|_a > 0 \text{ and } |w|_b = |w|_c \\ & |w|_1 \text{ even} \Rightarrow |w|_a = 0 \text{ and } |w|_b \neq |w|_c\}. \end{aligned}$$

Then

$$\begin{aligned} L &= L_e \cup L_o \quad L_e = \{w \in L \mid |w|_1 \text{ even}\}, L_o = \{w \in L \mid |w|_1 \text{ odd}\}. \\ \text{where } L_e &= L_{1e} \cdot L_{a+} \cdot L_{b=c}. \\ \text{where } L_{1e} &= L((11)^*). \\ \text{where } L_{a+} &= L(aa^*). \\ \text{where } L_{b=c} &= h_{bc}(L_{ab}), \quad \text{and } h_{bc} = \{a \mapsto b, b \mapsto c\}. \\ \text{where } L_o &= L_{1o} \cdot L_{b\neq c}. \\ \text{where } L_{1o} &= L(1(11)^*). \\ \text{where } L_{b\neq c} &= L_{b>c} \cup L_{b<c}. \\ \text{where } L_{b>c} &= L(bb^*) \cdot L_{bc}. \\ \text{where } L_{b<c} &= L_{bc} \cdot L(cc^*). \end{aligned}$$

Since it reduces L to regular and canonical CF languages, the decomposition, in itself, suffices to prove that L is a CFL. To construct a CFG generating L we need only to use obvious CFGs for the simple languages and to apply the constructions of the closure proofs.

$$\begin{aligned} L((11)^*) = L(G_{1e}) &\quad \text{where} \quad G_{1e} \text{ is } S_{1e} \longrightarrow 11S_{1e} \mid \varepsilon. \\ L(aa^*) = L(G_{a+}) &\quad \text{where} \quad G_{a+} \text{ is } S_{a+} \longrightarrow aS_{a*},\ S_{a*} \longrightarrow aS_{a*} \mid \varepsilon. \\ h_{bc}(L_{ab}) = L(G_{bc}) &\quad \text{where} \quad G_{bc} \text{ is } S_{bc} \longrightarrow bS_{bc}c \mid \varepsilon. \\ L_{1o} = L(G_{1o}) &\quad \text{where} \quad G_{1o} \text{ is } S_{1o} \longrightarrow 1S_{1e}. \\ L(bb^*) = L(G_{b+}) &\quad \text{where} \quad G_{b+} \text{ is } S_{b+} \longrightarrow bS_{b*},\ S_{b*} \longrightarrow bS_{b*} \mid \varepsilon. \\ L(cc^*) = L(G_{c+}) &\quad \text{where} \quad G_{c+} \text{ is } S_{c+} \longrightarrow cS_{c*},\ S_{c*} \longrightarrow cS_{c*} \mid \varepsilon. \\ L_{b>c} = L(G_{b>c}) &\quad \text{where} \quad G_{b>c} \text{ is } S_{b>c} \longrightarrow S_{b+}S_{bc}. \\ L_{b<c} = L(G_{b<c}) &\quad \text{where} \quad G_{b<c} \text{ is } S_{b<c} \longrightarrow S_{bc}S_{c+}. \\ L_{b\neq c} = L(G_{b\neq c}) &\quad \text{where} \quad G_{b\neq c} \text{ is } S_{b\neq c} \longrightarrow S_{b>c} \mid S_{b<c}. \\ L_e = L(G_e) &\quad \text{where} \quad G_e \text{ is } S_e \longrightarrow S_{1e}S_{a+}S_{bc}. \\ L_o = L(G_o) &\quad \text{where} \quad G_o \text{ is } S_o \longrightarrow S_{1o}S_{b\neq c}. \\ L = L(G) &\quad \text{where} \quad G \text{ is } S \longrightarrow S_e \mid S_o. \end{aligned}$$

In other words, G is

$$\begin{array}{ccc}
S \longrightarrow S_e \mid S_o & S_e \longrightarrow S_{1e}S_{a+}S_{bc} & S_o \longrightarrow S_{1o}S_{b\neq c} \\
S_{1e} \longrightarrow 11S_{1e} \mid \varepsilon & S_{a+} \longrightarrow aS_{a*} & S_{a*} \longrightarrow aS_{a*} \mid \varepsilon \\
S_{bc} \longrightarrow bS_{bc}c \mid \varepsilon & S_{1o} \longrightarrow 1S_{1e} & S_{b\neq c} \longrightarrow S_{b>c} \mid S_{b<c} \\
S_{b>c} \longrightarrow S_{b+}S_{bc} & S_{b<c} \longrightarrow S_{bc}S_{c+} & S_{b+} \longrightarrow bS_{b*} \\
S_{b*} \longrightarrow bS_{b*} \mid \varepsilon & S_{c+} \longrightarrow cS_{c*} & S_{c*} \longrightarrow cS_{c*} \mid \varepsilon.
\end{array}$$

80. Give a CFG for $L_{80} = \{a^ib^jc^k \mid i \neq j \text{ or } j \neq k\}$ by decomposing the language into trivial languages.

Another approach that often works is to start with an inductive definition of the language and then convert it, nearly directly, to a CFG. For example, let $L_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$. To show that this is a CFL, start by considering how strings in the language can be constructed from other strings. The simplest string in the language is just ε. Moreover, if we have strings $w_1, w_2 \in L_2$ then we can construct another string in L_2 by concatenating them. In addition, we can construct a another string by adding an 'a' to the left end of w_1 and a 'b' to its right end or *vice versa.* This may be enough. If it is not our attempt to prove that every string in L_2 is generated by the CFG we come up with will fail. Fortunately, the cases on which it fails *should* give us examples of how we need to extend the definition. Putting this together as an inductive definition:

- $\varepsilon \in L_2$.
- If $w_1, w_2 \in L_2$ then $w_1w_2 \in L_2$.
- If $w_1 \in L_2$ then $aw_1b, bw_1a \in L_2$.
- Nothing else.

We could stop here to prove that this is a complete definition of L_2, but as we are asked for a CFG we may as well postpone it. Note that the proofs will be similar in structure.

Converting this to a CFG we get:

$$\begin{array}{rlll}
\mathcal{G}_2: & S & \longrightarrow & \varepsilon \\
& S & \longrightarrow & SS \\
& S & \longrightarrow & aSb \mid bSa
\end{array}$$

(Note that the CFG and the inductive definition work in opposite directions, so to speak.)

The fact that every string in $L(\mathcal{G}_2)$ is in L_2 is nearly immediate:

81. Carry this out; show that $L(\mathcal{G}_2) \subseteq L_2$.

The proof that every string in L_2 is in $L(\mathcal{G}_2)$ follows the proof of Claim 6:

82. Carry this out; show that $L_2 \subseteq L(\mathcal{G}_2)$.

Note that this implies that this CFG and the CFG of Section 15.1 generate the same language.

83. Show that the CFG of Section 15.1 can be converted to $\mathcal{G}_2$.

84. Let L_D be the set of all strings of balanced parenthesis, i.e., $\Sigma = \{(,)\}$, each left parenthesis has a matching right parenthesis and pairs of matching parenthesis are properly nested. Examples of strings in this language:

$$()(()) \qquad () \qquad \varepsilon \qquad (((())))$$

Examples of strings not in this language:

$$(\qquad (() \qquad (())()) \qquad (()()))()$$

Show that this is a CFL by first defining it inductively.

17 Normal Forms for CFGs

We have defined the context-free languages by defining a class of phrase-structure grammars in which the form of the productions was restricted: the context-free grammars. It turns out that we can actually restrict the form of CFGs much more severely without decreasing their power to generate languages. The advantage of doing this is that in constructing proofs for CFLs we can assume the grammar that generates them has a much simpler structure, thus reducing the complexity of the proofs significantly.

17.1 Useless Symbols

We start by noting that there are circumstances under which a non-terminal is actually useless in the sense that it can play no role in any derivation of a string in Σ^* from S. Clearly, we can simplify any grammar in which such useless non-terminals occur, without affecting the language it generates, by eliminating all productions that involve them. While one might wonder why such non-terminals would ever find their way into a grammar, it is, in fact, not necessarily obvious that a given non-terminal is useless—one question we will want to resolve is whether there is an effective procedure, an algorithm, for eliminating them. We will show later that there are simplified forms for certain classes of CFGs for which there is no algorithm for converting arbitrary CFGs in the class to the simplified form.

Definition 57 *Suppose $G = \langle V, \Sigma, S, P\rangle$ is a grammar. A symbol $X \in V \cup \Sigma$ is* useless *in G if either*

- *(Unreachable) there is no sentential form $\alpha X \beta$ such that $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha X \beta$ or*
- *(Unproductive) there is no string $w \in \Sigma^*$ such that $X \overset{*}{\underset{G}{\Longrightarrow}} w$.*

In either case X can play no part in a derivation in G of any string in Σ^* and all productions in which X occurs can simply be eliminated from G.

Lemma 25 *If $L = L(G)$ for an CFG G then $L = L(G')$ for a CFG G' in which no useless symbols occur.*

We can obtain an algorithm for constructing G' from a G that possibly includes useless symbols based on inductive definitions of the sets of reachable and productive symbols.

Let Productive(G) be the set of productive symbols in G. Then

- $\Sigma \subseteq \text{Productive}(G)$.
- If $X \longrightarrow \alpha \in (P \cap (V \times \text{Productive}(G)^*))$ then $X \in \text{Productive}(G)$.
- Nothing else.

The inductive clause says if there is a production in P in which X rewrites to some string of productive symbols then X is productive.

Let Reachable(G) be the set of reachable symbols in G. Then

- $S \in \text{Reachable}(G)$.
- If $X \in \text{Reachable}(G)$ and $X \longrightarrow \alpha Y \beta \in P$, for some $\alpha, \beta \in (V \cup \Sigma)^*$ and $Y \in V \cup \Sigma$, then $Y \in \text{Reachable}(G)$.
- Nothing else.

Here the inductive clause says that if there is some production in P in which a reachable symbol X rewrites to a string including Y then Y is reachable as well.

Any inductive definition of this sort defines an algorithm which builds the set in stages starting with the base cases. Note that convergence (termination) of the algorithm is insured by the fact that the set at each stage is at least as large as it was in the previous stage. Since the set is limited to be a subset of the set of all symbols in the grammar which is, of course, finite, there can only be finitely many stages at which the set actually grows. But, if there is some stage at which the set *doesn't* grow, then it will never grow in any subsequent stage. (Why?) Hence, the algorithm converges after finitely many stages.

We then construct G' by first eliminating unproductive symbols and then eliminating unreachable symbols:

$$P'' = P \cap (\text{Productive}(G) \times (\text{Productive}(G))^*).$$

$$P' = P'' \cap (\text{Reachable}(G'') \times (\text{Reachable}(G''))^*).$$

Note that the order is critical. There may be some productive non-terminals that are reachable only via productions that involve non-productive non-terminals. If we calculate the set of reachable symbols prior to eliminating the non-productive non-terminals these will be left as useless symbols in the grammar. On the other hand, every symbol that is involved in producing some string of terminals from a reachable symbol is, itself, reachable. Thus no symbol will become non-productive when we eliminate the unreachable symbols.

85. Let G be the grammar in which P is:

$$\begin{array}{rcl} S & \longrightarrow & aSb \mid aXY \mid \varepsilon \\ X & \longrightarrow & aX \mid aS \\ Y & \longrightarrow & Yb \\ Z & \longrightarrow & Xb \end{array}$$

   (a) What is Productive(G)?
   (b) Let G_1 be the grammar in which $P_1 = P \cap (\text{Productive}(G) \times (\text{Productive}(G))^*)$. What is P_1.?
   (c) What is Reachable(G_1)?
   (d) Let G_2 be the grammar in which $P_2 = P_1 \cap (\text{Reachable}(G_1) \times (\text{Reachable}(G_1))^*)$. What is P_2?
   (e) What is Reachable(G)?
   (f) Let G_3 be the grammar with $P_3 = P \cap (\text{Reachable}(G) \times (\text{Reachable}(G))^*)$. What is P_3?
   (g) What is Productive(G_3)?
   (h) Let G_4 be the grammar with $P_4 = P_3 \cap (\text{Productive}(G_3) \times (\text{Productive}(G_3))^*)$. What is P_4?

17.2 ε-Productions

In Section 14 we defined a positive grammar as one that included no ε-productions. Clearly, the empty string can never be derived by a positive CFG; the language it generates cannot include ε.

86. Prove that if G is a positive CFG then there is no derivation of ε from any non-terminal in G.
[**Hint:** A derivation, of course, is just a sequence of strings in $(V \cup \Sigma)^*$ that are related by the directly derives relation. What can you say about the length of the strings making up such a sequence under the assumption that G is positive.]

The converse, however, is not true. It may be the case that a grammar cannot derive ε even though it does contain ε-productions.

87. Give an example of a CFG that includes an ε-production but cannot derive the empty string.
    [**Hint:** A nearly trivial grammar will do. Try one that generates the language $\{a\}$.]

Definition 58 *A language is* positive *if it does not include the empty string.*

Lemma 26 *If $L = L(G)$ is a positive CFL then $L = L(G')$ for a CFG G' with no ε-*productions *(i.e., is $L(G')$ for a positive CFG G').*

Again, we prove this by giving the construction of $G' = \langle V, \Sigma, S, P' \rangle$ from $G = \langle V, \Sigma, S, P \rangle$. Let Nullable($G$) be the set of symbols $X \in V$ such that $X \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon$. (This set is also know as the *kernel* of G, Ker(G).)

Inductively,

- $\varepsilon \in \text{Nullable}(G)$,
- If $X \longrightarrow \alpha \in P \cap (V \times (\text{Nullable}(G))^*)$ then $X \in \text{Nullable}(G)$.
- Nothing else.

In other words, ε is nullable and every non-terminal that directly derives either ε of any string of nullable non-terminals is also nullable.

To remove the ε-productions from G we 'back them up':
Let

$$\text{Nulled}_G(A \longrightarrow \alpha_1 \cdots \alpha_n) = \{A \longrightarrow \alpha'_1 \cdots \alpha'_n \mid \alpha'_i \in \{\alpha_i, \varepsilon\} \text{ if } \alpha_i \in \text{Nullable}(G), \alpha'_i = \alpha_i \text{ otherwise } \}.$$

The function Nulled$_G$ returns every variation of a production in which zero or more nullable symbols have been deleted (replaced with ε). (Note that this always includes the original production as well.) Since the rhs of a production is just a string, any symbol we replace with εwill simply disappear. (Unless, of course, it is already the only symbol left.)
Then

$$P' = \bigcup_{A \longrightarrow \alpha_1 \cdots \alpha_n \in P} [\text{Nulled}_G(A \longrightarrow \alpha_1 \cdots \alpha_n)] \cap V \times (V \cup \Sigma)^+,$$

that is, P extended with every variation of its productions in which zero or more nullable symbols have been 'nulled' and with all ε-productions removed. It is easy to show that, under the assumption that $L(G)$ is positive, $L(G) = L(G')$.

88. What happens if $L(G)$ is *not* positive?

Corollary 9 *Every CFL can be generated by a CFG with no useless symbols and no ε-productions other than, possibly, $S \longrightarrow \varepsilon$.*

Since every non-positive language is the union of a positive language and $\{\varepsilon\}$.

17.3 Unit Productions

Definition 59 *A production is a* unit production *if it is of the form $X \longrightarrow Y$, i.e., if it is in $(V \times V)$.*

Note that productions in $(V \times \Sigma)$ are *not* unit productions.

Lemma 27 *If $L = L(G)$ for a CFG G without ε-productions then $L = L(G')$ for a CFG G' without ε-, or unit productions.*

Here again we 'back up' the unit productions and then eliminate them. If $G = \langle V, \Sigma, S, P\rangle$ let $G' = G = \langle V, \Sigma, S, P'\rangle$, where:

$$P' = \{A \longrightarrow \alpha \mid A \overset{*}{\underset{G}{\Longrightarrow}} B \text{ and } B \longrightarrow \alpha \in P\} \setminus (V \times V).$$

Since there are no ε-productions in P, if $A \overset{*}{\underset{G}{\Longrightarrow}} B$ it must be through a sequence of unit productions. This simply adds an A production that directly derives everything directly derivable from B in this case. Note that this potentially *adds* a great many unit productions, since it adds a unit production for A for every one for B. These all get eliminated along with the original unit productions when we subtract the set of all unit productions over V.

Putting these together, we get:

Lemma 28 *If $L = L(G)$ for any CFG G then $L = L(G')$ for a CFG G' without useless symbols, unit productions, or ε-productions other than, possibly, $S \longrightarrow \varepsilon$.*

The transformations must be applied in the order: remove ε-productions, remove unit productions, remove non-productive symbols and finally remove unreachable symbols.

89. Why can't we remove useless symbols first?
[**Hint:** Can useless symbols be introduced in eliminating either ε- or unit productions?]

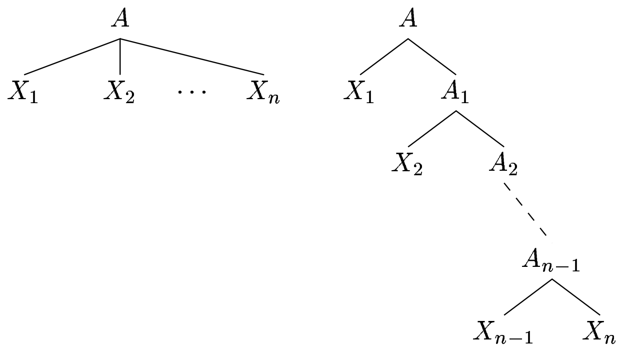
17.4 Chomsky Normal Form (CNF)

Definition 60 *A CFG is in* Chomsky Normal Form (CNF) *all productions are of the form:*

$$A \longrightarrow BC, \ \textit{for } B, C \in V \qquad \textit{or} \qquad A \longrightarrow \sigma, \ \textit{for } \sigma \in \Sigma.$$

Theorem 10 *Every positive CFL is generated by some CFG in CNF.*

Proof (Sketch)**:** Let $L = L(G)$. Convert G to a grammar with no unit or ε-productions.
Add new non-terminals X_σ for each $\sigma \in \Sigma$, replace every occurrence of σ in productions in P with X_σ and add each production of the form $X_\sigma \longrightarrow \sigma$.
Every production is now of the form $X_\sigma \longrightarrow \sigma$ or $A \longrightarrow X_1 \cdots X_n$ for $X_i \in V$ and $n > 1$. It remains to convert each one of the rules in which $n > 2$ into a sequence of binary branching rules. We do this in the same way one represents n-branching trees as binary branching trees:



where the A_i are new non-terminals unique to *each* production. ⊣

90. Convert the grammar

$$\begin{array}{lllcl} G: & S & \longrightarrow & T & \mid & S + T \\ & T & \longrightarrow & x & \mid & (S) \end{array}$$

to CNF.

Thus (with Corollary 9) every CFL is generated by a CFG in CNF, with the possible addition of the single production $S \longrightarrow \varepsilon$. In doing proofs about CFLs we will need only consider derivations in CNF grammars, and these are very simple, indeed: every production of a grammar in CNF has one of two simple forms—it rewrites a non-terminal either to exactly two non-terminals or to a single terminal. Consequently, every derivation tree will be, in essence, binary branching and every subtree in a derivation tree will yield a non-empty string.

91. Suppose G is a CFG in CNF and $S \underset{G}{\overset{*}{\Longrightarrow}} w$. Give an upper bound on the length of derivations of w from S in G.
    [**Hint:** Note that we are *not* concerned with the depth of the derivation tree but, rather, (in essence) with how many nodes it has.]

92. Give a lower bound on the length of derivations of w from S in G under the same assumptions.

93. Prove that the class of CFLs is closed under reversal:

$$L^{\mathrm{R}} = \{w^{\mathrm{R}} \mid w \in L\}.$$

    (w^{R} is defined in Definition 4 in Section 2.1.)
    [**Hint:** Let G be a CFG in CNF that generates L. Show how to transform G into G' such that $L^{\mathrm{R}} = L(G')$. CNF is not actually necessary but makes the proof somewhat simpler.]

17.5 Greibach Normal Form (GNF)

Another normal form for CFGs is due to Greibach.

Definition 61 (Greibach Normal Form (GNF)) *A CFG $G = \langle V, \Sigma, S, P\rangle$ is in* Greibach Normal Form *iff each of its productions is of the form $A \longrightarrow a\alpha$ where 'a' is a terminal and α is a (possibly empty) string of non-terminals. (Thus $P \subseteq V \times \Sigma V^*$.)*

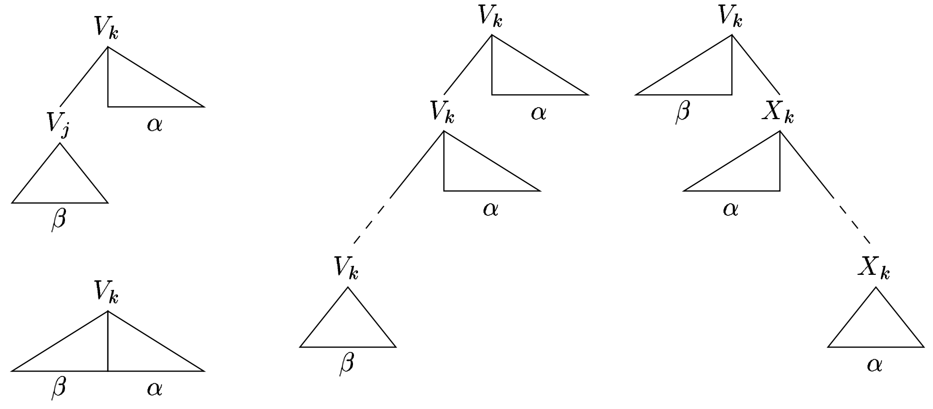
Theorem 11 (Greibach) *Every positive CFL is generated by a CFG that is in GNF.*

As with the other normal forms we have encountered so far, the proof consists of a construction reducing arbitrary CFGs to that normal form along with a proof that the construction is correct. We will give the construction along with an idea of why it works, but we will skip the actual proof of its correctness. We will also give an example of the application of the construction but, because it is quite tedious and not very instructive, you will *not* be expected to carry it out yourself. The important points you should get from this section are the structure of GNF, the fact that there is an effective procedure for reducing CFGs to GNF, and that, consequently, we can assume GNF when building proofs or algorithms for arbitrary positive CFGs.

To convert an arbitrary CFG G (for a positive language) to GNF, convert it first to G' in CNF. Let the non-terminals of G' be $V_1, V_2, \ldots V_m$ in arbitrary order. Then apply the following algorithm:

```
For each $V_k \in V$, add a new non-terminal $X_k$ to $V$
for $1 \leq k \leq m$ do
{  (A) for $1 \leq j < k$ do
   {  (A.1) for each $V_k \longrightarrow V_j\alpha$ do
      {  (A.1.1) for each $V_j \longrightarrow \beta$ do
         {  add $V_k \longrightarrow \beta\alpha$ }
         remove $V_k \longrightarrow V_j\alpha$ }
      (A.2) for each $V_k \longrightarrow V_k\alpha$ do
      {  add $X_k \longrightarrow \alpha$ and $X_k \longrightarrow \alpha X_k$
         remove $V_k \longrightarrow V_k\alpha$ }
      (A.3) for each $V_k \longrightarrow \beta$ where $\beta$ does not begin with $V_k$ do
      {  add $V_k \longrightarrow \beta X_k$ }}}
(B) for $m > k \geq 1$ (in decreasing order)
{  for each $V_k \longrightarrow V_m\alpha$ do
   {  for each $V_m \longrightarrow \beta$ do
      {  add $V_k \longrightarrow \beta\alpha$ }
      remove $V_k \longrightarrow V_m\alpha$ }}
```

The effect of block A is to transform the grammar such that if the rhs of a production for a non-terminal V starts with a non-terminal V', then V properly precedes V' in the ordering. It does this by, in essence, rotating the derivation trees of the grammar (See Figure 4.) Note that this implies that the last non-terminal in the ordering, at least, does not rewrite to any string that starts with a non-terminal. Block B then "backs-up" the productions for those initial non-terminals in the same way that block A.1 backs-up the



Block A.1:
$V_k \longrightarrow V_j\alpha,\ j < i$

Block A.2:
$V_k \longrightarrow V_k\alpha$

Figure 4: Converting to GNF

productions for V_j. Since it works from the last non-terminal in the ordering to the first the productions for the initial non-terminals will have already been backed-up. Consequently, no production will start with a non-terminal.

Applying this to the example grammar (with the non-terminals ordered $S, S_1, S_2, T, T_1, A, L, R$), we get, after the first *for* block:

$$
\begin{array}{rcllll}
S & \longrightarrow & LS_2 & \mid\ x & \mid\ LS_2X_1 & \mid\ xX_1 \\
X_1 & \longrightarrow & S_1 & \mid\ S_1X_1 & & \\
S_1 & \longrightarrow & AT & & & \\
S_2 & \longrightarrow & LS_2R & \mid\ xR & \mid\ LS_2X_1R & \mid\ xX_1R \\
T & \longrightarrow & LT_1 & \mid\ x & & \\
T_1 & \longrightarrow & LS_2R & \mid\ xR & \mid\ LS_2X_1R & \mid\ xX_1R \\
A & \longrightarrow & + & & & \\
L & \longrightarrow & (& & & \\
R & \longrightarrow &) & & &
\end{array}
$$

and after the second:

$$\begin{array}{rcllll}
S & \longrightarrow & (S_2 & \mid x & \mid (S_2X_1 & \mid xX_1 \\
X_1 & \longrightarrow & +T & \mid +TX_1 & & \\
S_1 & \longrightarrow & +T & & & \\
S_2 & \longrightarrow & (S_2R & \mid xR & \mid (S_2X_1R & \mid xX_1R \\
T & \longrightarrow & (T_1 & \mid x & & \\
T_1 & \longrightarrow & (S_2R & \mid xR & \mid (S_2X_1R & \mid xX_1R \\
A & \longrightarrow & + & & & \\
L & \longrightarrow & (& & & \\
R & \longrightarrow &) & & &
\end{array}$$

94. Suppose G is a CFG in GNF and $S \overset{*}{\underset{G}{\Longrightarrow}} w$. Give both an upper and a lower bound on the length of derivations of w from S in G.

18 Deciding Membership

The strongest attribute of GNF is the fact that every production generates at least one terminal symbol. Thus, as the exercise shows, given a CFG $\mathcal{G}$ in GNF and a string w there is a length n_w (which depends on $|w|$) such that $S \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} w$ iff $S \overset{n_w}{\underset{\mathcal{G}}{\Longrightarrow}} w$. Consequently, we can check whether $w \in L(\mathcal{G})$ simply by systematically generating all productions of $\mathcal{G}$ of length n_w and checking to see whether the last sentential form is w. Since every CFG for a positive language can be converted to GNF, we can use this solution for any CFG that generates a positive language. Moreover, it is simple to extend this to handle any CFL.

95. Give an algorithm to test if $\varepsilon \in L(\mathcal{G})$, where $\mathcal{G}$ is any CFG.
[**Hint:** Consider Section 17.2.]

96. Put this together with the ideas of this section to give a completely general algorithm that decides membership for any CFG.

18.1 Recursive Descent Parsing

While the strategy of generating all productions of a given length and testing to see if they derive w gives us an algorithm for deciding membership it leaves open the question of how to systematically generate these productions. One simple approach is start with S and apply productions in P in some fixed order, recursively applying the productions to the left-most non-terminal of the result. This systematically carries out every left-most derivation of the grammar.

We can implement this with a procedure which takes a sentential form α and a string w and returns TRUE iff it finds a derivation of w from α in $\mathcal{G}$:

```
Parse($\alpha, w$)
  if $\alpha = w = \varepsilon$ then Return(TRUE)
  if $\alpha = \sigma\beta$ and $w = \sigma v$ then Return(Parse($\beta, v$))
  if $\alpha = X\beta$ then
  {  for each $X \longrightarrow \gamma \in P$ (in order)
     {  if Parse($\gamma\beta, w$) then Return(TRUE)
        }
     }
  Else Return(FALSE)
```

The idea is that $w \in L(\mathcal{G})$ iff Parse(S, w). Unfortunately, this doesn't quite work. The problem is that it is not guaranteed to terminate and and in the case that it fails to terminate there may be derivations of w from S that have not yet been explored.

97. Carry out the procedure for Parse(S, ab) for the grammar:

$$S \longrightarrow AB \qquad A \longrightarrow Sa \qquad A \longrightarrow \varepsilon \qquad B \longrightarrow bS \qquad B \longrightarrow \varepsilon$$

with the productions ordered left to right as given.

Note that if the procedure does not terminate then there must be some branch of a derivation tree that can be arbitrary long—the recursion is not well-founded. What we need is a way of forcing the recursion to reach a base case in finitely many steps. Here we can exploit the form of GNF grammars. Note that every sentential form in a leftmost derivation of a grammar in GNF consists of an initial string of terminals followed by a string of non-terminals. Furthermore, the string of terminals in any sentential form is a strict prefix of the string of terminals in every sentential form that follows it in the derivation. Thus we are generating w in left-to-right order. The idea is to drop the portion of w that has been already generated. When we apply a production $X \longrightarrow \sigma\gamma$ to a sentential form $X\beta$ with $w = \sigma v$ we can recur with just $(\beta\gamma, v)$:

$$X\beta \Longrightarrow \sigma\gamma\beta \overset{*}{\Longrightarrow} \sigma v \quad \Leftrightarrow \quad X \longrightarrow \sigma\gamma \text{ and } \gamma\beta \overset{*}{\Longrightarrow} v.$$

Note that there is no reason to try any production with a rhs that does not start with σ. So, if we assume GNF we can modify the procedure:

```
Parse($\alpha, w$)    (Assuming GNF)
  if $\alpha = w = \varepsilon$ then Return(TRUE)
  if $\alpha = X\beta$ and $w = \sigma v$ then
  {  for each $X \longrightarrow \sigma\gamma \in P$ (in order)
     {  if Parse($\gamma\beta, v$) then Return(TRUE)
        }
     }
  Else Return(FALSE)
```

Now the length of the string we are testing is strictly decreasing as we recur. Thus we can recur no more than $|w|$ times before reach the base case at ε.