Debugging Bad CFG Using User Control Parse Exercise
Martha Kosa

From your programming coursework, you should know that you must debug your programs if they do not produce the desired output when run. You discover errors or bugs (as coined by Grace Hopper) via testing. You then must modify your code to fix the errors and retest it to verify that the bugs have been fixed. The purpose of a context-free grammar is to generate a language. You design a context-free grammar by creating production rules. From the production rules, the other parts of the grammar (nonterminal symbols, terminal symbols, and starting nonterminal symbol) can be easily extracted. In the absence of a formal proof that your grammar generates exactly the strings of the language (which may not be easy for nontrivial languages containing infinitely many strings), you should test your grammar to gain confidence that your grammar only generates strings in the corresponding language and that strings in the corresponding language can be generated by the grammar.

How is a string generated by a grammar? A string of terminal symbols is generated by a grammar if it can be derived from the start symbol by applying production rules. The string consisting of only the start symbol is the initial sentential form. How is a production rule applied? A production rule can be applied if the current sentential form has at least one nonterminal symbol that corresponds to the lefthand side of the rule; the nonterminal symbol is replaced by the rule's righthand side, resulting in a new sentential form, which becomes the current one. If the sentential form has no nonterminal symbols, the corresponding string is generated by the grammar.

Consider the language $\{a^{3n} b^{3n} \mid n \geq 1\}$. The shortest three strings in the language are aaabbb, aaaaaabbbbbb, and aaaaaaaaabbbbbbbbb. Let's test this proposed grammar to generate the language. The production rules are as follows:

- S → aaaS
- S → bbbT
- T → bbbT
- T → bbb

Does this grammar generate exactly the set corresponding to the language above? It should generate every string in the set, and it should NOT generate any string NOT in the set.

Let's find out! Build a CFG with the four rules as given above.

Your grammar should look like the following:

The first question to ask is if the grammar can generate the shortest string in the language, aaabbb. Let's use JFLAP's **User Control Parse** feature to investigate.

Try It!

1. Select ***Input > User Control Parse***.
2. Enter **aaabbb** in the box next to **Input**. Your JFLAP window should look similar to the following, after possibly adjusting your window and/or subwindow sizes.

1. Click the **Start** button. What happened in the bottom right subwindow? Why?
2. Click on the one applicable production rule, and then the **Step** button.
3. Click on the one applicable production rule, and then the **Step** button.
4. You now have two applicable production rules. Click on the one with the shorter righthand side, and then the **Step** button. What message do you see below the **Input** section?
5. Click the **Previous** button, click on the applicable production rule with the longer righthand side, and then the **Step** button. Will it ever be possible to generate your desired string? Why or why not?
6. Choose **Derivation Table** instead of the default **Noninverted Tree** in the combo box to the right of the **Step** button, and repeat Steps 3 through 7 above. This shows you the progression of sentential forms. Where can you find the equivalent sentential forms in the noninverted tree?

We have discovered one bug in our grammar so far. There is a string in the language, **aaabbb**, that cannot be generated. Are there other bugs? Can invalid strings be generated? Let's investigate!

In every production rule of the grammar, the number of a's and the number of b's on the righthand side is a multiple of 3, so we can never generate a string with the number of a's or the number of b's not being a multiple of 3. Remember that 0 is a multiple of 3. You can verify this using some sample strings (such as **aaaabb**) and JFLAP's **User Control Parse**, as before.

Because of the recursive **S → aaaS** rule and the **S → bbbT** rule, any b's will always follow a's. You can verify this using some sample strings (such as **bbbaaa**) and the **User Control Parse**.

What other possible bug can occur? What about strings with a group of a's followed by a group of b's in which the number of b's does not match the number of a's? We have two choices now for the relationship between the number of b's and the number of a's. Either the number of b's is larger than the number of a's, or the number of b's is smaller than the number of a's.

Try It!

1. What happens when you attempt to parse **aaabbbbbb**?
2. What happens when you attempt to parse **aaaaaabbb***?*
3. The string **aaaaaabbb** cannot be generated. The rules **S** ⍰ **bbbT**, **T** ⍰ **bbbT**, and **T** ⍰ **bbb** cause any valid string to have at least six consecutive b's. What is the smallest candidate string containing at least six consecutive b's that are preceded with a larger number of a's that is a multiple of 3? What happens when you attempt to parse that string?

The problem with our grammar is that the number of a's is not tied to the number of b's. When an a is generated, a b should also be generated. Our grammar is the wrong type of grammar. It is a right-linear grammar, which can be converted to a regular grammar. By using the Context-Free Pumping Lemma, we can prove that the language is not regular; thus, a regular grammar will never work. We need to go back to the drawing board.

Consider the first few valid strings of our language. The string **aaabbb** is our smallest string, and then follows **aaaaaabbbbbb**. Notice that the pattern **aaabbb** appears in the middle of **aaaaaabbbbbb**, with **aaa** preceding it and **bbb** following it. This suggests a recursive rule: **S** ⍰ **aaaSbbb**. We can stop the recursion by using the rule **S** ⍰ **aaabbb**.

Try It!

1. Create a new JFLAP grammar with the two rules described above. Your grammar should look like the following.

1. Attempt to parse the strings mentioned in the previous exercises with the **User Control Parse** feature. It should be possible to generate the valid strings, and it should be impossible to generate the invalid strings. You should see a result similar to the following when attempting to parse **aaaaaaaaabbbbbb**.