(1) (24 pts.) Say that a context-free grammar $G = (V, \Sigma, \mathcal{R}, S)$ is in “short form” if for each $A \in V$, the rules for the variable $A$ collectively have no more than two occurrences of variables total on all their right-hand sides. This form allows $A \rightarrow \epsilon$ freely and allows unit rules $A \rightarrow B$ (but at most two of them for each $A$) unlike Chomsky normal form. For argument’s sake we’ll fix $\Sigma = \{0, 1\}$ for this problem.

Prove that for every regular language $L$ there is a CFG $G$ in short form such that $L = L(G)$. Prove this by structural induction on the following grammar $G_{\text{REG}}$ that generates regular expressions $R$:

$$E \rightarrow \emptyset \mid \epsilon \mid 0 \mid 1 \mid (E \cup E) \mid (E \cdot E) \mid (E^*)$$

Put another way, describe and verify an algorithm to convert any regular expression $R$ into a short-form CFG $G$ such that $L(G) = L(R)$, so that the algorithm works by recursion on sub-expressions. That way your answer should follow the same structure as the conversion from regexps to equivalent NFAs shown in class.

**Answer:** Define $P_E = \text{“Every regexp $R$ I derive has a CFG $G$ in short form such that } L(G) = L(R).\text{”} \text{ The rule } E \rightarrow \epsilon \text{ is immediately satisfied by taking } G \text{ to have the production } S \rightarrow \epsilon \text{ and nothing else. The rule } E \rightarrow \emptyset \text{ is similarly done with } G \text{ having no rules at all, or the useless rule } S \rightarrow S, \text{ say. The other base rules } E \rightarrow 0 \text{ and } E \rightarrow 1 \text{ are handled by having the algorithm output } G \text{ with just the rule } S \rightarrow 0 \text{ or } S \rightarrow 1, \text{ respectively. The four basic } G\text{’s obey short form and so uphold } P_E. \text{ Now for the other three rules:}

- $E \rightarrow (E \cup E)$: Suppose $E \Rightarrow^* R$ using this rule first. Then $R$ parses as $R_1 \cup R_2$ where $E \Rightarrow^* R_1$ and $E \Rightarrow^* R_2$. By IH $P_E$ on RHS (twice), our algorithm already outputs grammars $G_1 = (V_1, \Sigma, \mathcal{R}_1, S_1)$ and $G_2 = (V_2, \Sigma, \mathcal{R}_2, S_2)$ in short form such that $L(G_1) = L(R_1)$ and $L(G_2) = L(R_2)$. Define $G_3 = (V_3, \Sigma, \mathcal{R}_3, S_3)$ by lumping together the variables and rules of $G_1$ and $G_2$ plus the new start variable $S_3$ with the new rules $S_3 \rightarrow S_1 \mid S_2$.

Then $G_3$ still has at most two right-hand-side variables per rule and $L(G_3) = L(G_1) \cup L(G_2)$ (by construction) = $L(R_1) \cup L(R_2)$ (by induction) = $L(R)$. So $L(G_3) = L(R)$ with $G_3$ in short form, which upholds $P_E$ on LHS.

- $E \rightarrow (E \cdot E)$: Suppose $E \Rightarrow^* R$ utrf. Then $R =: R_1 \cdot R_2$ where $E \Rightarrow^* R_1$ and $E \Rightarrow^* R_2$. By IH $P_E$ on RHS (twice), our algorithm already outputs grammars $G_1 = (V_1, \Sigma, \mathcal{R}_1, S_1)$ and $G_2 = (V_2, \Sigma, \mathcal{R}_2, S_2)$ in short form such that $L(G_1) = L(R_1)$ and $L(G_2) = L(R_2)$. Define $G_4$ this time with the new start variable $S_4$ and single rule $S_4 \rightarrow S_1S_2$.

plus the other variables and rules. Then $G_4$ still obeys short form and we have:

$$L(G_4) = L(G_1) \cdot L(G_2) \quad \text{(by construction)}$$
$$= L(R_1) \cup L(R_2) \quad \text{(by induction hypothesis)}$$
$$= L(R).$$
Thus $P_E$ is upheld on LHS.

- $E \rightarrow (E^*)$: Suppose $E \Longrightarrow R$ utrf. Then $R =: R_1^*$ where $E \Longrightarrow R_1$. By IH $P_E$ on RHS, we have a short form grammar $G_1 = (V_1, \Sigma, R_1, S_1)$ such that $L(G_1) = L(R_1)$. Define $G_5 = (V_5, \Sigma, R_5, S_5)$ by $V_5 = V_1 \cup S_5$ (here it is understood that $S_5$ is a new symbol, i.e., $S_5 \notin V_1$) and

$$R_5 = R_1 \cup \{S_5 \rightarrow S_5 S_1 \mid \epsilon\}.$$ 

Then $G_5$ is still in short form since the new rules still have only two occurrences of variables total—the $\epsilon$ not counting.

By virtue of proving $P_E$ in this manner, we have simultaneously pseudocoded the entire algorithm for the conversion and proved that the algorithm is correct.

(2) (21 pts. total) Consider the following context-free grammar $G$:

$$
\begin{align*}
S & \rightarrow AC \mid DC \\
A & \rightarrow aS \mid BA \\
B & \rightarrow \epsilon \mid SCS \\
C & \rightarrow BD \mid AS \\
D & \rightarrow BB \mid b
\end{align*}
$$

(a) Find the whole set $N$ of nullable variables. Then carry out the step that adds rules skipping any subset of occurrences of nullable variables to get a new grammar $G_1$. Note that if $S$ is nullable then you get $L(G_1) = L(G) \setminus \{\epsilon\}$, else you get $L(G_1) = L(G)$. (Do not do the text’s initial step of adding a new start variable $S_0$. 6 + 6 = 12 pts.)

(b) Your grammar $G_1$ will have several unit rules—but don’t include “self-rules” like $A \rightarrow A$. Draw a directed graph whose nodes are the five variables and which has an edge $(A, B)$ if $A \rightarrow B$ is a unit rule. Then take the transitive closure of the graph, which will tell you all pairs $(A, B)$ such that $A \Longrightarrow^* B$. Here we still ignore self-loops; that is, we only consider $B \neq A$. (6 + 6 = 12 pts.)

(c) Show the grammar $G_2$ that you get upon making all right-hand sides of rules for $B$ become right-hand sides of rules for $A$ whenever $A \Longrightarrow^* B$, then finally deleting all the unit rules. (6 pts.)

(d) Convert $G_2$ all the way into a grammar $G_3$ in Chomsky normal form, such that $L(G_3) = L(G)$. (−9 pts.; sorry for the answer being unspeakably ugly compared to the original $G$, really like the text’s example in that regard.)

Answer: (a) It is easy to see by inspection that the nullable variables are first $B$, then $D$, then $C$, and finally $S$. The variable $A$ is not nullable because the rule $A \rightarrow BA$ perpetuates it
until you finally do $A \rightarrow aS$ which derives the terminal $a$. Then deleting subsets of occurrences of $S, B, C, D$ gives $G_1 =$

\[
S \rightarrow AC | DC | A | D | C \quad \text{(the $C$ came from $DC$ not $AC$)} \\
A \rightarrow aS | BA | a \quad \text{(ignore $A \rightarrow A$)} \\
B \rightarrow SCS | CS | SS | SC | S | C \\
C \rightarrow BD | AS | B | D | A \quad \text{(but not $S$ since $A$ isn’t nullable)} \\
D \rightarrow BB | b | B
\]

(b,c) The graph has edges from $S$ into $A, C, D$, from $D$ into $B$, from $B$ into $S$ which completes a cycle (and $C$), and from $C$ into $A, B, D$ which completes other cycles. Nothing out of $A$, however, only into $A$. The transitive closure makes $S$ go to $B$ as well and adds $S$ as an option for $C$, which further makes $S, B, C, D$ all go to each other and all go to $A$. So the variables except $A$ all lump together all of their non-unit options, which gives the answer for (c):

\[
S \rightarrow AC | DC | aS | BA | a | SCS | CS | SS | SC | BD | AS | BB | b \\
A \rightarrow aS | BA | a \\
B \rightarrow \text{(same as $S$)} \\
C \rightarrow \text{(same as $S$)} \\
D \rightarrow \text{(same as $S$)}
\]

(d) Alias $X_a \rightarrow a$ and use one long-rule variable $Y$ to make $S \rightarrow SY, Y \rightarrow CS$ in place of the lone long rule $S \rightarrow SCS$. Finally, since $S$ was originally nullable, add a new start variable $S_0$ with rules $S_0 \rightarrow \epsilon$ | (same as $S$). Yuck! We could simplify a little by noting that since $B, C, D$ really are equivalent to $S$ we can just substitute $S$ for them and discard duplicate right-hand sides. So the grammar after step (c) really “condenses” to

\[
S \rightarrow AS | SS | aS | SA | a | b | SSS \\
A \rightarrow aS | SA | a
\]

So the final Chomsky NF grammar is not too painful but still ugly:

\[
S_0 \rightarrow \epsilon | AS | SS | X_aS | SA | a | b | SY \\
S \rightarrow AS | SS | X_aS | SA | a | SY \\
A \rightarrow X_aS | SA | a \\
X_a \rightarrow a \\
Y \rightarrow SS.
\]

[Well, by this point it is abundantly clear that $L(G) = (a+b)^*$ so we could have just used $S_0 \rightarrow \epsilon | XS | a | b, S \rightarrow XS | a | b, X \rightarrow a | b.$]
(3) (18 pts.) Let $\Sigma = \{a, b\}$, and let $L$ be the language of palindromes over $\Sigma$ that have twice as many $a$’s as $b$’s. That is,

$$L = \{x \in \Sigma^* : x = x^R \land \#a(x) = 2 \cdot \#b(x)\}.$$ 

Prove via the CFL Pumping Lemma that $L$ is not a context-free language. (Hint: Try $x$ of the form $a^p b^p a^p$. 18 pts., making the whole problem set “out of” 63 pts.)

Answer: Let any $p > 0$ be given. Take $s = a^p b^p a^p$. Let any breakdown $s = uvxyz$ with $|vxy| \leq p$, $vy \neq \epsilon$, be given. Then $s^{(0)} = uv^0xy^0z = uxz$ does not belong to $L$, as shown by the following case analysis:

(i) If $vy$ includes no $a$’s then it must include at least one $b$’s, say $r b$’s. Then $s^{(0)} = a^p b^{p-r} a^p$ is still a palindrome but it has more than twice as many $a$’s as $b$’s so it fails the second condition used to define $L$.

(ii) If $vy$ includes some $a$’s then it touches only the left-hand $a^p$ or the right-hand $a^p$, not both since $|vxy| \leq p$. Thus in $s^{(0)}$ the left-hand and right-hand sides no longer balance, and since at least one $b$ remains between them, $s^{(0)}$ is not a palindrome.

Thus in both of these mutually exhaustive cases we get $s^{(0)} \notin L$, so $L$ is not a CFL by the CFL Pumping Lemma.

Footnotes: It is in fact true that every CFG can be converted into short form, not just those arising inductively from the regular expressions. This idea is more naturally iterative than recursive and is (IMHO) frankly “grungy”: Given any variable $A$ with $r$-many rules $A \rightarrow X_i$ defined for it, make a separate variable $A_i$ with the single rule $A_i \rightarrow X_i$ instead. Then arrange $A_1, \ldots, A_r$ as the leaves of a binary tree with $A$ at the root. Allocate new variables $A'_j$ to the tree’s other internal nodes besides the root, and if $A'_j$ has children $A'_k$ and $A_i$ (say), give $A'_j$ the unit rules $A'_j \rightarrow A'_k \mid A_i$. Not only the new variables but also the original variable $A$ now obey short form, except that an original right-hand side $X_i$ may have $|X_i| = q > 2$. No matter—just add $q - 2$ “Chomsky long-rule variables” $Y_1, \ldots, Y_{q-2}$ and use them to break down $X_i$ the same way as for the final step of Chomsky normal form. (I considered making this extra credit but feared mentioning this would generate greater confusion—the cherry on my April Fool’s joke is that the real extra credit would have involved similar final Chomsky steps to the “ersatz” extra credit.)

This fact renders “short form” undistinguished as a concept, and it was made up to constrain the algorithm and provide a different context from the text’s exercises 2.16 and 2.17 (which follow from the same meat of the argument, $S_3 \rightarrow S_1 \mid S_2$, $S_4 \rightarrow S_1 S_2$, and $S_5 \rightarrow S_1 S_5 \mid \epsilon$). The grammar concepts that are distinctive to regular languages are called left-linear and right-linear. These are defined by allowing every rule $A \rightarrow X$ to have at most one variable in $X$, and either all such cases have $X$ begin with that variable (left-linear) or end with it (right-linear). Unlike “short form” there is no limit on the total number of right-hand sides for $A$ that have a variable in them. Sometimes I’ve given for homework the task of converting a (G)NFA into such grammars and vice-versa, sometimes with the further restriction $|X| \leq 2$ which defines regular grammars which form the third (bottom) level of the so-called “Chomsky Hierarchy.”
The exact algorithm for finding all nullable variables is a general kind worth noting. Here is some C-like code for it:

```
set<Variable> NULLABLE = \emptyset;
bool itGrew = true;
while (itGrew) {
    itGrew = false;
    for (each rule A --> X) {
        if (A \in NULLABLE) { continue; }
        if (X \in NULLABLE*) {
            NULLABLE += \{A\};
            itGrew = true;
        }
    }
}
return NULLABLE;
```

You might expect me to have initialized NULLABLE to be all variables $A$ with $A \to \epsilon$ as a given rule. But actually they all get added in the first for-each iteration through all the rules, thanks to the fact that $\emptyset^* = \epsilon$. That’s a nifty coding convenience.

The thing to note is that this algorithm is doubly iterative—it has a series of passes thru each rule, but may make as many as $k + 1$ passes total where $k = |V|$. It can’t make any more passes because each pass adds a variable to NULLABLE, else the whole algorithm halts because there was no change on that pass. So this algorithm runs in time roughly order-of $|V| \cdot |R|$, which is at worst quadratic time in the overall symbol-size of the grammar. In the last week of term this will become part of saying that the problem of whether a CFG derives $\epsilon$ (that is, whether $S$ is nullable) belongs to the class $P$ of problems that are solvable in polynomial time.

We can vary the algorithm by initializing NULLABLE to be $\Sigma$ in place of $\emptyset$. Then the first pass catches all variables that have at least one purely terminal rule. The whole algorithm then finds all variables that are capable of deriving themselves to terminals at all. Any leftover variables are called deadwood, and they and their rules can be safely deleted from the grammar without changing the language. This sometimes happens during the conversion to Chomsky normal form. Another thing that can happen is that some variables become unreachable from the start variable. That is roughly similar to what we’ve already been doing by economizing the breadth-first search of the NFA-to-DFA conversion (and sometimes the Cartesian product construction too) and come to think of it that algorithm likewise has an iterative form with an “itGrew?” stopping feature.