> *The product of mental labor — science — always stands far below its value, because the labor-time necessary to reproduce it has no relation at all to the labor-time required for its original production. For example, a schoolboy can learn the binomial theorem in an hour.*
>
> — Karl Marx, *Theories of Surplus Value* (1863)

> *Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: Hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "**Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.**" Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*
>
> — Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

5 Context-Free Languages and Grammars

5.1 Definitions

Intuitively, a language is regular if it can be built from individual strings by concatenation, union, and repetition. In this note, we consider a wider class of ***context-free*** languages, which are languages that can be built from individual strings by concatenation, union, and *recursion*.

Formally, a language is context-free if and only if it has a certain type of recursive description known as a ***context-free grammar***, which is a structure with the following components:

- A finite set Σ, whose elements are called ***symbols*** or ***terminals***.
- A finite set Γ disjoint from Σ, whose elements are called ***non-terminals***.
- A finite set R of ***production rules*** of the form $A \to w$, where $A \in \Gamma$ is a non-terminal and $w \in (\Sigma \cup \Gamma)^*$ is a string of symbols and variables.
- A ***starting*** non-terminal, typically denoted S.

For example, the following eight production rules describe a context free grammar with terminals $\Sigma = \{0, 1\}$ and non-terminals $\Gamma = \{S, A, B, C\}$:

$$\begin{array}{llll} S \to A & A \to 0A & B \to B1 & C \to \varepsilon \\ S \to B & A \to 0C & B \to C1 & C \to 0C1 \end{array}$$

Normally we write grammars more compactly by combining the right sides of all rules for each non-terminal into one list, with alternatives separated by vertical bars.[1] For example, the previous grammar can be written more compactly as follows:

$$\begin{aligned} S &\to A \mid B \\ A &\to 0A \mid 0C \\ B &\to B1 \mid C1 \\ C &\to \varepsilon \mid 0C1 \end{aligned}$$

[1] Yes, this means we now have *three* symbols $\cup$, $+$, and $\mid$ with exactly the same meaning. Sigh.

For the rest of this lecture, I will *almost* always use the following notational conventions.

- Monospaced digits (`0`, `1`, `2`, ...), and symbols (`◇`, `$`, `#`, `•`, ...) are explicit terminals.
- Early lower-case Latin letters ($a, b, c, \ldots$) represent unknown or arbitrary terminals in Σ.
- Upper-case Latin letters ($A, B, C, \ldots$) and the letter S represent non-terminals in Γ.
- Late lower-case Latin letters ($\ldots, w, x, y, z$) represent strings in $(\Sigma \cup \Gamma)^*$, whose characters could be either terminals or non-terminals.

We can ***apply*** a production rule to a string in $(\Sigma \cup \Gamma)^*$ by replacing any instance of the non-terminal on the left of the rule with the string on the right. More formally, for any strings $x, y, z \in (\Sigma \cup \Gamma)^*$ and any non-terminal $A \in \Gamma$, applying the production rule $A \to y$ to the string xAz yields the string xyz. We use the notation $\boldsymbol{xAz \rightsquigarrow xyz}$ to describe this application. For example, we can apply the rule $C \to \texttt{0}C\texttt{1}$ to the string $\texttt{00}C\texttt{1}BAC\texttt{0}$ in two different ways:

$$\texttt{00}\underline{C}\texttt{1}BAC\texttt{0} \rightsquigarrow \texttt{00}\underline{\texttt{0}C\texttt{1}}\texttt{1}BAC\texttt{0} \qquad \texttt{00}C\texttt{1}BA\underline{C}\texttt{0} \rightsquigarrow \texttt{00}C\texttt{1}BA\underline{\texttt{0}C\texttt{1}}\texttt{0}$$

More generally, for any strings $x, z \in (\Sigma \cup \Gamma)^*$, we say that $\boldsymbol{z}$ ***derives from*** $\boldsymbol{x}$, written $\boldsymbol{x \rightsquigarrow^* z}$, if we can transform x into z by applying a finite sequence of production rules, or more formally, if either

- $x = z$, or
- $x \rightsquigarrow y$ and $y \rightsquigarrow^* z$ for some string $y \in (\Sigma \cup \Gamma)^*$.

Straightforward definition-chasing implies that, for any strings $w, x, y, z \in (\Sigma \cup \Gamma)^*$, if $x \rightsquigarrow^* y$, then $wxz \rightsquigarrow^* wyz$.

The ***language*** $\boldsymbol{L(w)}$ of any string $w \in (\Sigma \cup \Gamma)^*$ is the set of all strings in Σ^* that derive from w:

$$L(w) := \{x \in \Sigma^* \mid w \rightsquigarrow^* x\}.$$

The language ***generated by*** a context-free grammar G, denoted $\boldsymbol{L(G)}$, is the language of its starting non-terminal. Finally, a language is ***context-free*** if it is generated by some context-free grammar.

Context-free grammars are sometimes used to model natural languages. In this context, the symbols are *words*, and the strings in the languages are *sentences*. For example, the following grammar describes a simple subset of English sentences. (Here I diverge from the usual notation conventions. Strings in ⟨angle brackets⟩ are non-terminals, and regular strings are terminals.)

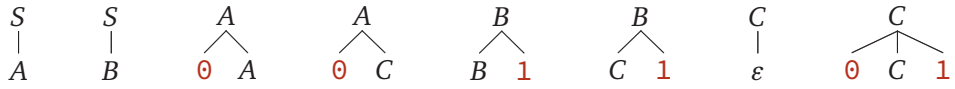
$$\begin{aligned}
\langle\text{sentence}\rangle &\to \langle\text{noun phrase}\rangle\langle\text{verb phrase}\rangle\langle\text{noun phrase}\rangle \\
\langle\text{noun phrase}\rangle &\to \langle\text{adjective phrase}\rangle\langle\text{noun}\rangle \\
\langle\text{adj. phrase}\rangle &\to \langle\text{article}\rangle \mid \langle\text{possessive}\rangle \mid \langle\text{adjective phrase}\rangle\langle\text{adjective}\rangle \\
\langle\text{verb phrase}\rangle &\to \langle\text{verb}\rangle \mid \langle\text{adverb}\rangle\langle\text{verb phrase}\rangle \\
\\
\langle\text{noun}\rangle &\to \text{dog} \mid \text{trousers} \mid \text{daughter} \mid \text{nose} \mid \text{homework} \mid \text{time lord} \mid \text{pony} \mid \cdots \\
\langle\text{article}\rangle &\to \text{the} \mid \text{a} \mid \text{some} \mid \text{every} \mid \text{that} \mid \cdots \\
\langle\text{possessive}\rangle &\to \langle\text{noun phrase}\rangle\text{'s} \mid \text{my} \mid \text{your} \mid \text{his} \mid \text{her} \mid \cdots \\
\langle\text{adjective}\rangle &\to \text{friendly} \mid \text{furious} \mid \text{moist} \mid \text{green} \mid \text{severed} \mid \text{timey-wimey} \mid \text{little} \mid \cdots \\
\langle\text{verb}\rangle &\to \text{ate} \mid \text{found} \mid \text{wrote} \mid \text{killed} \mid \text{mangled} \mid \text{saved} \mid \text{invented} \mid \text{broke} \mid \cdots \\
\langle\text{adverb}\rangle &\to \text{squarely} \mid \text{incompetently} \mid \text{barely} \mid \text{sort of} \mid \text{awkwardly} \mid \text{totally} \mid \cdots
\end{aligned}$$

5.2 Parse Trees

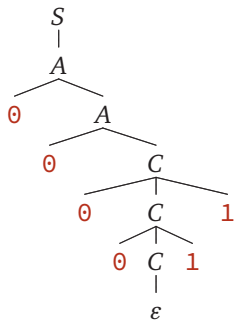
It is often useful to visualize derivations of strings in $L(G)$ using a ***parse tree***. The parse tree for a string $w \in L(G)$ is a rooted ordered tree where

- Each leaf is labeled with a terminal or the empty string ε. Concatenating these in order from left to right yields the string w.
- Each internal node is labeled with a non-terminal. In particular, the root is labeled with the start non-terminal S.
- For each internal node v, there is a production rule $A \to \omega$ where A is the label of v and the symbols in ω are the labels of the children of v in order from left to right.

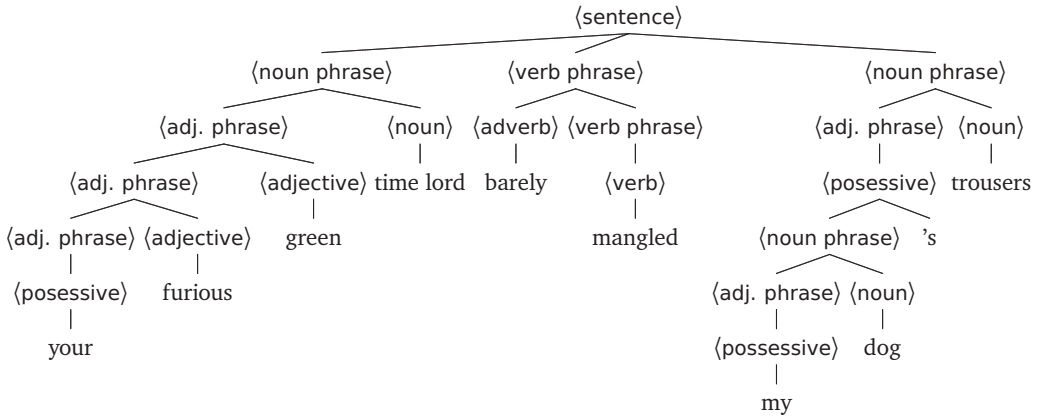
In other words, the production rules of the grammar describe *template trees* that can be assembled into larger parse trees. For example, the simple grammar on the previous page has the following templates, one for each production rule:



The same grammar gives us the following parse tree for the string `000011`:

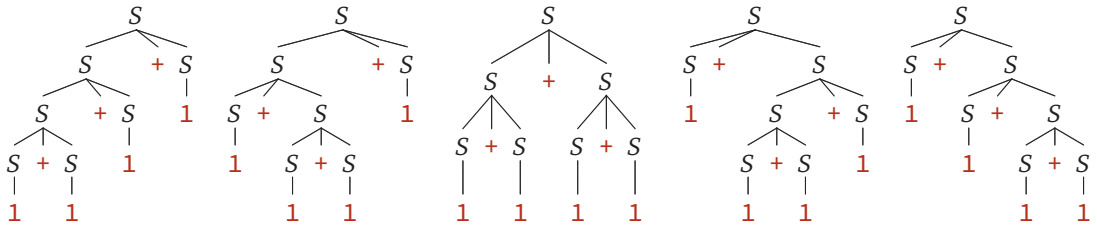


Our more complicated "English" grammar gives us parse trees like the following:



Any parse tree that contains at least one node with more than one non-terminal child corresponds to several different derivations. For example, when deriving an "English" sentence, we have a choice of whether to expand the first ⟨noun phrase⟩ ("your furious green time lord") before or after the second ("my dog's trousers").

A string w is ***ambiguous*** with respect to a grammar if there is more than one parse tree for w, and a grammar G is ***ambiguous*** is some string is ambiguous with respect to G. Neither of the previous example grammars is ambiguous. However, the grammar $S \to 1 \mid S{+}S$ is ambiguous, because the string $1{+}1{+}1{+}1$ has five different parse trees:



A context-free language L is ***inherently ambiguous*** if every context-free grammar that generates L is ambiguous. The language generated by the previous grammar (the regular language $(1{+})^*1$) is *not* inherently ambiguous, because the unambiguous grammar $S \to 1 \mid 1{+}S$ generates the same language.

5.3 From Grammar to Language

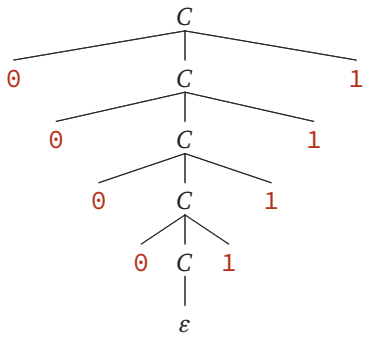
Let's figure out the language generated by our first example grammar

$$S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad C \to \varepsilon \mid 0C1.$$

Since the production rules for non-terminal C do not refer to any other non-terminal, let's begin by figuring out $L(C)$. After playing around with the smaller grammar $C \to \varepsilon \mid 0C1$ for a few seconds, you can probably guess that its language is $\{\varepsilon, 01, 0011, 000111, \ldots\}$, that is, the set all of strings of the form 0^n1^n for some integer n. For example, we can derive the string 00001111 from the start non-terminal S using the following derivation:

$$C \rightsquigarrow 0C1 \rightsquigarrow 00C11 \rightsquigarrow 000C111 \rightsquigarrow 0000C1111 \rightsquigarrow 0000\varepsilon1111 = 00001111$$

The same derivation can be viewed as the following parse tree:



In fact, it is not hard to *prove* by induction that $L(C) = \{0^n1^n \mid n \geq 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.

Lemma 5.1. *$C \rightsquigarrow^* 0^n1^n$ for every non-negative integer n.*

Proof: Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* 0^k1^k$ for every non-negative integer $k < n$. There are two cases to consider.

- If $n = 0$, then $\mathtt{0}^n\mathtt{1}^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
- Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* \mathtt{0}^{n-1}\mathtt{1}^{n-1}$. Thus, the rule $C \to \mathtt{0}C\mathtt{1}$ implies that $C \rightsquigarrow \mathtt{0}C\mathtt{1} \rightsquigarrow^* \mathtt{0}(\mathtt{0}^{n-1}\mathtt{1}^{n-1})\mathtt{1} = \mathtt{0}^n\mathtt{1}^n$.

In both cases, we conclude that that $C \rightsquigarrow^* \mathtt{0}^n\mathtt{1}^n$, as claimed. □

Lemma 5.2. *For every string $w \in L(C)$, we have $w = \mathtt{0}^n\mathtt{1}^n$ for some non-negative integer n.*

Proof: Fix an arbitrary string $w \in L(C)$. Assume that for any string $x \in L(C)$ such that $|x| < |w|$, we have $x = \mathtt{0}^k\mathtt{1}^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.

- If $w = \varepsilon$, then $w = \mathtt{0}^0\mathtt{1}^0$.
- Otherwise, $w = \mathtt{0}x\mathtt{1}$ for some string $x \in L(C)$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = \mathtt{0}^k\mathtt{1}^k$ for some integer k. Then we have $w = \mathtt{0}^{k+1}\mathtt{1}^{k+1}$.

In both cases, we conclude that $w = \mathtt{0}^n\mathtt{1}^n$ for some non-negative integer n, as claimed. □

The first proof uses induction on strings, following the boilerplate proposed in the very first lecture; in particular, the case analysis mirrors the recursive definition of "string". The second proof uses *structural induction* on the parse tree of the string $\mathtt{0}^n\mathtt{1}^n$; the case analysis mirrors the recursive definition of the language of S, as described by the production rules. In both proofs, as in every proof by induction, the inductive hypothesis is "Assume there is no smaller counterexample."

Similar analysis implies that $L(A) = \{\mathtt{0}^m\mathtt{1}^n \mid m > n\}$ and $L(B) = \{\mathtt{0}^m\mathtt{1}^n \mid m < n\}$, and therefore $L(S) = \{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$.

5.3.1 Careful With Those Epsilons

There is an important subtlety in the proof of Lemma 5.2. The proof is written as induction on the length of the string w; unfortunately, this induction pattern does not work for all context-free grammars. Consider the following ambiguous grammar

$$S \to \varepsilon \mid SS \mid \mathtt{0}S\mathtt{1} \mid \mathtt{1}S\mathtt{0}.$$

A bit of experimentation should convince you that $L(S)$ is the language of all binary strings with the same number of 0s and 1s. We cannot use the string-induction boilerplate for this grammar, because there are arbitrarily long[2] derivations of the form

$$S \rightsquigarrow SS \rightsquigarrow S \rightsquigarrow SS \rightsquigarrow S \rightsquigarrow SS \rightsquigarrow SS \rightsquigarrow \cdots \rightsquigarrow w,$$

which alternately apply the productions $S \to SS$ and $S \to \varepsilon$. Specifically, even if we knew that our arbitrary string w can be written as xy for some strings $x, y \in L(S)$, we cannot guarantee that $|x| < |w|$ and $|y| < |w|$, to we cannot apply the standard string-induction hypothesis.

However, we can still argue inductively about this grammar, by considering a *minimum-length* derivation of w, and basing the case analysis on the first production in this derivation. Here's an example of this induction boilerplate in action, with the modified boilerplate language highlighted.

[2] but not infinite; derivations are finite *by definition*!

Lemma 5.3. *For every string $w \in L(S)$, we have $\#(0, w) = \#(1, w)$.*

Proof: Let w be an arbitrary string in $L(S)$. Fix an minimum-length derivation of w.

Assume that for any string $x \in L(S)$ that is shorter than w, we have $x = 0^k 1^k$ for some non-negative integer k. There are four cases to consider, depending on the first production in our fixed derivation.

- Suppose the first production is $S \to \varepsilon$. Then $w = \varepsilon$ and therefore $\#(0, w) = \#(1, w) = 0$ by definition.
- Suppose the first production is $S \to SS$. Then $w = xy$ for some strings $x, y \in L(S)$. Both x and y must be non-empty; otherwise, we could shorten our derivation of w. Thus, both x and y are shorter than w. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ and $\#(0, y) = \#(1, y)$, so $\#(0, w) = \#(0, x) + \#(0, y) = \#(1, x) + \#(1, y) = \#(1, w)$.
- Suppose the first production is $S \to 0S1$. Then $w = 0x1$ for some string $x \in L(S)$. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ so $\#(0, w) = \#(0, x) + 1 = \#(1, x) + 1 = \#(1, w)$.
- Finally, suppose the first production is $S \to 1S0$. Then $w = 1x0$ for some string $x \in L(S)$. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ so $\#(0, w) = \#(0, x) + 1 = \#(1, w) + 1 = \#(1, w)$.

In all cases, we conclude that $\#(0, w) = \#(1, w)$, as claimed. □

Another (more traditional) way to handle this issue is to fix an *arbitrary* derivation, and then induct on the length of the derivation, rather than the length of the string itself. The case analysis is still based on the first production in the chosen derivation.

In fact, this subtlety only matters for grammars that either contain a *nullable* non-terminal A such that $A \rightsquigarrow^* \varepsilon$ or *equivalent* nonterminals A and B such that $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. We describe algorithms to identify these pathologies and remove them from the grammar (without changing its language) in Section 5.9 below.

5.3.2 Mutual Induction

Another pitfall in induction proofs for context-free languages is that non-terminals may invoke each other. Consider, for example, the grammar

$$S \to 0A1 \mid \varepsilon \qquad A \to 1S0 \mid \varepsilon$$

Because each non-terminal appears on the right side of a production rule for the other, we must argue about $L(S)$ and $L(A)$ simultaneously.

Lemma 5.4. $L(S) = (01)^*$.

Proof: We actually prove simultaneously that $L(S) = (01)^*$ and $L(A) = (10)^*$.

First, we claim that for any non-negative integer n, we have $(01)^n \in L(S)$ and $(10)^n \in L(A)$. Let n be an arbitrary non-negative integer, and assume, for all non-negative integers $m < n$, that $(01)^m \in L(S)$ and $(10)^m \in L(A)$. There are two cases to consider.

- If $n = 0$, the production rules $S \to \varepsilon$ and $A \to \varepsilon$ immediately imply $S \rightsquigarrow \varepsilon = (01)^n$ and $A \rightsquigarrow \varepsilon = (10)^n$.

- Suppose $n > 0$. We easily observe that $(\texttt{01})^n = \texttt{0}(\texttt{10})^{n-1}\texttt{1}$, so the production rule $S \to \texttt{0}A\texttt{1}$ and inductive hypothesis imply $S \rightsquigarrow \texttt{0}A\texttt{1} \rightsquigarrow^* (\texttt{01})^n$. Symmetrically, $(\texttt{10})^n = \texttt{1}(\texttt{01})^{n-1}\texttt{0}$, so the production rule $A \to \texttt{1}S\texttt{0}$ and the inductive hypothesis implies $A \rightsquigarrow \texttt{1}S\texttt{0} \rightsquigarrow^* (\texttt{10})^n$.

Next we claim that for every string $w \in L(S)$, we have $w = (\texttt{01})^n$ for some non-negative integer n, and for every string $w \in L(A)$, we have $w = (\texttt{10})^n$ for some non-negative integer n. The proof requires two stages.

- Let w be an arbitrary string in $L(S)$, and assume for all $x \in L(A)$ such that $|x| < |w|$ that $x = (\texttt{10})^n$ for some non-negative integer n. There are two cases to consider.
  - If $w = \varepsilon$, then $w = (\texttt{01})^0$.
  - Suppose $w = \texttt{0}x\texttt{1}$ for some string $x \in L(A)$. The inductive hypothesis implies $x = (\texttt{10})^n$ for some non-negative integer n. It follows that $w = \texttt{0}(\texttt{10})^n\texttt{1} = (\texttt{01})^{n+1}$.
- Let w be an arbitrary string in $L(A)$, and assume for all $x \in L(S)$ such that $|x| < |w|$ that $x = (\texttt{01})^n$ for some non-negative integer n. There are two cases to consider.
  - If $w = \varepsilon$, then $w = (\texttt{10})^0$.
  - Suppose $w = \texttt{1}x\texttt{0}$ for some string $x \in L(S)$. The inductive hypothesis implies $x = (\texttt{01})^n$ for some non-negative integer n. It follows that $w = \texttt{1}(\texttt{01})^n\texttt{0} = (\texttt{10})^{n+1}$.

Together these two claims imply $L(S) = (\texttt{01})^*$ and $L(A) = (\texttt{10})^*$, as required. □

5.4 More Examples

Here are some more examples of context-free languages and grammars that generate them, along with brief sketches of correctness proofs.

- Palindromes in $\{\texttt{0}, \texttt{1}\}^*$:

$$S \to \texttt{0}S\texttt{0} \mid \texttt{1}S\texttt{1} \mid \texttt{0} \mid \texttt{1} \mid \varepsilon$$

  This grammar is a straightforward translation of the recursive definition of palindrome.

- Strings in $(\texttt{0} + \texttt{1})^*$ that are *not* palindromes.

$$S \to \texttt{0}S\texttt{0} \mid \texttt{1}S\texttt{1} \mid \texttt{0}Z\texttt{1} \mid \texttt{1}Z\texttt{0}$$
$$Z \to \varepsilon \mid \texttt{0}Z \mid \texttt{1}Z$$

  A string w is a non-palindrome if and only if $w = x\texttt{0}z\texttt{1}x^R$ or $w = x\texttt{1}z\texttt{0}x^R$ for some (possibly empty) strings x and y.

- Strings in $\{\texttt{0}, \texttt{1}\}^*$ with the same number of $\texttt{0}$s and $\texttt{1}$s:

$$S \to \texttt{0}S\texttt{1} \mid \texttt{1}S\texttt{0} \mid SS \mid \varepsilon$$

  A non-empty string w has the same number of $\texttt{0}$s and $\texttt{1}$s if and only one of the following conditions holds:
  - We can write $w = xy$ for some non-empty strings x and y such that $\#(\texttt{0}, x) = \#(\texttt{1}, x)$ and $\#(\texttt{0}, y) = \#(\texttt{1}, y)$.

  - $\#(0,x) > \#(1,x)$ for every non-empty proper prefix x of w. In this case, $w = 0z1$ for some string z with $\#(0,z) = \#(1,z)$.
  - $\#(0,x) < \#(1,x)$ for every non-empty proper prefix x of w. In this case, $w = 1z0$ for some string z with $\#(0,z) = \#(1,z)$.

- Strings in $\{0,1\}^*$ with the same number of 0s and 1s, again:

$$S \to 0S1S \mid 1S0S \mid \varepsilon$$

  Let w be any non-empty string such that $\#(0,w) = \#(1,w)$, let x be the shortest non-empty prefix of w such that $\#(0,x) = \#(1,x)$, and let y be the complementary suffix of w, so $w = xy$. It is not hard to prove that x begins and ends with different symbols, so either $w = 0z1y$ or $w = 1z0y$, where $\#(0,y) = \#(1,y)$ and $\#(0,z) = \#(1,z)$.

- Strings in $\{0,1\}^*$ in which the number of 0s is greater than or equal to the number of 1s:

$$S \to 0S1 \mid 0S \mid 1S0 \mid S0 \mid SS \mid \varepsilon \qquad\qquad S \to 0S1S \mid 0SS \mid 1S0S \mid S0S \mid \varepsilon$$

  We have to different grammars, each constructed from a grammar for strings with equal 0s and 1s by either dropping the 1 or keeping the 1 from the right side of each production rule containing a 1. For example, we split the production rule $S \to 0S1$ in the first grammar into two production rules $S \to 0S1$ and $S \to 0S$.

  If we add the trivial production $S \to 0$ to the first grammar, we can remove two redundant productions to get the simpler grammar

$$S \to 0S1 \mid 1S0 \mid SS \mid 0 \mid \varepsilon$$

- Strings in $\{0,1\}^*$ with *different* numbers of 0s and 1s:

$$\begin{aligned} S &\to O \mid I && \text{(different)} \\ O &\to E0O \mid E0E && \text{(more 0s)} \\ I &\to E1I \mid E1E && \text{(more 1s)} \\ E &\to 0E1E \mid 1E0E \mid \varepsilon && \text{(equal)} \end{aligned}$$

  We can argue correctness by considering each non-terminal in turn, in reverse order.

  - E generates all strings with the same number of 0s and 1s, as in the previous example.
  - I generates all strings with more 1s than 0s. Any such string can be decomposed into its longest prefix with the same number of 0s and 1s (E), followed by a 0, followed by a suffix with at least as many 0s as 1s (I or E).
  - Symmetrically, O generates all strings with more 0s than 1s.
  - Finally, S generates all strings with different numbers of 0s and 1s. Any such string either has more 0s (O) or more 1s (I)

- Balanced strings of parentheses:

$$S \to (S) \mid SS \mid \varepsilon \qquad \text{or} \qquad S \to (S)S \mid \varepsilon$$

  Here we have two grammars for the same language. The first one uses simpler productions, and is a bit closer to the natural recursive definition. However, the first grammar is

ambiguous — consider the string `()()()` — while the second grammar is not. The second grammar decomposes any balanced string of parentheses into its shortest non-empty balanced prefix, which must start with `(` and end with `)`, and the remaining suffix, which must be balanced.

- Unbalanced strings of parentheses—the complement of the previous language:

$$\begin{aligned}
S &\to L \mid RX && \text{(unbalanced)}\\
L &\to E\texttt{(}L \mid E\texttt{(}E && \text{(more left parens)}\\
R &\to E\texttt{)}R \mid E\texttt{)}E && \text{(more right parens)}\\
E &\to \varepsilon \mid \texttt{(}E\texttt{)}E \mid \texttt{)}E\texttt{(}E && \text{(equal left and right)}\\
X &\to \varepsilon \mid \texttt{(}X \mid \texttt{)}X && \text{(anything)}
\end{aligned}$$

  A string w of parens is balanced if and only if both (a) w has the same number of left and right parens and (b) no prefix of w has more right parens than left parens. (Proving this fact is a good homework exercise.) Thus, a string w of parens is *unbalanced* if and only if *either* w has more left parens than right parens *or* some prefix of w has more right parens than left parens.

- Arithmetic expressions, possibly with redundant parentheses, over the variables `X` and `Y`:

$$\begin{aligned}
E &\to E\texttt{+}T \mid T && \text{(expressions)}\\
T &\to T\texttt{×}F \mid F && \text{(terms)}\\
F &\to \texttt{(}E\texttt{)} \mid \texttt{X} \mid \texttt{Y} && \text{(factors)}
\end{aligned}$$

  Every *E*xpression is a sum of *T*erms, every *T*erm is a product of *F*actors, and every *F*actor is either a variable or a parenthesized *E*xpression.

- Regular expressions over the alphabet $\{\texttt{0}, \texttt{1}\}$ *without* redundant parentheses

$$\begin{aligned}
S &\to T \mid T\texttt{+}S && \text{(Regular expressions)}\\
T &\to F \mid FT && \text{(Terms = summable expressions)}\\
F &\to \texttt{Ø} \mid W \mid \texttt{(}T\texttt{+}S\texttt{)} \mid X\texttt{*} \mid \texttt{(}Y\texttt{)*} && \text{(Factors = concatenable expressions)}\\
X &\to \texttt{Ø} \mid \texttt{Ɛ} \mid \texttt{0} \mid \texttt{1} && \text{(Directly starrable expressions)}\\
Y &\to T\texttt{+}S \mid F\texttt{•}T \mid X\texttt{*} \mid \texttt{(}Y\texttt{)*} \mid ZZ && \text{(Starrable expressions needing parens)}\\
W &\to \texttt{Ɛ} \mid Z && \text{(Words = strings)}\\
Z &\to \texttt{0} \mid \texttt{1} \mid ZZ && \text{(Non-empty strings)}
\end{aligned}$$

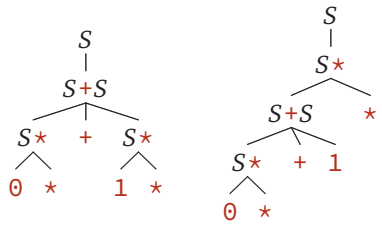
  Every regular expression is a sum of terms; every term is a concatenation of factors. Every factor is either the empty-set symbol, a string, a nontrivial sum of terms in parens, or a starred expression. The expressions `Ø*`, `Ɛ*`, `0*`, and `1*` require no parentheses; otherwise, the starred subexpression is either a nontrivial sum of terms, a nontrivial concatenation of factors, a starred expression, or a string of length 2 or more.

  The "epsilon" symbol `Ɛ` in the production rules for W and Z does *not* represent the empty string *per se*, but rather an actual symbol that might appear in a regular expression. The empty string is not a regular expression, but the one-symbol string `Ɛ` is a regular expression that represents the set containing only the empty string!

The final grammar illustrates an important subtlety for certain applications of context-free grammars. This grammar is considerably more complicated than one might initially expect from the definition of regular languages. It's tempting to suggest a much simpler grammar like

$$S \to \emptyset \mid \varepsilon \mid 0 \mid 1 \mid S+S \mid SS \mid S^* \mid (S)$$

but this is incorrect! This grammar does correctly generate all regular expressions *as raw strings*, but it allows parse trees that do not respect the *meaning* of the regular expression. For example, this "simpler" grammar can parse the regular expression string 0^*+1^* in two different ways:



The first tree correctly parses the regular expression string 1^*+0^* as the expression $(1^*)+(0^*)$ but without the redundant parentheses. The second tree incorrectly parses the same string as $(1^*+0)^*$, which describes a *very* different regular language!

5.5 Regular Languages are Context-Free

The following inductive argument proves that every regular language is also a context-free language. Let L be an arbitrary regular language, encoded by some regular expression R. Assume that any regular expression simpler than R represents a context-free language. ("Assume no smaller counterexample.") We construct a context-free grammar for L as follows. There are several cases to consider.

- Suppose L is empty. Then L is generated by the trivial grammar $S \to S$.
- Suppose $L = \{w\}$ for some string $w \in \Sigma^*$. Then L is generated by the grammar $S \to w$.
- Suppose L is the union of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 \cup L_2$ is generated by the production rule $S \to S_1 \mid S_2$.
- Suppose L is the concatenation of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 L_2$ is generated by the production rule $S \to S_1 S_2$.
- Suppose L is the Kleene closure of some regular language L_1. The inductive hypothesis implies that L_1 is context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1. Then $L = L_1^*$ is generated by the production rule $S \to \varepsilon \mid S_1 S$.

In every case, we have found a context-free grammar that generates L, which means L is context-free.

In the previous lecture note, we proved that the context-free language $\{0^n 1^n \mid n \geq 0\}$ is not regular. (In fact, this is the *canonical example* of a non-regular language.) Thus, context-free grammars are strictly more expressive than regular expressions.

5.6 Not Every Language is Context-Free

Again, you may be tempted to conjecture that *every* language is context-free, but a variant of our earlier cardinality argument implies that this is not the case.

Any context-free grammar over the alphabet Σ can be encoded as a string over the alphabet $\Sigma \cup \Gamma \cup \{\varepsilon, \to, |, \$\}$, where $ indicates the end of the production rules for each non-terminal. For example, our example grammar

$$S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad C \to \varepsilon \mid 0C1$$

can be encoded as the string

S→A|B$A→0A|0C$B→B1|C1$C→ε|0C1$

We can further encode any such string as a *binary* string by associating each symbol in the set $\Sigma \cup \Gamma \cup \{\varepsilon, \to, |, \$\}$ with a different binary substring. Specifically, if we encode each of the grammar symbols $\varepsilon, \to, |, \$$ as a string of the form 11^*0, each terminal in Σ as a string of the form 011^*0, and each non-terminal as a string of the form 0011^*0, we can unambiguously recover the grammar from the encoding. For example, applying the code

ε ↦ 10	0 ↦ 010	S ↦ 0010
→ ↦ 110	1 ↦ 0110	A ↦ 00110
| ↦ 1110		B ↦ 001110
$ ↦ 11110		C ↦ 0011110

transforms our example grammar into the 136-bit string

00101100011011100011101111000110
11001000110111001000111101111000
11101100011100110111000111100110
11110001111011010111001000111100
11011110.

Adding a 1 to the start of this bit string gives us the binary encoding of the integer

102 231 235 533 163 527 515 344 124 802 467 059 875 038.

Our construction guarantees that two different context-free grammars over the same alphabet (ignoring changing the names of the non-terminals) yield different positive integers. Thus, the set of context-free grammars over any alphabet is *at most* as large as the set of integers, and is therefore countably infinite. (Most integers are not encodings of context-free grammars, but that only helps us.) It follows that the set of all context-free *languages* over any fixed alphabet is also countably infinite. But we already showed that the set of *all* languages over any alphabet is uncountably infinite. So almost all languages are non-context-free!

There are techniques for proving that specific languages are not context-free, just as there are for proving certain languages are not regular; unfortunately, they are beyond the scope of this course. In particular, the $\{0^n1^n0^n \mid n \geq 0\}$ is not context-free. (In fact, this is the *canonical example* of a non-context-free language.)

*5.7 Recursive Automata

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* languages; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata. Recursive automata were introduced by Walter Woods in 1970 for natural language parsing; Wodds' terminology *recursive transition networks* is more common among computational linguists.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$
- A set $M = \{M_A \mid A \in N\}$ of NFAs, called ***modules***, over the alphabet $\Sigma \cup N$. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \emptyset$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A unique ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A nondeterministic ***transition function*** $\delta_A : Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

- A ***read*** transition consumes the first symbol in the input and changes the local state within the active module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the active module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses a module name A, pushes some state in $\delta(q, A)$ onto the stack, and then changes the local state to s_A (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of the active module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

$$\begin{aligned}
\textbf{read:} \quad & (ax, q, \sigma) \longmapsto (x, q', \sigma) && \text{for some } q' \in \delta(q, a) \\
\textbf{epsilon:} \quad & (w, q, \sigma) \longmapsto (w, q', \sigma) && \text{for some } q' \in \delta(q, \varepsilon) \\
\textbf{call:} \quad & (w, q, \sigma) \longmapsto (w, s_A, q' \cdot \sigma) && \text{for some } A \in N \text{ and some } q' \in \delta(q, A) \\
\textbf{return:} \quad & (w, t_A, q \cdot \sigma) \longmapsto (w, q, \sigma)
\end{aligned}$$

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

★★★

Reformulate recursive automata using recursion-as-magic, analogously to the nondeterminism-as-magic in string NFAs or Han and Wood's expression automata. A recursive automaton module M_A accepts a string w if any only if there is a finite sequence of transitions

$$s_A = q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_\ell} q_\ell = t_A$$

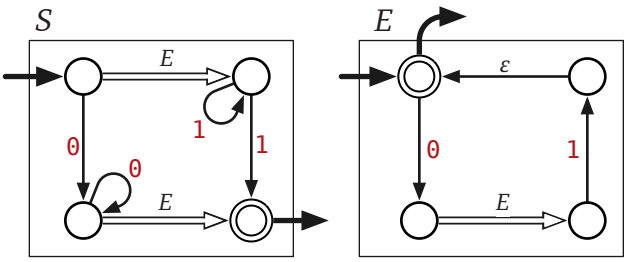
among states in Q_A, and a decomposition of w into substrings $x_1 \bullet x_2 \bullet \cdots \bullet x_\ell$, where one of the following conditions holds for each index i:

- $\alpha_i = \varepsilon$ and $x_i = \varepsilon$
- $\alpha_i \in \Sigma$ and $x_i = \alpha_i$
- $\alpha_i \in N$ and module M_{α_i} accepts x_i.

This model is then easily extended to more general transition labels:

- Recursive *string*-automata allow transitions to be labeled by either strings in Σ^* or module names, and the first two bullets in the previous list become "$\alpha_i = x_i$".
- Recursive *expression*-automata allow transitions to be labeled by either regular expressions over Σ or module names, and the first two bullets in the previous list become "α_i is a regular expression and x_i matches α_i".
- We could even consider *recursive-expression* automata, which allow transitions to be labeled by arbitrary regular expressions over $\Sigma \cup N$.
- We could even even consider recursive-*grammar* automata, which allow transitions to be labeled by arbitrary *context-free grammars* over $\Sigma \cup N$. WEEEE/EEEEW!

For example, the following recursive automaton accepts the language $\{0^m 1^n \mid m \neq n\}$. The recursive automaton has two component modules; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{0^n 1^n \mid n \geq 0\}$. White arrows indicate recursive transitions. The large arrow into each module indicates that module's start state; the large arrow leading out of each module indicates that modules terminal state.



A recursive automaton for the language $\{0^m 1^n \mid m \neq n\}$

Lemma 5.5. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★

Direct construction from the CFG, with one module per nonterminal.

□

For example, the context-free grammar

$$S \to 0A \mid B1$$
$$A \to 0A \mid E$$
$$B \to B1 \mid E$$
$$E \to \varepsilon \mid 0E0$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 5.6. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \to a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \to [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \to [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \to \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$S \to EA \mid 0B \qquad E \to \varepsilon \mid 0X$$
$$A \to 1A \mid 1T \qquad X \to EY$$
$$B \to 0B \mid ET \qquad Y \to 1Z$$
$$T \to \varepsilon \qquad Z \to E$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

5.8 Chomsky Normal Form

For many algorithmic problems involving context-free grammars, it is helpful to consider grammars with a particular special structure called ***Chomsky normal form***, abbreviated ***CNF***:

- The starting non-terminal S does not appear on the right side of any production rule.
- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- The right side of every other production rule is either a single terminal symbol or a string of exactly two non-terminals—that is, every other production rule has the form $A \to BC$ or $A \to a$.

A particularly attractive feature of CNF grammars is that they yield *full binary* parse trees; in particular, every parse tree for a string of length $n > 0$ has exactly $2n - 1$ non-terminal nodes. Consequently, any string of length n in the language of a CNF grammar can be derived in exactly $2n - 1$ production steps. It follows that we can actually determine whether a string belongs to the language of a CNF grammar by brute-force consideration of all possible derivations of the appropriate length.

For arbitrary context-free grammars, there is no similar upper bound on the length of a derivation, and therefore no similar brute-force membership algorithm, because the grammar may contain additional ***ε-productions*** of the form $A \to \varepsilon$ and/or ***unit productions*** of the form $A \to B$, where both A and B are non-terminals. Unit productions introduce nodes of degree 1 into any parse tree, and ε-productions introduce leaves that do not contribute to the word being parsed.

Fortunately, it *is* possible to determine membership in the language of an arbitrary context-free grammar, thanks to the following theorem. Two context-free grammars are ***equivalent*** if they define the same language.

Every context-free grammar is equivalent to a grammar in Chomsky normal form.

Moreover, there are algorithms to automatically convert any context-free grammar into Chomsky normal form. Unfortunately, these conversion algorithms are quite complex, but for most applications of context-free grammars, the details of the conversion are unimportant—it's enough to know that the algorithms exist. For the sake of completeness, however, I will describe one such conversion algorithm in the next section.

*5.9 CNF Conversion Algorithm

I'll actually prove a stronger statement: Not only can we convert any context-free grammar into Chomsky normal form, but we can do so *quickly*. We analyze the running time of our conversion algorithm in terms of the ***total length*** of the input grammar, which is just the number of symbols needed to write down the grammar. Up to constant factors, the total length is the sum of the lengths of the production rules.

Theorem 5.7. *Given an arbitrary context-free grammar with total length L, we can compute an equivalent grammar in Chomsky normal form with total length $O(L^2)$ in $O(L^2)$ time.*

Our algorithm consists of several relatively straightforward stages. Efficient implementation of some of these stages requires standard graph-traversal algorithms, which are described in a different part of the course.

0. Add a new starting non-terminal. Add a new non-terminal S' and a production rule $S' \to S$, where S is the starting non-terminal for the given grammar. S' will be the starting non-terminal for the resulting CNF grammar. (In fact, this step is necessary only when $S \rightsquigarrow^* \varepsilon$, but at this point in the conversion process, we don't yet know whether that's true.)

1. Decompose long production rules. For each production rule $A \to \omega$ whose right side w has length greater than two, add new production rules of length two that still permit the derivation $A \rightsquigarrow^* \omega$. Specifically, suppose $\omega = \alpha\chi$ for some symbol $\alpha \in \Sigma \cup \Gamma$ and string $\chi \in (\Sigma \cup \Gamma)^*$. The algorithm replaces $A \to \omega$ with two new production rules $A \to \alpha B$ and $B \to \chi$, where B is a new non-terminal, and then (if necessary) recursively decomposes the production rule $B \to \chi$. For example, we would replace the long production rule $A \to 0BC1CB$ with the following sequence of short production rules, where each A_i is a new non-terminal:

$$A \to 0A_1 \qquad A_1 \to BA_2 \qquad A_2 \to CA_3 \qquad A_3 \to 1A_4 \qquad A_4 \to CB$$

This stage can significantly increase the number of non-terminals and production rules, but it increases the *total length* of all production rules by at most a small constant factor.[3] Moreover, for the remainder of the conversion algorithm, every production rule has length at most two. The running time of this stage is $O(L)$.

2. Identify nullable non-terminals. A non-terminal A is ***nullable*** if and only if $A \rightsquigarrow^* \varepsilon$. The recursive definition of $\rightsquigarrow^*$ implies that A is nullable if and only if the grammar contains a production rule $A \to \omega$ where ω consists entirely of nullable non-terminals (in particular, if $\omega = \varepsilon$). You may be tempted to transform this recursive characterization directly into a recursive algorithm, but this is a bad idea; the resulting algorithm would fall into an infinite loop if (for example) the same non-terminal appeared on both sides of the same production rule. Instead, we apply the following ***fixed-point*** algorithm, which repeatedly scans through the entire grammar until a complete scan discovers no new nullable non-terminals.

```
NULLABLES(Σ, Γ, R, S):
  Γ_ε ← ∅        ⟨⟨known nullable non-terminals⟩⟩
  done ← FALSE
  while ¬done
      done ← TRUE
      for each non-terminal A ∈ Γ \ Γ_ε
          for each production rule A → ω
              if ω ∈ Γ_ε*
                  add A to Γ_ε
                  done ← FALSE
  return Γ_ε
```

At this point in the conversion algorithm, if S' is ***not*** identified as nullable, then we can safely remove it from the grammar and use the original starting nonterminal S instead.

As written, NULLABLES runs in $O(nL) = O(L^2)$ time, where n is the number of non-terminals in Γ. Each iteration of the main loop except the last adds at least one non-terminal to Γ_ε, so the

[3] In most textbook descriptions of the CFG conversion algorithm, this stage is performed *last*, after removing ε-productions and unit productions. But with the stages in that traditional order, removing ε-productions could *exponentially* increase the length of the grammar in the worst case! Consider the production rule $A \to (BC)^k$, where B is nullable but C is not. Decomposing this rule first and then removing ε-productions introduces about $3k$ new rules; whereas, removing ε-productions first introduces 2^k new rules, most of which then must then be further decomposed!

algorithm halts after at most $n+1 \leq L$ iterations, and in each iteration, we examine at most L production rules. There is a faster implementation of NULLABLES that runs in $O(n+L) = O(L)$ time,[4] but since other parts of the conversion algorithm already require $O(L^2)$ time, we needn't bother.

3. Eliminate ε-productions. First, remove every production rule of the form $A \to \varepsilon$. Then for each production rule $A \to w$, add all possible new production rules of the form $A \to w'$, where w' is a ***non-empty*** string obtained from w by removing one nullable non-terminal. For example, if if the grammar contained the production rule $A \to BC$, where B and C are both nullable, we would add two new production rules $A \to B \mid C$. Finally, if the starting nonterminal S' was identified as nullable in the previous stage, add the production rule $S' \to \varepsilon$; this will be the *only* ε-production in the final grammar. This phase of the conversion runs in $O(L)$ time and at most triples the number of production rules.

4. Merge equivalent non-terminals. We say that two non-terminals A and B are ***equivalent*** if they can be derived from each other: $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. Because we have already removed ε-productions, any such derivation must consist entirely of unit productions. For example, in the grammar

$$S \to B \mid C, \qquad A \to B \mid D \mid CC \mid \mathtt{0}, \qquad B \to C \mid AD \mid \mathtt{1}, \qquad C \to A \mid DA, \qquad D \to BA \mid CS,$$

non-terminals A, B, C are all equivalent, but S is not in that equivalence class (because we cannot derive S from A) and neither is D (because we cannot derive A from D).

Construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Then two non-terminals are equivalent if and only if they are in the same strong component of G. Compute the strong components of G in $O(L)$ time using, for example, the algorithm of Kosaraju and Sharir. Then merge all the non-terminals in each equivalence class into a single non-terminal. Finally, remove any unit productions of the form $A \to A$. The total running time for this phase is $O(L)$. Starting with our example grammar above, merging B and C with A and removing the production $A \to A$ gives us the simpler grammar

$$S \to A, \qquad A \to AA \mid D \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

We could further simplify the grammar by merging all non-terminals reachable from S using only unit productions (in this case, merging non-terminals S and S), but this further simplification is unnecessary.

5. Remove unit productions. Once again, we construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Because no two non-terminals are equivalent, G is acyclic. Thus, using topological sort, we can index the non-terminals $A_1, A_2, \ldots, A_n$ such that for every unit production $A_i \to A_j$ we have $i < j$, again in $O(L)$ time; moreover, we can assume that the starting non-terminal is A_1. (In fact, both the dag G and the linear ordering of non-terminals was already computed in the previous phase!!)

Then for each index j in decreasing order, for each unit production $A_i \to A_j$ and each production $A_j \to \omega$, we add a new production rule $A_i \to \omega$. At this point, all unit productions are

[4] Consider the bipartite graph whose vertices correspond to non-terminals and the right sides of production rules, with one edge per rule. The faster algorithm is a modified breadth-first search of this graph, starting at the vertex representing ε.

redundant and can be removed. Applying this algorithm to our example grammar above gives us the grammar

$$S \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad A \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

In the worst case, each production rule for A_n is copied to each of the other $n-1$ non-terminals. Thus, this phase runs in $\Theta(nL) = O(L^2)$ time and increases the length of the grammar to $\Theta(nL) = O(L^2)$ in the worst case.

This phase dominates the running time of the CNF conversion algorithm. Unlike previous phases, no faster algorithm for removing unit transformations is known! There are grammars of length L with unit productions such that any equivalent grammar without unit productions has length $\Omega(L^{1.499999})$ (for any desired number of 9s), but this lower bound does not rule out the possibility of an algorithm that runs in only $O(L^{3/2})$ time. Closing the gap between $\Omega(L^{3/2-\varepsilon})$ and $O(L^2)$ has been an open problem since the early 1980s!

6. Protect terminals. Finally, for each terminal $a \in \Sigma$, we introduce a new non-terminal A_a and a new production rule $A_a \to a$, and then replace a with A_a in every production rule of length two.

This completes the conversion to Chomsky normal form! As claimed, the total running time of the algorithm is $O(L^2)$, and the total length of the output grammar is also $O(L^2)$.

To see the conversion algorithm in action, let's apply these stages one at a time to our very first example grammar for the language $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$:

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

0. Add a new starting non-terminal S'.

$$\underline{S' \to S} \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

1. Decompose the long production rule $C \to \mathtt{0}C\mathtt{1}$.

$$S' \to S \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad \underline{C \to \varepsilon \mid \mathtt{0}D} \qquad \underline{D \to C\mathtt{1}}$$

2. Identify C as the only nullable non-terminal. Because S' is not nullable, remove the production rule $S' \to S$.

3. Eliminate the ε-production $C \to \varepsilon$.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \mid \underline{\mathtt{0}} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \underline{\mathtt{1}} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \underline{\mathtt{1}}$$

4. No two non-terminals are equivalent, so there's nothing to merge.

5. Remove the unit productions $S' \to S$, $S \to A$, and $S \to B$.

$$S \to \underline{\mathtt{0}A \mid \mathtt{0}C \mid B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{0} \mid \mathtt{1}}$$
$$A \to \mathtt{0}A \mid \mathtt{0}C \mid \mathtt{0} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{1} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \mathtt{1}.$$

6. Finally, protect the terminals $\mathtt{0}$ and $\mathtt{1}$ to obtain the final CNF grammar.

$$S \to \underline{EA \mid EC \mid BF \mid CF \mid \mathtt{0} \mid \mathtt{1}}$$
$$A \to \underline{EA \mid EC} \mid \mathtt{0} \qquad B \to \underline{BF \mid CF} \mid \mathtt{1}$$
$$C \to \underline{ED} \qquad D \to \underline{CF} \mid \mathtt{1}$$
$$\underline{E \to \mathtt{0}} \qquad \underline{F \to \mathtt{1}}$$

Exercises

1. Describe context-free grammars that generate each of the following languages. The function $\#(x, w)$ returns the number of occurrences of the ***substring*** x in the string w. For example, $\#(\texttt{0}, \texttt{101001}) = 3$ and $\#(\texttt{010}, \texttt{1010100011}) = 2$. These are *not* listed in order of increasing difficulty.

   (a) All strings in $\{\texttt{0},\texttt{1}\}^*$ whose length is divisible by 5.

   (b) All strings in $\{\texttt{0},\texttt{1}\}^*$ representing a non-negative multiple of 5 in binary.

   (c) $\{w \in \{\texttt{0},\texttt{1}\}^* \mid \#(\texttt{0}, w) = 2 \cdot \#(\texttt{1}, w)\}$

   (d) $\{w \in \{\texttt{0},\texttt{1}\}^* \mid \#(\texttt{0}, w) \neq 2 \cdot \#(\texttt{1}, w)\}$

   (e) $\{w \in \{\texttt{0},\texttt{1}\}^* \mid \#(\texttt{00}, w) = \#(\texttt{11}, w)\}$

   (f) $\{w \in \{\texttt{0},\texttt{1}\}^* \mid \#(\texttt{01}, w) = \#(\texttt{10}, w)\}$

   (g) $\{w \in \{\texttt{0},\texttt{1}\}^* \mid \#(\texttt{0}, w) = \#(\texttt{1}, w)$ and $|w|$ is a multiple of $3\}$

   (h) $\{\texttt{0},\texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$

   (i) $\{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (j) $\{\texttt{0},\texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (k) $\{\texttt{0}^n\texttt{1}^m \mid 0 \leq 2m \leq n < 3m\}$

   (l) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^{i+j} \mid i, j \geq 0\}$

   (m) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i = j$ or $j = k\}$

   (n) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i \neq j$ or $j \neq k\}$

   (o) $\{\texttt{0}^i\texttt{1}^j\texttt{0}^j\texttt{1}^i \mid i, j \geq 0\}$

   (p) $\left\{w\#\texttt{0}^{\#(\texttt{0},w)} \;\middle|\; w \in \{\texttt{0},\texttt{1}\}^*\right\}$

   (q) $\{xy \mid x, y \in \{\texttt{0},\texttt{1}\}^*$ and $x \neq y$ and $|x| = |y|\}$

   (r) $\left\{x\#y^R \;\middle|\; x, y \in \{\texttt{0},\texttt{1}\}^*$ and $x \neq y\right\}$

   (s) $\{x\#y \mid x, y \in \{\texttt{0},\texttt{1}\}^*$ and $\#(\texttt{0}, x) = \#(\texttt{1}, y)\}$

   (t) $\{\texttt{0},\texttt{1}\}^* \setminus \{ww \mid w \in \{\texttt{0},\texttt{1}\}^*\}$

   (u) All strings in $\{\texttt{0},\texttt{1}\}^*$ that are *not* palindromes.

   (v) All strings in $\{\texttt{(},\texttt{)},\diamond\}^*$ in which the parentheses are balanced and the symbol $\diamond$ appears at most four times. For example, $\texttt{()(())}$ and $\texttt{(}\diamond\diamond\texttt{(()()}\diamond\texttt{)()())}\diamond$ and $\diamond\diamond\diamond$ are strings in this language, but $\texttt{)(()}$ and $\texttt{(}\diamond\diamond\diamond\texttt{)}\diamond\diamond$ are not.

2. Describe recursive automata for each of the languages in problem 1. ("Describe" does not necessarily mean "draw"!)

3. Prove that if L is a context-free language, then L^R is also a context-free language. *[Hint: How do you reverse a context-free grammar?]*

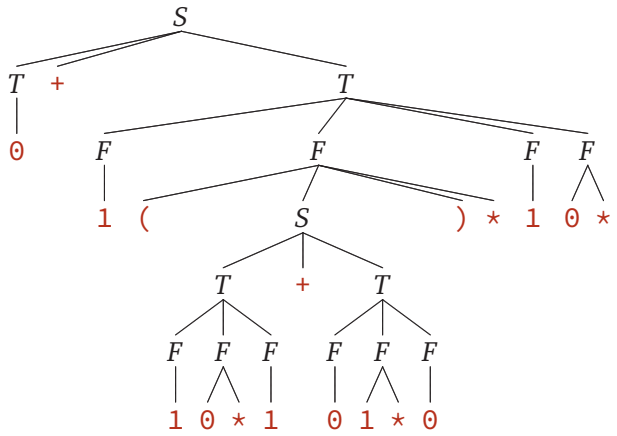
4. Consider a generalization of context-free grammars that allows any *regular expression* over $\Sigma \cup \Gamma$ to appear on the right side of a production rule. Without loss of generality, for each non-terminal $A \in \Gamma$, the generalized grammar contains a single regular expression $R(A)$. To

apply a production rule to a string, we replace any non-terminal A with an arbitrary word in the language described by $R(A)$. As usual, the language of the generalized grammar is the set of all strings that can be derived from its start non-terminal.

For example:, the following generalized context-free grammar describes the language of all regular expressions over the alphabet $\{0, 1\}$:

$S \to (T+)^* T + \emptyset$	(Regular expressions)
$T \to \varepsilon + F^* F$	(Terms = summable expressions)
$F \to (0 + 1 + (S))(* + \varepsilon)$	(Factors = concatenable expressions)

Here is a parse tree for the regular expression `0+1(10*1+01*0)*10*` (which represents the set of all binary numbers divisible by 3):



Prove that every *generalized* context-free grammar describes a context-free language. In other words, show that allowing regular expressions to appear in production rules does not increase the expressive power of context-free grammars.