Converting Context Free Grammars (CFG) to Chomsky Normal Form (CNF)

In the textbook, a procedure for converting CFGs to CNF is provided in example 2.10, on pages 110-111. I have gone through this example on the lecture on Wed Mar 1st. This example shows step by step how each transformation is made on the grammar; however, it is too brief to provide context for why the changes are made. In this document, I will try to cover this example in a greater amount of depth than what is provided in the book and my slide so that you can apply the method to other problems in the future.

Let's first recap the definition of CNF. This is any context free grammar consisting of only the following types of rules:

1. A → BC. The right-hand side is exactly two non-terminals (we represent these with capital letters in the book's notation). **Neither of these non-terminals may be the start state**. The left-hand side may be the start variable.
2. A → a. The right-hand side is a singular terminal, **which is not ε** (these terminals are represented with lower-case letters in the book's notation).
3. S → ε. The left-hand side is the start variable, and the right-hand side is ε.

From these rules, we can tell that the only case where ε is allowed is as a rule from the start state. Furthermore, since the start state can't be on the right-hand side of any rule, we effectively make it so that any string other than the empty string is generated using rules that don't involve ε. Every time we substitute a non-terminal, we either increase the number of non-terminals by 1 or decrease it by 1 and add a terminal.

For reference, the following types of rules are invalid in CNF:

1. A → BCD. (More than 2 non-terminals)
2. A → B. (Less than 2 non-terminals)
3. A → Bc. (Non-terminal and a terminal are mixed)
4. A → BS. (Start variable is part of the right-hand side)
5. A → ε. (Non-terminal other than start state on left-hand side of ε)

In the example 2.10, we are given the following grammar that contains a mix of all these types of invalid rules.

S → ASA | aB
A → B | S
B → b | ε

The ASA rule is an example of the 1st and 4th type of invalid rule, aB is an example of the 3rd type, B and S are examples of the 2nd type (and also 4th type), and the B → ε rule is an example of the 5th type. On the next page, I will demonstrate each of the ways these invalid types of rules are removed.

Step 1: Make a new start variable. This step is only necessary if the 4th type of invalid rule is present. Creating a new start variable allows other rules to contain references to the old start variable, while nothing has the new start variable as part of its right-hand side. This will remain the case for the rest of the procedure. The new grammar in our example looks as follows, with the addition depicted in bold:

$\mathbf{S_0 \rightarrow S}$
$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$

Step 2: Eliminate ε rules. This is where the complexity starts. We need to subtract rules while simultaneously adding new rules, to produce a grammar that is **equivalent** to the old one, but closer to being proper CNF. Our book example shows two sets of changes, with additions marked in bold, and subtractions shown in a sort of grey font. Let's look at the first one:

$S_0 \rightarrow S$
$S \rightarrow ASA \mid aB \mid \mathbf{a}$
$A \rightarrow B \mid S \mid \boldsymbol{\varepsilon}$
$B \rightarrow b \mid \varepsilon$

Okay, so the first thing was to remove the ε from B, which is good. But then we added ε to A? Why? Well, the answer was that A has a rule that allows it to be B, and since B could previously be ε, we have to make this new rule to make things equivalent. Don't worry though, we can totally remove all of the ε rules, just by having an iterative process. You should also note the second addition: $S \rightarrow a$. This is because we had $S \rightarrow aB$, and since B could previously have been the empty string, we effectively delete it to allow a new rule. Okay, second iteration!

$S_0 \rightarrow S$
$S \rightarrow ASA \mid aB \mid a \mid \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S}$
$A \rightarrow B \mid S \mid \varepsilon$
$B \rightarrow b$

Now we're removing the ε from A, and adding to S all of the new rules where A would be replaced with ε. ASA could be SA if the first A was ε, AS if the second was ε, or S if both were ε. So in other words, we're just applying substitutions until we can get rid of the ε. It is possible that we may end up pushing the ε all the way up to the start variable. In that case, if it's the only rule left, we can leave it there. The empty string is in the language after all.

Step 3: Eliminate unit rules. Here we want to eliminate every case we have some $A \rightarrow B$ type of rule, including our original rule for the start state, which is $S_0 \rightarrow S$. As in the previous step, we are both deleting rules and adding new ones. Thankfully, this step is actually quite easy, because every substitution in this step is to either delete a redundant rule like $A \rightarrow A$, or replace a rule like $A \rightarrow B$ by copying all of B's rules onto A's rules. So if we had $B \rightarrow c \mid DE$, then we now have $A \rightarrow c \mid DE$ instead of $A \rightarrow B$. The following are all of the replacements from the book, with the same **bold for addition** and grey for deletion pattern we have seen previously.

Eliminate redundant state:

> $S_0 \rightarrow S$
> $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S$
> $A \rightarrow B \mid S$
> $B \rightarrow b$

Copy-paste S onto S_0

> $S_0 \rightarrow S$ **| ASA | aB | a | SA | AS**
> $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
> $A \rightarrow B \mid S$
> $B \rightarrow b$

Copy-paste B and S onto A

> $S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
> $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
> A → **b | ASA | aB | a | SA | AS**
> $B \rightarrow b$

And now we only have invalid rules of type 1 (more than 2 non-terminals) and 3 (mixed terminals and non-terminals). This leaves us with one final step:

Step 4: Fix everything else by adding additional rules. This is where many of you may be confused because while the book provides an example of the final CNF grammar, it doesn't explain how to do this. So, let's look at our grammar again. We have two big rule violations: cases where the right-hand side is aB, and cases where the right-hand side is ASA.

The first case is easy to fix. For every terminal symbol, let there be an associated non-terminal. We already have B → b, so create something like U → a. Now replace all instances of a with U, unless it's an already valid rule (like S → a). This should leave us with only one type of invalid rule – the case of having more than two non-terminals in a rule.

The second case, similarly involves creating new rules, and replacing non-terminals. With a string of k non-terminals, we can turn it into k-1 by creating a non-terminal for the first two symbols of it. So for S → ASA, we can make a rule Y → AS from the first two non-terminals, and turn the new rule into S → YA. With three non-terminals, this step needs only be done once, but with more, we just repeat it. If we had A → BCDE, we could make F → BC, then G → FD, and finally let A → GE would be the replacement for our old rule.

Following these steps should turn every rule in a CFG that is invalid for a CNF, into a set of rules that are valid.