Note: In any grammar here, the meaning and usage of P (productions) is equivalent to R (rules).

1a) $G = (\{R, S, T\}, \{0,1\}, P, S)$ where $P$ is:

$$S \rightarrow R0R$$
$$R \rightarrow R0R1R \mid R1R0R \mid T$$
$$T \rightarrow 0T \mid \varepsilon$$

(S generates the first 0. R generates pairs of 0s and 1s. T generates 0s or wipes out Rs. Since T can wipe out Rs, the first rule creates a 0 that can be at the beginning, end, or somewhere in the middle of the string. T guarantees that we can have as many 0s as we want, but S guarantees that we have at least one more 0 than 1.)

1b) Begin with the CFG grammar, $G = (\{S, S_1, S_2, S_3, R_1, R_2, R_3, A, B, C\}, \{a, b, c\}, P, S)$ where $P$ is:

(S produces the union of the three cases)

$$S \rightarrow S_1 \mid S_2 \mid S_3$$

(These two rules produce strings with at least 2 symbols “out of order”)

$$S_1 \rightarrow R_1baR_1 \mid R_1caR_1 \mid R_1cbR_1$$
$$R_1 \rightarrow aR_1 \mid bR_1 \mid cR_1 \mid \varepsilon$$

(These two rules produce strings of form $a^ib^jc$ with $i \neq j$)

$$S_2 \rightarrow R_2C \mid R_2$$
$$R_2 \rightarrow aR_1b \mid A \mid B$$

(These two rules produce strings of form $a^ib^jc$ with $j \neq k$)

$$S_3 \rightarrow AR_3 \mid R_3$$
$$R_3 \rightarrow bR_3c \mid B \mid C$$

(These three rules produce a string of characters of the same symbol)

$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$
$$C \rightarrow cC \mid c$$

Convert to CNF

0. We skip creating a new start variable since the existing start variable, S, is non-recursive.

1. Remove $\varepsilon$-productions. $R_1$ is the only nullable variable.

$$S \rightarrow S_1 \mid S_2 \mid S_3$$
$$S_1 \rightarrow R_1baR_1 \mid R_1caR_1 \mid R_1cbR_1 \mid baR_1 \mid caR_1 \mid cbR_1 \mid R_1ba \mid R_1ca \mid R_1cb \mid ba \mid ca \mid cb$$
$$R_1 \rightarrow aR_1 \mid bR_1 \mid cR_1 \mid a \mid b \mid c$$
$$S_2 \rightarrow R_2C \mid R_2$$
$$R_2 \rightarrow aR_1b \mid A \mid B$$
$$S_3 \rightarrow AR_3 \mid R_3$$
$$R_3 \rightarrow bR_3c \mid B \mid C$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$
$$C \rightarrow cC \mid c$$

2. Remove unit productions.

$$S \rightarrow R_1baR_1 \mid R_1caR_1 \mid R_1cbR_1 \mid baR_1 \mid caR_1 \mid cbR_1 \mid R_1ba \mid R_1ca \mid R_1cb \mid ba \mid ca \mid cb \mid R_3C \mid aR_2b \mid aA \mid a \mid bB \mid b \mid AR_3 \mid bR_3c \mid cC \mid c$$
CSE355 Spring 2013 – Homework 3 Sample Solutions

\[
G = (\{S\}, \{0, 1\}, P, S) \text{ where } P \text{ is:}
\]

\[
S \rightarrow R_1 b a R_1 | R_1 c a R_1 | R_1 c b R_1 | b a R_1 | c a R_1 | c b R_1 | R_1 b a | R_1 c a | R_1 c b | b a | c a | c b
\]

\[
R_1 \rightarrow a R_1 | b R_1 | c R_1 | a | b | c
\]

\[
S_2 \rightarrow R_2 C | a R_2 b | a A | a | b B | b
\]

\[
R_2 \rightarrow a R_2 b | a A | a | b B | b
\]

\[
S_3 \rightarrow A R_3 | b R_3 c | b B | b | c C | c
\]

\[
R_3 \rightarrow b R_3 c | b B | b | c C | c
\]

\[
A \rightarrow a A | a
\]

\[
B \rightarrow b B | b
\]

\[
C \rightarrow c C | c
\]

3. Remove useless (nonproductive, unreachable) productions.

\[
S \Rightarrow R_1 b a R_1 | R_1 c a R_1 | R_1 c b R_1 | b a R_1 | c a R_1 | c b R_1 | R_1 b a | R_1 c a | R_1 c b | b a | c a | c b | R_2 C | a R_2 b | a A | a | b B | b | A R_3 | b R_3 c | c C | c
\]

\[
R_1 \rightarrow a R_1 | b R_1 | c R_1 | a | b | c
\]

\[
R_2 \rightarrow a R_2 b | a A | a | b B | b
\]

\[
R_3 \rightarrow b R_3 c | b B | b | c C | c
\]

\[
A \rightarrow a A | a
\]

\[
B \rightarrow b B | b
\]

\[
C \rightarrow c C | c
\]

4. Add variables to make RHS have exactly two variables or exactly one terminal.

\[
S \Rightarrow R_3 V_1 | R_3 V_3 | R_3 V_4 | V_3 V_2 | V_3 V_5 | R_1 V_6 | R_1 V_7 | R_1 V_8 | V_6 V_2 | V_6 V_5 | V_7 V_3 | V_8 V_5 | V_9 C | V_{10} C | V_a V_2 | V_a V_4 | V_a V_5 | V_a V_6 | V_a V_7 | V_a V_9 | V_a V_{10} | V_b V_1 | V_b V_2 | V_b V_5 | V_b V_6 | V_b V_7 | V_b V_9 | V_b V_{10} | V_c A | V_c B | V_c C
\]

\[
R_1 \rightarrow V_a R_1 | V_b R_1 | V_c R_1 | a | b | c
\]

\[
R_2 \rightarrow V_b V_9 | V_a A | a | V_b B | b
\]

\[
R_3 \rightarrow V_b V_{10} | V_b B | b | V_c C | c
\]

\[
A \rightarrow V_a A | a
\]

\[
B \rightarrow V_b B | b
\]

\[
C \rightarrow V_c C | c
\]

\[
V_a \rightarrow a
\]

\[
V_b \rightarrow b
\]

\[
V_c \rightarrow c
\]

\[
V_1 \rightarrow V_b V_2
\]

\[
V_2 \rightarrow V_3 R_1
\]

\[
V_3 \rightarrow V_c V_2
\]

\[
V_4 \rightarrow V_c V_5
\]

\[
V_5 \rightarrow V_b R_1
\]

\[
V_6 \rightarrow V_b V_3
\]

\[
V_7 \rightarrow V_c V_4
\]

\[
V_8 \rightarrow V_c V_b
\]

\[
V_9 \rightarrow R_2 V_b
\]

\[
V_{10} \rightarrow R_2 V_c
\]

1e) \[G = (\{S\}, \{0, 1\}, P, S) \text{ where } P \text{ is:}
\]

\[
S \Rightarrow 0 S 0 | 1 S 1 | 0 | 1 | \varepsilon
\]

Page 2/8
(The first two productions make matching characters on either side of the string, working from the outside in. The next two productions allow us to wipe out the last S with either a 0 or a 1; this places that character in the center of the string and yields an odd length string. Alternately, the last production allows us to wipe out the last S with an ε, resulting in an even length string.)

1d) \( G = (\{S, S'\}, \{a, b\}, P, S) \) where \( P \) is:
\[
S \rightarrow aSb | S' \\
S' \rightarrow bS'a | \varepsilon
\]
(S' generates the n as and bs on the outside. S' generates the m as and bs on the inside.)

1e) If \( L \) is a finite subset of \( \{a, b\}^* \), we can make a CFG, \( G = (\{S, W_1, W_2, ..., W_n\}, \{a, b\}, P, S) \), for \( L \). Let \( n \) be the number of strings in \( L \) (\( n \) is a fixed finite number since \( L \) is a finite subset). For each string of terminals, \( w_i \), in \( L \), add to \( P \) a production of the form \( W_i \rightarrow w_i \), and a production \( S \rightarrow W_i \). We do this \( n \) times, and then we have a grammar that produces all strings in \( L \). This grammar is context-free because there is a single non-terminal on the left-hand side of every production. (Since all productions are either of the form \( A \rightarrow Bx \) or \( A \rightarrow x \) where \( A, B \) are non-terminals and \( x \in \{a, b\}^* \), all productions are left-linear, and therefore this grammar is also regular. We know that all regular grammars are CFGs, although it is not the case that all CFGs are also regular grammars.)

2a) Informal description: match 1s to 0s in a way that ensures if you have symbols on the stack, they are of the same type (all either 0s or 1s) so the number of symbols on the stack is the “surplus” of that symbol, or how many more of that symbol we’ve seen than the other. Therefore, once we’ve read all of the input, accept iff there is a 0 on top of the stack.

2b) Informal description: we have three pieces to deal with. The first piece reads \( \{a, b, c\}^* \), followed by either ba, ca, or cb, followed by \( \{a, b, c\}^* \). It does not use its stack. The second piece reads and pushes as, followed by popping one a for every b read. If the number of bs exceeds as, it pushes bs. Then it reads any number of cs. It accepts iff there is either an a or at least one b on the top of the stack at the end of the input. The third piece is similar to the second piece, except it checks bs against cs. It reads but not not track as. Then it pushes bs to the stack as it reads bs. Last, it pops one b for every c
read. If the number of cs exceeds the number of bs, it pushes cs. It accepts iff there is at least one b or c on top of the stack after it has read all of the input. The three pieces are joined to the start state via ϵ-transitions.

2c) Informal description: since a palindrome over alphabet \{0, 1\} basically has the form \(x\{0,1, \epsilon\}x^R\), we want a portion of the PDA that pushes \(x\) to the stack, a portion to guess the middle of the string and whether the string has even length, and therefore ϵ-transition to the next portion, or odd length and transition while “throwing away” either a 0 or a 1, and then a portion to pop \(x^R\) from the stack while
matching it against $x$. Accept iff we read all of the input and end with an empty stack, which means $x$ matched exactly to $x^R$.

2d) Informal description: read and push the first $n$ as. Read and push the $m$ bs. Read the $m$ as and pop one b for every a read. Read the $n$ bs and pop an a for every b read. Accept iff all of the input is read and we finish with an empty stack.

2e) Since we don't know what $L$ is, we can't produce a PDA, but we can describe how we would do so. Informal description: we're going to build a PDA that mimics the grammar. For every string in $L$, we will build a PDA that reads the string and ends in a final state. Then, create a new start state and $\varepsilon$-transition to the old start state of every PDA created for a string.

Constructive description: Let $P = (Q, \{a, b\}, \varepsilon, \delta, q_0, F)$. For every $w_i$ in $L$, add $q_{i0}$ to $Q$ and add $\delta(q_{i0}, \varepsilon) = (q_{i0}, \varepsilon)$. Then for every $x_j$ with $w_i = x_1 x_2 \ldots x_n$, add $q_{ji}$ to $Q$ and add $\delta(q_{ji-1}, x_j) = (q_{ji}, \varepsilon)$. Add $q_m$ to $F$. Since every string is of finite length, and as $L$ is a finite subset of $\{a, b\}^*$, we have a finite number of strings, we will have a finite number of states and a finite number of transitions to add, so this procedure will terminate. Note that this PDA does not use its stack.

3) First, to show that $R$ is a CFL, let's produce a CFG, $G=\{S, T\}, \{a, b, c, (, ), \cup, \circ, *, \varepsilon, \emptyset\}, P, S)$, for $R$, where $P$ is:

$$S \rightarrow (S) \mid S \cup S \mid S^* \mid a \mid b \mid c \mid \varepsilon \mid \emptyset$$

(You need to provide some kind of justification that $L(G) = R$, but not necessarily using an inductive proof.) To show that $L(G) = R$, we can refer to the inductive definition of regular expressions to see that everything this grammar generates is a regular expression, and that it generates all regular expressions using alphabet $\{a, b, c\}$. First, let's show that $R \subseteq L(G)$.

- Base case: Choose $r \in R$ with $|r| = 1$. Then, apply one of the rules $S \rightarrow a \mid b \mid c \mid \varepsilon \mid \emptyset$ to produce a derivation of $r$ in $L(G)$. Then, if $r \in R$, $|r| = 1$, $r \in L(G)$.
- Inductive Hypothesis: Assume it is the case that if $r \in R$, $|r| < k$, $k > 1$, there is a derivation of $r$ using productions in $G$, so $r \in L(G)$. 

Page 5/8
Inductive Step: If \(|r| = k, k> 1\), it must be the case that \(r\) is of the form \((R_1), R_1\cup R_2, R_1\cap R_2,\) or \(R,\) \(\ast\). Using one of the productions \(S \rightarrow (S)|S\cup S|S-S|S^\ast\) we can generate a string in any one of these forms. In any of these cases, \(|R_1|, |R_2| < k\), so we know that \(R_1, R_2 \in R\) implies \(R_1, R_2 \in L(G)\) by the inductive hypothesis, and so there is a derivation of the grammar that can produce \(R_1\) and \(R_2\). Then, there is a derivation of the grammar that can produce \(r\), so \(r \in L(G)\).

This shows that \(R \subseteq L(G)\). Now, let’s show that \(L(G) \subseteq R\).

• Base case: Choose \(r \in L(G)\) with number of steps in a derivation of \(r, n = 1\). Then the only productions that could have been applied to produce a string of terminals in one step are from the set \(S \rightarrow a | b | c | \varepsilon | \emptyset\). Then, \(r\) is an element of the set \{a, b, c, \varepsilon, \emptyset\}. These are the primitive regular expressions, and so \(r \in R\).

• Inductive Hypothesis: Assume it is the case that if \(r \in L(G), n<k, k>1\), then \(r \in R\).

• Inductive Step: Choose \(r \in L(G)\) with number of steps in a derivation of \(r, n = k, k>1\). Since \(n>1\), one of the productions from the set \(S \rightarrow (S)|S\cup S|S-S|S^\ast\) must have been applied. Then, \(r\) is an element from the set \{(\(R_i\), \(R_1\cup R_2, R_1\cap R_2, R^\ast\)\}. Then, for any of these elements, there is a derivation of \(R_1, R_2\) in less than \(k\) steps, so, \(R_1, R_2 \in R\). Since all of these elements are regular expressions if their components are regular expressions, and \(R_1, R_2 \in R\), then \(r \in R\).

This shows that \(R \subseteq L(G)\). Since \(R \subseteq L(G)\) and \(L(G) \subseteq R\), then \(R = L(G)\). Since \(G\) is a CFG, \(L(G) = R\), then \(R\) is a CFL.

To show that \(R\) is not regular, we must show that \(R\) cannot be pumped by applying pumping for regular languages. Let \(\ell\) be the pumping constant. Choose \(w = (a^\ell)^p\). We can produce this string by applying the production \(S \rightarrow (S)\) \(p\) times followed by \(S \rightarrow a\), so \(w \in R\). Any decomposition of \(w = xyz\) with \(|xy| \leq \ell\) and \(|y| > 0\) is such that \(y\) is composed of one or more \(a\) symbols. Let \(|y| = k\). Then for \(i = 0\), the pumped string is \(w_i = (a^{\ell+k})^p\). Since \(k > 0\), \(p-k < \ell\). Therefore, \(w_i\) could not have been produced by a derivation of \(G\), and is therefore not in \(R\), and therefore, \(R\) is not regular.

4a) \(L = \{a^n b^n a^n b^n : n,m \geq 0\}\)

Let \(\ell\) be the pumping constant. Choose \(w = a^\ell b^{2\ell} a^\ell b^{2\ell}\). \(w \in L, |w| > \ell\). 

Case 1: \(|v| = 0, |y| = k, k>0\), and \(y\) consists of just one symbol. Let \(i = 0\). If \(y\) is as, then \(p-k < \ell\), and if \(y\) is bs, then \(2p-k < 2\ell\), so \(w_0\) is not in the language. The cases where \(|y|=0, |v|=j, j>0\), and \(v\) consists of just one symbol is symmetric.

Case 2: \(|v| = 0, |y| = k, k>0\), and \(y\) consists of two symbols. Let \(i = 2\). If \(y\) is n as followed by m bs, then \(w_2 = a^n b^m a^n b^m a^n b^m\), and \(w_2\) is not in the language. The cases where \(y\) occurs on the other side of the string, is composed of bs followed by as, or is empty and \(v\) is nonempty and meets these conditions are similar enough to state, but not necessary to show.

Case 3: \(|v| = j, |y| = k, j, k > 0\). Since \(j+k \leq \ell\), either \(vxy\) consists of only one symbol, or at most two symbols. If \(vxy\) consists of only one symbol, this is similar enough to case 1 to state, but not necessary to show. If \(vxy\) consists of two symbols with either \(v\) or \(y\) consisting of two symbols, this is similar enough to case 2. If \(vxy\) consists of two symbols with \(v\) consisting of all as and \(y\) consisting of all bs, then if \(i = 0\), the pumped string \(w_0\) is either \(a^n b^{m+j-k} a^n b^m\) or \(a^n b^m a^n b^{m+k}\). Likewise, if \(v\) is all bs and \(y\) is all as, then \(w_0 = a^n b^m a^n b^{m-k}\). In all of these cases, \(w_0\) is not in the language.

For all decompositions of \(w\), there exists an \(i\) for which \(uv^ixy^i z\) is not in the language, this language is not context-free.
4b) \( L = \{a^ib^jc^k \mid i \geq j \geq k \geq 0 \} \)

Let \( p \) be the pumping constant. Choose \( w = a^p b^p c^p \). \( p \geq p > 0 \) so \( w \in L \) and \( |w| > p \).

Case 1: \(|v| = 0, |y| = k, k > 0\) and \( y \) contains only one symbol. If \( y \) contains as, let \( i = 0 \). Then \( w_0 = a^p b^p c^p \). Then \( n_k < n_k \) so \( w_0 \notin L \). If \( y \) contains bs, let \( i = 2 \). Then \( w_2 = a^p b^p c^p \). Then \( n_k < n_k \) so \( w_2 \notin L \). The case where \( y \) is empty and \( v \) is nonempty and contains one symbol is symmetric. This also covers the case where both \( v \) and \( y \) are nonempty and both contain the same symbol.

Case 2: \(|v| = 0, |y| = k, k > 0,\) and \( y \) contains two symbols. If \( y \) contains as followed by bs, then let \( i = 2 \). Then \( w_2 = a^p b^a c^p \) and \( w_2 \notin L \). If \( y \) contains bs followed by as, then let \( i = 2 \). Then \( w_2 = a^p b^a c^p \) and \( w_2 \notin L \). The case where \( y \) is empty and \( v \) is nonempty and contains two symbols is symmetric. This also covers the case where both \( v \) and \( y \) are nonempty, but one of \( v \) or \( y \) contains two symbols.

Case 3: \(|v| = j, |y| = k, j, k > 0,\) both \( v \) and \( y \) contain different symbols. If \( v \) contains as and \( y \) contains bs, let \( i = 0 \). Then \( w_0 = a^j a^m b^p c^p \). Then \( n_k > n_k \) so \( w_0 \notin L \). If \( v \) contains bs and \( y \) contains cs, let \( i = 2 \). Then \( w_2 = a^p b^a c^p \) and \( w_2 \notin L \). This also covers the case where both \( v \) and \( y \) are nonempty and both contain the same symbol (both \( v \) and \( y \) are all as, or \( v \) and \( y \) are all bs).

For any decomposition of \( w \), there exists an \( i \) such that \( w_i \notin L \), so \( w \) cannot be pumped using pumping for context-free languages, so \( L \) is not context-free.

4c) \( L = \{ \text{palindromes in} \ \{a, b\} \} \) that have the same number of as and bs

Let \( p \) be the pumping constant. Choose \( w = a^p b^p a^p \). Since \( n_b = 2p \) and \( n_a = 2p \), and if \( x = a^p b^p \) then \( w = xx^p \), so \( w \in L \), and \( |w| > p \).

Case 1: \(|v| = 0, |y| = k, 0 < k < p\) and \( y \) contains only one symbol. Then when \( i = 0 \), \( 2p > 2p \) -k, so \( n_y \neq n_y \) so \( w_0 \notin L \). The case where \( y \) is empty and \( v \) is nonempty and contains all one symbol is symmetric. This also covers the case where both \( v \) and \( y \) are nonempty and both contain the same symbol (both \( v \) and \( y \) are all as, or \( v \) and \( y \) are all bs).

Case 2: \(|v| = 0, |y| = k, 0 < k < p\) and \( y \) contains two symbols. If \( y \) contains as followed by bs, when \( i = 2 \), \( w_2 = a^j a^m b^p a^p \) and \( w_2 \notin L \). If \( w \) contains bs followed by as, then \( i = 2 \). Then \( w_2 = a^j a^m b^p a^p \) and \( w_2 \notin L \). This also covers the case where both \( v \) and \( y \) are nonempty, but one of \( v \) or \( y \) contains two symbols.

Case 3: \(|v| = j, |y| = k, j, k > 0,\) both \( v \) and \( y \) contain only one symbol, but the symbol for \( v \) is different from that for \( y \). There are two subcases. If \( v \) contains as and \( y \) contains bs, then for \( i = 0 \), \( w_0 = a^p b^{p^2 + q} \). Then \( p - j < p \), so this is not a palindrome. Else, if \( v \) contains bs and \( y \) contains as, for \( i = 0 \), \( w_0 = a^p b^{p^2 + j} \). \( p - k < p \), so this is not a palindrome. In either case, \( w_0 \notin L \).

For any decomposition of \( w \), there exists an \( i \) such that \( w_i \notin L \), so \( w \) cannot be pumped using pumping for context-free languages, so \( L \) is not context-free.

5a) No. Let's prove this by providing a counter example. Let \( L = \{a^n b^n : n > 0 \} \). Then let \( G = (\{S\}, \{a, b\}, P, S) \) be a CFG with \( L(G) = L \), where \( P \) is:

\[
\begin{align*}
S &\rightarrow aSb | \varepsilon \\
\end{align*}
\]

Now make \( G' \) by adding the rule \( S \rightarrow SS | \varepsilon \)

\[
\begin{align*}
S &\rightarrow aSb | SS | \varepsilon \\
\end{align*}
\]

There is a string, \( w \), in \( L(G') \) that has the derivation

\[
S \rightarrow aSb \rightarrow aSSb \rightarrow aaSbSa \rightarrow aabSbaa \rightarrow aababb \rightarrow aababb
\]
but $w \notin L^*$, so $L(G') \neq L^*$.

5b) Yes. For this, let's consider a parse tree. The problem with the grammar from 5a is that, due to the recursive start variable, given a parse tree for a string, we could create some terminals, then later create additional copies of the start variable, and then create more terminals. Thus it is not the case that the terminals produced by a particular instance of $S$ (as the $S$ closest to those terminal leaves in the parse tree) must appear in a contiguous substring of the derived string. Put another way, in the grammar from 5a, we were able to make more copies of the start variable after generating other terminals and variables, and were therefore not able to guarantee that any strings produced must be composed of a string from the language concatenated with other strings from the language. Indeed, the actual result was an interleaving, where we inserted one string from $L$ in the middle of another string from $L$.

To guarantee this, we need to be sure that, if $w = x_1x_2...x_n$, and particular occurrence of $S$, $S'$, is the $S$ closest to leaves $x_j$ and $x_l$ with $j < k < l$, then $S'$ is also the occurrence of $S$ closest to $x_k$. For example, if $w = x_1x_2x_3x_4x_5 = aababb$, consider this derivation with occurrences of $S$ labeled:

- $S_1 \rightarrow aS_2b \rightarrow aS_3S_4b \rightarrow aaS_5bS_4b \rightarrow aabS_5b \rightarrow aabaS_6bb \rightarrow aababb$

using the derivation above, then $x_1$ and $x_6$ are produced by $S_1$, while $x_2$ and $x_3$ are produced by $S_3$ and $x_4$ and $x_5$ are produced by $S_4$.

Grammars in CNF afford us this guarantee when we add the rule $S \rightarrow SS | \varepsilon$ because they have a non-recursive start variable. Then there is no path in which a variable other than the start variable occurs between the root and an occurrence of the start variable. This ensures that we cannot insert a string produced by a derivation from $S$ (a string in $L$) into another string produced by a derivation from $S$ (another string in $L$). The rule does allow us to create two or more $S$ variables side by side in a sentential form, allowing us to produce strings from $L$ concatenated with strings from $L$. Alternately, we can also produce the empty string. Thus, it is the case that, given a grammar, $G$, in CNF, with $L = L(G)$, if we add the rule $S \rightarrow SS | \varepsilon$, we get exactly $L^*$.

To illustrate, let's look at the grammar from 5a converted to CNF.

- $S \rightarrow V_aV_1 | V_aV_b | \varepsilon$
- $S \rightarrow V_aV_1 | V_aV_b$
- $V_1 \rightarrow SV_b$
- $V_a \rightarrow a$
- $V_b \rightarrow b$

Now let's add the rule $S_0 \rightarrow S_0S_0 | \varepsilon$. Note that we're adding this instead of $S \rightarrow SS...$ because $S_0$ is now the start variable instead of $S$.

- $S_0 \rightarrow V_aV_1 | V_aV_b | S_0S_0 | \varepsilon$
- $S \rightarrow V_aV_1 | V_aV_b$
- $V_1 \rightarrow SV_b$
- $V_a \rightarrow a$
- $V_b \rightarrow b$

Since we began with a non-recursive start variable, there is no variable that can produce $S_0$ other than $S_0$ itself. If I apply any rule other than $S_0 \rightarrow S_0S_0$, I replace $S_0$ with some variable that cannot produce another $S_0$ later in the parse tree. Then all terminals produced by this $S_0$ will be a string from $L$. I can also produce as many $S_0$s as I want so I can concatenate as many strings from $L$ together as I want, or alternately, I can immediately apply $S_0 \rightarrow \varepsilon$ to produce the empty string.