Chapter 6

Context-Free Grammars

A context-free grammar (CFG) is another representation for a context-free language (CFL). CFGs are less restrictive on the right side of productions than regular grammars, so CFGs are able to generate a proper superset of the languages generated by regular grammars.

In this chapter we use JFLAP to construct and parse context-free grammars. We also show examples of context-free grammars and NPDA's that are equivalent by using JFLAP to convert a CFG into an equivalent NPDA and to convert an NPDA into an equivalent CFG.

6.1 Creating and Parsing Context-Free Grammars

In this section we will cover CFGs and show how to parse them in JFLAP. Then we will look at a simple, yet inefficient, parsing method. In Chapter 8 we will study more efficient parsing methods.

6.1.1 Entering a CFG in JFLAP

Section 3.1 described how to use the grammar input window. We will use the same grammar editor for inputting a CFG. Start JFLAP and choose Grammar. Input the grammar shown in Figure 6.1 or load the file ex6.1a.

<table>
<thead>
<tr>
<th></th>
<th>→</th>
<th>aSb</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>→</td>
<td>bB</td>
</tr>
<tr>
<td>S</td>
<td>→</td>
<td>bB</td>
</tr>
<tr>
<td>B</td>
<td>→</td>
<td>λ</td>
</tr>
</tbody>
</table>

Figure 6.1: A simple CFG.
6.1. CREATING AND PARSING CONTEXT-FREE GRAMMARS

A context-free grammar $G$ is defined using the four-tuple $G = (V, T, S, P)$, where $V$ is a set of variables, $T$ is a set of terminals, $S$ represents the start variable, and $P$ is a set of productions. Productions are in the form $A \rightarrow x$, where $A$ is a single variable and $x$ is a string of zero or more terminals and variables.

In JFLAP, these parts are represented as stated in Chapter 3.1. Symbols in the grammar are always represented as single characters, variables are represented by uppercase letters, terminals are represented as any symbol that is not an uppercase letter, and the start variable $S$ can be any variable. JFLAP assumes the start variable is the variable in the left side of the first production shown. For example, in Figure 6.1, the variables are \{S, B\}, the start variable is $S$, the terminals are \{a, b\}, and there are four productions shown.

6.1.2 Parsing with the Brute-Force Parser

What is the language of the grammar in Figure 6.1? Let's derive some strings using the brute-force parser we used in Chapter 3. Select Input: Brute Force Parse. Click the pop-up menu Noninverted Tree and select Derivation Table. Enter the string “aaabbbbb” in the text field labeled Input. Press Return, or click on the Start button. Repeatedly click Step until the string is derived. The productions used are shown on the left under the heading Productions and the sentential forms in the derivation are shown on the right under the heading Derivation, as shown in Figure 6.2.

Now we know one string in the language of this grammar. One way to figure out a description of all the strings in the language is to think about what type of string each variable can derive. We introduce the term $V$ production where $V$ is a variable to mean a production with $V$ on the left side. What can the variable $S$ derive? By looking at the first three sentential forms in the derivation of $aaabbbbbb$ in Figure 6.2, $S$ can derive \{$aSb, aaSbb, aaaSbb, \ldots$\}. By thinking about all the $S$ productions, we can see other sentential forms $S$ could derive, such as \{$bB, aabBb, aabBBbb, \ldots$\}, and come to the conclusion that $S$ can derive \{$a^nBb^n \mid n \geq 0$\}. If you focus on the variable $B$, what can it derive? Let’s force $B$ to be the start symbol by moving the first production past the last

<table>
<thead>
<tr>
<th>Production</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S$</td>
</tr>
<tr>
<td>$S \rightarrow aSb$</td>
<td>$aSb$</td>
</tr>
<tr>
<td>$S \rightarrow aSb$</td>
<td>$aaSbb$</td>
</tr>
<tr>
<td>$S \rightarrow aSb$</td>
<td>$aaaSbb$</td>
</tr>
<tr>
<td>$S \rightarrow bB$</td>
<td>$aaabBBbb$</td>
</tr>
<tr>
<td>$B \rightarrow bbB$</td>
<td>$aaabbbBbbb$</td>
</tr>
<tr>
<td>$B \rightarrow \lambda$</td>
<td>$aaabbbbBbb$</td>
</tr>
</tbody>
</table>

Figure 6.2: A derivation of $aaabbbbbb$. 
production and entering $B \rightarrow B$ as the first production. Enter in a few input strings to determine what $B$ can derive. $B$ can derive the strings \{$\lambda, bb, bbbb, bbbbbbb \ldots$\}, or rather $(bb)^*$, an even number of $b$'s. The combination of what $S$ and $B$ can derive implies the language of this grammar is \{${a^n}b{(bb)^*}b^n \mid n \geq 0$\}.

After experimenting with the $B$ productions, be sure to put the grammar back into its original form.

**Tip** In JFLAP you can determine the strings a particular variable $A$ can derive by making $A$ the start symbol. Enter the first production at the end of the production list to save it, and then enter $A \rightarrow A$ as the first production.

**Note** JFLAP only parses input strings, which are formed of terminals. If you enter a sentential form with a variable for the input, JFLAP gives an error message.

### 6.1.3 Parse Tree

A parse tree is a visualization of a derivation. Each production in the derivation is visualized as nodes in a tree: the symbols on the right side of the production are children and the symbols on the left side of the production are variables. For example, the production $B \rightarrow bbB$ would be visualized as the tree in Figure 6.3. A parse tree for an input string starts with the start variable as the root of the tree, and for each production that is applied, the corresponding variable node in the tree is expanded. The tree is complete when all the leaves in the tree are terminals or $\lambda$. Thus, the leaves read from left to right reproduce the input string.

In the grammar from Figure 6.1, we saw the derivation of the string $aaabbbbbbb$. Now let's see the same derivation as a parse tree. Click on **Start** again to restart the parse, then click on **Derivation Table** and select **Noninverted Tree**. The start variable is shown as a node. Click on **Step** once and $S$ is expanded into $aSb$. Note in the bottom left of the window that a message appears informing you of the production that was used, **Derived $aSb$ from $S$**. Click on **Step** repeatedly until the parse tree is complete. The completed parse tree is shown in Figure 6.4.

**Note** In JFLAP's parse trees, variables are shown as green nodes and terminals are shown as yellow nodes.

---

![Figure 6.3: A derivation visualized as a tree.](image-url)
Parse trees show the productions that are used in the derivation, but they do not keep the order the productions were applied in if there is more than one variable to be replaced. We will see this in a later example.

6.1.4 How Brute-Force Parsing Works

How does brute-force parsing determine a derivation for an input string? The Brute-Force Parsing Algorithm is so named because it takes a brute-force approach by trying all possible derivations. Internally to JFLAP (and not shown), it builds a tree of possible sentential forms, called a derivation tree. In this derivation tree the nodes are sentential forms, with the start variable as the root of the tree.

For example, consider the grammar from Figure 6.1 (load file ex6.1a if it is not loaded). Start the brute-force parser, enter the input string *aabb*, and click Start. Note the message below the input string that says String accepted! 5 nodes generated. The derivation tree for this string (JFLAP does not display this tree) is shown in Figure 6.5. The tree starts with the start variable and generates all possible sentential forms after one production replacement (*aSb* and *bB*). Since the sentential form *bB* starts with *b* and the string we are trying to derive starts with *a*, the node *bB* does not need to be expanded further (illustrated by the line drawn underneath it). All valid nodes are then expanded by one more production producing *aaSbb* and *abBS* (not valid), one more time producing *aaaSbb* (not valid) and *aabBb*, and finally *aabb* (circled since it is the input string!) and *aabbBbb* (not valid). The message above 5 nodes generated gives you an idea of
how large the derivation tree is, since JFLAP does not display it. In this case the '5' represents
the five nodes that were not immediately rejected as being impossible.

How efficient is brute-force parsing? Trying every possible combination works fine if the gram-
mar has few choices, but it could be extremely inefficient. JFLAP tries to prune the tree to cut
down on its size in several ways. JFLAP eliminates those sentential forms whose currently derived
 terminals are incompatible with the target string, those sentential forms it has already seen, and
those sentential forms whose length of non-λ-deriving variables and terminals is greater than the
length of the target string. Even with JFLAP's pruning, parsing can still take a long time. Dismiss
the tab and go back to the grammar and add the production $S \rightarrow SS$ to the grammar. Then select
the brute-force parser again. Enter the string $aabb$. This string was accepted before, so it should
be accepted again. It is, but this time there are 62 nodes in the tree instead of five! Figure 6.6
shows the first two levels of this derivation tree and illustrates how rapidly it is growing. Note that
the node labeled $SS$ derives six nodes after an additional step; however, two of them are $SSS$, so
JFLAP combines them into one node.

Enter in another input string $aaabbbbbb$ and click Start. This string was accepted before and
should be accepted now. Under the Input label you will see a message Parser running. Nodes
generated and a number followed by a second number in parentheses. The first number is the
current number of nodes in the parse tree that have not been rejected yet, and the second number

Figure 6.6: Start of the derivation tree of $aabb$ after adding $S \rightarrow SS$. 
is the number of nodes on the current level that are being examined. This number can be quite large! As you can see, the parsing is taking a long time due to the additional production and choices available. The parsing of this string will probably take too long, so you should try another string. To stop the parsing, click on Pause. If you really want to continue with this string, you can click Resume.

6.2 Converting a CFG to an NPDA

In this section we will use JFLAP to show how to convert a CFG to an equivalent NPDA with exactly three states using the LL parsing method. LL parsing will be explained in more detail in Chapter 8. The details of LL parsing are not needed for this conversion.

The idea behind the conversion from a CFG to an equivalent NPDA is to derive the productions through the stack. The conversion starts by pushing the start variable on the stack. Whenever there is a variable on the top of the stack, the conversion replaces the variable by its right side, thus applying a production toward deriving the string. Whenever a terminal is on top of the stack, the conversion pops the symbol off and makes sure that the current input symbol is the same. Thus the terminals on the stack are popped off in the same order they appear in the input string. If the stack is emptied, then all the variables were replaced and all the terminals matched, so the string is accepted.

6.2.1 Conversion

We will convert the CFG in Figure 6.1 (also in file ex6.1a) to an NPDA. Open the grammar if it is not already open and select Convert : Convert CFG to PDA (LL). You will see a view much like that in Figure 6.7. On the left is the CFG and on the right is the incomplete NPDA with three states. There are four transitions that have already been added to the NPDA. The transition from $q_0$ to $q_1$ pushes the start variable on the top of the stack. There is one transition from $q_1$ to $q_1$ for each terminal in the language. This transition ensures that when a symbol is read in the input, it must appear on top of the stack. The final transition from $q_1$ to $q_2$ ensures that the stack is empty before accepting. The transitions that correspond to the productions in the CFG are missing.

The goal for an input string is the following: If $q_2$ is reached, then all the symbols in the input string were pushed on the stack using the productions in the grammar and popped off the stack in the same order they appear in the input string. We have already added the “popped off” loop transitions on $q_1$. We must now add the “pushed on” loop transitions on $q_1$ for each production to get the terminals on the stack. For each production, the left side of the production is popped and the right side of the production is pushed.
Figure 6.7: Start of conversion of a CFG to an NPDA.

**Tip** If you don’t like the layout of the NPDA, use the Attribute Editor tool to rearrange the states.

In JFLAP, each production $A \rightarrow BCD$ will be replaced by a loop transition on state $q_1$ with the label $\lambda, A; BCD$. Add the missing transitions. Select the Transition Creator tool and click on state $q_1$. Input the transition $\lambda, S; aSb$ corresponding to the production $S \rightarrow aSb$. Since you created the transition corresponding to $S \rightarrow aSb$, the box to the right of this production is checked.

There is an alternative way to enter labels. Select a production whose corresponding transition has not yet been created and then click on Create Selected. JFLAP will add the corresponding transition. Add the remaining transitions. The resulting NPDA is shown in Figure 6.8. Select

Figure 6.8: Completion of converting a CFG to an NPDA.
Export, and the NPDA will be placed into a new window. To see that the NPDA accepts the same strings as the CFG, run the same input string \texttt{aaabbbbb} on the NPDA. Try other strings on both the CFG and the NPDA to convince yourself.

### 6.2.2 Algorithm

We describe the algorithm to convert a CFG $G$ into an equivalent NPDA $M$.

1. Start with a CFG $G = (V,T,S,P)$.
2. Create an NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ with three states such that $Q = \{q_0, q_1, q_2\}$, $q_0 = q_0$; $F = \{q_2\}$, $\Sigma = T$, and $\Gamma = V \cup T \cup \{Z\}$.
3. Create a starting transition, $\delta(q_0, \lambda, Z) = (q_1, SZ)$.
4. Create an ending transition, $\delta(q_1, \lambda, Z) = (q_2, Z)$.
5. For each terminal $a \in T$, create the transition $\delta(q_1, a, \lambda) = (q_1, \lambda)$.
6. For each production $A \rightarrow BCD \in P$, create the transition $\delta(q_1, \lambda, A) = (q_1, BCD)$.

### 6.3 Converting an NPDA to a CFG

We now show how to convert an NPDA to a CFG. The idea behind the conversion from an NPDA into an equivalent CFG is to convert each transition into one or more productions that mimic the behavior. To simplify the process JFLAP will require that the NPDA be in a certain format. If it is not, the user must convert the NPDA into the appropriate format. All transitions in the NPDA must pop exactly one symbol and push exactly zero or two symbols. In other words, with each transition the stack will increase its size by one or decrease its size by one. There are no restrictions on the number of terminals read on any transition. The NPDA must have one final state and at least one transition into the final state that pops $Z$ off the stack.

For those transitions that pop one symbol and push no symbols, one production is generated to mimic this behavior. For those transitions that pop one symbol and push two symbols, a lot of productions are generated to indicate possible ways these two symbols that are pushed could eventually be popped off the stack. Many useless productions will be generated in addition to the correct productions.

#### 6.3.1 Conversion

We will convert the NPDA in Figure 6.9 to an equivalent CFG. Load the NPDA from file ex6.3a and select Convert : Convert to grammar.
Oops! An error message appears, as shown in Figure 6.10. The NPDA is not in the correct format. The error message states that the transition from q1 to q2 is supposed to pop one symbol (it does) and push 0 or 2 symbols (it pushes only one). We must replace the transition with one state and two correctly formatted transitions. These two transitions simulate the deleted transition by pushing an additional symbol on the stack in the first transition and then popping that symbol off the stack in the second transition.

Figure 6.11 shows the NPDA with the transition b, a; b from before replaced with new state q5 and two transitions. The first new transition b, a; xb pops the a off the stack and pushes the b onto
the stack, as before. It also pushes the extra symbol $x$ onto the stack, so it is pushing two items on the stack. The second new transition $\lambda, x; \lambda$ pops the extra symbol off the stack.

Fix your NPDA. Now is it in the correct format? It has one start state, one final state, and all the transitions pop one symbol off the stack and push zero or two symbols onto the stack. Again select Convert : Convert to grammar. This time there should be no error message.

You will see a view much like that in Figure 6.12, except there is no highlighting yet in the NPDA and there are no productions yet on the right side.

To convert the NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ into a CFG $G = (V, T, S, P)$, we must define the parts of the grammar based on parts of the NPDA.

Variables in the grammar will be represented in the form $(q_i, A_q_j)$ where $q_i$ and $q_j$ represent states from the NPDA and $A$ is a stack symbol. The meaning of $(q_i, A_q_j)$ is, "if when moving along a path from state $q_i$ to $q_j$, the stack is exactly the same except that $A$ is removed from the stack."

The start variable $S$ will be represented by our goal, $(q_0, Z, q_4)$. Figure 6.13 illustrates its meaning. We start in $q_0$ with only $Z$ on the stack; we walk through some path (represented by the
squiggly lines), pushing and popping symbols off the stack; and eventually arrive in \(q_4\) with nothing on the stack. The stack changes size by decreasing by one between \(q_0\) and \(q_4\). (If other symbols are pushed on along the way, they must be popped off so the overall effect is the stack size decreased by one.)

We now describe how to convert each transition into equivalent productions. There are two types of transitions. First, we will describe how to generate a production for a transition that decreases its size by 1 by looking at the transition from \(q_5\) to \(q_2\), \(\lambda, x; \lambda\). Note the stack contents will stay exactly the same except one symbol (\(x\)) is popped off the stack, as shown in Figure 6.14.

For transitions of this type \(\delta(q_i, A, q_j) = (q_j, \lambda)\), there is one production equivalent,

\[(q_i, A, q_j) \rightarrow a,\]

meaning from \(q_i\) to \(q_j\) there is one symbol popped (\(A\)) and one symbol from the input (\(a\)), so we will derive this symbol.

To create the corresponding productions for a transition, click on the transition. Click on the transition \(\lambda, x; \lambda\) from \(q_5\) to \(q_2\) and you will see the equivalent production appear on the right side, \((q_5, x, q_2) \rightarrow \lambda\). Also click on the other transitions in which the size of the stack decreases by one, \(c, b; \lambda\) and \(d, Z; \lambda\), and you will see the equivalent productions. These are the first three productions shown in Figure 6.12.

Figure 6.14: When the stack decreases in size by 1.
Figure 6.15: Meaning of when the stack increases in size by 2.

Now we will see how to convert a transition in which the size of the stack increases in size by one by looking at the transition $b, a; xb$ from $q_1$ to $q_5$. For transitions of this type $\delta(q_i, a, A) = (q_j, BC)$ there are many productions generated. For all possible $q_k$ and $q_l$ we create the productions

$$(q_iAq_k) \rightarrow a(q_jBq_l)(q_kCq_k)$$

for all $q_k$ and $q_l$.

Figure 6.15 illustrates the meaning of this. In going from $q_i$ to $q_k$, the symbol $A$ is popped off the stack and the stack is exactly the same as it was in $q_i$, except the $A$ is no longer on the stack. The symbol $Y$ is shown to illustrate that there might be other symbols on the stack below the $A$ in $q_i$, and that those symbols should still be on the stack in $q_k$. Along the path from $q_i$ to $q_k$ there are other symbols that are pushed on $(B$ and $C$), but they must be pushed off before reaching $q_k$.

Click on the production $b, a; xb$ and you will see many productions generated and highlighted on the right side. Many of these productions will be useless, but some will be needed to derive a string. For an example, you should see the production

$$(q_1aq_3) \rightarrow b(q_5xq_2)(q_2bq_3).$$

Figure 6.16 illustrates this production. Starting in $q_1$ the $a$ is on top of the stack. In $q_3$ the stack is exactly the same, except the $a$ is removed from the stack. Along the way, $b$ and $x$ were pushed onto the stack and removed.

Figure 6.16: Example of when the stack increases in size by 2.
Click on the button **What's left?**. It should show that you have one more transition to convert to productions. Click on that transition and many productions are added to the grammar. You may have to scroll down to see the highlighted block of productions.

If the generated CFG is small enough, then you can can export the grammar to another window. Click on **Export**, and you will see that in this case the grammar is too large to export. Since we cannot see the derivation in JFLAP, a derivation of the string \(abcd\) follows so you can see that our conversion does accept a string in the language. You can run the same input string on the NPDA to see that it also accepts this string.

\[
(q_0 Z q_4) \Rightarrow a(q_1 a q_3) (q_3 Z q_4) \\
\Rightarrow a(q_1 a q_3) d \\
\Rightarrow ab(q_5 x q_2) (q_2 b q_3) d \\
\Rightarrow ab(q_2 b q_3) d \\
\Rightarrow abcd
\]

### 6.3.2 Algorithm

We describe the algorithm to convert an NPDA \(M\) into a CFG \(G\).

1. Start with an NPDA \(M = (Q, \Sigma, \Gamma, \delta, q_s, Z, F)\).

2. Modify \(M\) to have one final state \(q_f\).

3. Modify \(M\) such that each transition pops exactly one symbol and pushes either zero or two symbols.

4. Create a CFG \(G = (V, T, S, P)\) with \(V = \{(q_i A q_j) \mid q_i, q_j \in Q \text{ and } A \in \Gamma\}\) and with the start variable equal to \((q_s Z q_f)\).

5. For each transition \(\delta(q_i, a, A) = (q_j, \lambda)\), generate one production

   \[
   (q_i, A, q_j) \rightarrow a.
   \]

6. For each transition \(\delta(q_i, a, A) = (q_j, BC)\), generate the productions

   \[
   (q_i A q_k) \rightarrow a(q_j B q_l)(q_l C q_k) \text{ for all } q_k \text{ and } q_l.
   \]
6.4 Summary

In Section 6.1 we learned how to parse strings with a grammar using the brute-force method. A derivation of a string may be visualized either as a derivation table or a parse tree. The derivation table preserves the order the productions are applied; the parse tree does not. We also showed that brute-force parsing can be very inefficient.

In Section 6.2 we learned how to convert a CFG to an equivalent NPDA using the LL parsing method. The NPDA has exactly three states, and starts by pushing the start variable on the stack. Transitions are created that mimic productions by popping the left side of a production and pushing down its right side. Other productions are created to make sure the symbols read in the input string are popped off the stack in the same order.

In Section 6.3 we learned how to convert an NPDA to an equivalent CFG. The NPDA must first be modified to have one final state, and transitions that always pop exactly one symbol and push exactly zero or two symbols. Each transition that pushes zero symbols can be converted into one production in the grammar. Each transition that pushes two symbols is converted into many productions that mimic possible scenarios in which these two symbols can be eventually popped off the stack.

6.5 Exercises

1. For each CFG, list five strings in the language and give a written description of the language.

(a) ex6.5cfg-a
(b) ex6.5cfg-b
(c) ex6.5cfg-c
(d) ex6.5cfg-d

2. For each language listed, the corresponding CFG does not correctly represent the language. List a string that either is in the language and is not represented in the grammar, or is not in the language but is accepted by the grammar. Then modify the grammar so that it represents the language.

(a) \( L = \{a^nb^mc^p \mid 0 < n < m + p, m > 0, p > 0 \} \) and file ex6.5cfg-e
(b) \( L = \{(ab)^n(ba)^m \mid n > 0, m > 0 \} \) and file ex6.5cfg-f

3. Consider the ambiguous grammar in file ex6.5cfg-ambiguous and the string \textit{ababab}. Run the brute-force parser on this string. Which parse tree does JFLAP produce? Give two different parse trees for this string.
4. Write a CFG for each of the following languages. Create a set of test strings that includes strings that should be in the language and strings that should not be in the language, and verify with JFLAP.

(a) \( L = \{ a^nb^m \mid m < n, m > 0 \} \)
(b) \( \Sigma = \{ a, b \}, L = \{ w \in \Sigma^* \mid \text{aba is a substring and} \ n_a(w) \text{ is even} \} \)
(c) \( L = \{ a^nb^mc^n \mid n > 0, m > 0, n \text{ is even, } m \text{ is odd} \} \)
(d) \( L = \{ a^nb^m c^p \mid p = n + m, n > 0, m > 0 \} \)
(e) \( L = \{ a^nb^m c^n d^n \mid n \geq 0, m > 0 \} \)
(f) \( L = \{ a^nb^m \mid 0 < n \leq m \leq 3n \} \)

5. Consider the CFG in file `ex6.5-longparse` and the strings of the form \( a^nb^b \).

(a) Run the brute-force parser for \( n = 2, \ldots, 8 \) and record the number of nodes generated. What observation can you make about the time to parse?
(b) Remove the production \( S \rightarrow SS \) and again run the brute-force parser for \( n = 2, \ldots, 8 \) and record the number of nodes generated. What observation can you make about the time to parse?
(c) In addition to the previous change, change the production \( A \rightarrow AA \) to \( A \rightarrow aA \). Run the brute-force parser for \( n = 2, \ldots, 8 \) and record the number of nodes generated. What observation can you make about the time to parse?
(d) Is the modified grammar equivalent to the original grammar?

6. Convert each of the following CFGs into an NPDA. Export each NPDA and trace several strings in both the CFG and the NPDA.

(a) `ex6.5cfg-a`
(b) `ex6.5cfg-b`
(c) `ex6.5cfg-c`
(d) `ex6.5cfg-d`

7. For each of the following NPDA, list the language of the NPDA and convert the NPDA into a CFG. Export the CFG and trace several strings in both the NPDA and the CFG.

(a) `ex6.5-toCFGa`
(b) `ex6.5-toCFGb`
(c) `ex6.5-toCFGc`
8. Convert each of the following NPDA's into a CFG. You will first have to put the NPDA in the correct format. Once you have, save the modified NPDA and then run several input strings on both NPDA's to make sure that the modified NPDA represents the same language as the original NPDA. Then convert the NPDA to a CFG. The CFG will be too large to export.

(a) ex6.5-toCFGd
(b) ex6.5-toCFGc
(c) ex6.5-toCFGf