Unrestricted Grammars
Martha Kosa

The most important part of a grammar is its set of production rules because the productions specify the structure of the generated strings. In a context-free grammar, the form of each rule is very strict; the left-hand side must be a single nonterminal symbol, and the right-hand side is any combination of terminal and/or nonterminal symbols. The strict rule allow for simple parsing algorithms, such as the breadth-first top-down parsing algorithm. What happens if we relax the restriction on the left-hand sides for grammar rules? Anything goes now, as long as the left-hand sides are not [illegible]. We cannot create something from nothing. With the less restrictive left-hand sides, more languages can be generated. They will not be context-free.

Consider the language $\{0^n1^n \mid n \text{ [illegible] } 0\}$. The first few strings in this language are [illegible], 01, 0011, and 000111. The following context-free grammar generates the language:

What happens if we make a slight change to the language. Consider the language $\{0^n1^n2^n \mid n \text{ [illegible] } 0\}$. The first few strings in this language are [illegible], 012, 001122, and 000111222. We cannot make simple modifications to the above context-free grammar to generate the language. The Context-Free Pumping Lemma can be used to prove that the language is not context-free; thus, no context-free grammarexists to generate the language.

We can think about a simple modification to the language that is context-free, produce a context-free grammar for the modified langugage, and add some non-context-free rules to that grammar to obtain our desired language.

Consider the language $\{0^n(12)^n \mid n \text{ [illegible] } 0\}$. The first few strings in this language are [illegible], 012, 001212, and 000121212.

You may have noticed that we had a few extra rules. The nonterminal symbol M is to help distinguish between [illegible] and nonempty valid strings. S derives, via leftmost derivations, sentential forms of the form $00^kM(1T)^k12$ for each k [illegible] 0. The next leftmost derivation will apply the M [illegible] [illegible] rule to yield $00^k(1T)^k12$.

We now will modify some rules to produce our desired language. Notice that every T will be immediately followed by a 1. This will help us in developing our non-context-free rules. We need to move the 1's to be before all the T's so they will be clustered together. The rule T1 [illegible] 1T will help with this. The resulting sentential forms will be of the form $00^k11^kT^k2$ for each k [illegible] 0.

We cannot keep the rule T → 2 any more because this will allow invalid strings to be generated; we were not absolutely required to perform leftmost derivations. We will modify this rule to use some context. We will complete our valid strings from right to left. Notice that the rightmost T will be followed by 2 when k ≥ 1. We will replace the T → 2 rule with the non-context-free rule T2 → 22. We can apply this rule k times. Then we will have our desired $0^n1^n2^n$, where n ≥ 1. For n = 0, we just apply the S → rule. We show our modified grammar and a sample derivation.

Notice the grouping in the parse tree when the non-context-free rules are applied.

We also show the derivation so that you can see the ordering of the sentential forms.

JFLAP provides a feature to test if your grammar is context-free. Select ***Test > Test for Grammar Type***. If your grammar is not context-free (and our example is not), you will see the following dialog box.

If you test a true context-free grammar, you will see the following dialog box.

Try It!

1. What happens when you attempt to parse the strings 012, 021, 102, 120, 210, and 201?
2. How many strings of length 6 over = {0,1,2} have two 0's, two 1's, and two 2's?
3. How many of those strings above does our grammar generate?
4. Produce a grammar to generate the language $\{hip^n hop^n hooray^n \mid n \geq 0\}$.
5. Produce a grammar to generate the language $\{0^n1^n2^{2n} \mid n \geq 0\}$.

Let's do another example. Suppose that we want to produce strings consisting of two parts, where the first part is a word, and the second part is a formatted version of the word, such as an underlined version. What is the corresponding language? Let us assume that is our alphabet. For simplicity, we will repeat the word for the second part and delimit it by square brackets. We assume that words will not contain square brackets. This yields $\{w[w] \mid w \in \Sigma^*\}$. Some strings in the language are [] and cool[cool]. We can consider this language to be a language of echoed strings. In the real world, when you are setting up a new user account, you have to enter your proposed password a second time for verification. This is analogous to an echo.

What happens if we try to use a context-free grammar? We need to make a copy of each letter, and we need a left square bracket in the middle and a right square bracket at the end. Here is our first attempt.

What happens when we try to parse cool[cool]?

The string is rejected. Let's investigate further by performing a user-controlled parse.

Try It!

1. Select ***Input > User Control Parse*** to begin.
2. Enter **cool[cool]** in the input box.
3. Push the **Start** button.
4. Select the appropriate grammar rule to apply. There is only one valid possibility at this point. Push the **Step** button.
5. What is the current sentential form? Remember it can be found by concatentating the leaves of the parse tree in left to right order.
6. Continue applying appropriate grammar rules (there will still only be one valid possibility at each step) until you cannot apply them any more.
7. Why is the string rejected? What is the relationship between the substring before the [and the substring between the [and the]?

Below is the result when all possible grammar rules have been applied.

This grammar produces strings of the form $\{w[w^R] \mid w \in \Sigma^*\}$, where w^R is the reverse of w. It is impossible to produce a context-free grammar generating our desired language; the Context-Free Pumping Lemma can help prove the impossibility result.

What can we do? We need to reverse the substring between the [and the]. We can add some non-terminal symbols and some non-context-free rules. We can also distinguish between empty and non-empty parts, as in the previous example.

Try It!

1. Select ***Input > Brute Force Parse*** to begin.
2. Enter **cool[cool]** in the input box.
3. Push the **Start** button.
4. Why is there a delay before you learn that the string is accepted?
5. Push the **Step** button as many times as necessary to parse the string. Your JFLAP window should look like the following.

1.
2. Which grammar rules are responsible for generating all the characters before the [and between the [and the]?
3. How many [symbols appear in each sentential form?
4. How many] symbols appear in each sentential form?
5. Select **Derivation Table** from the combo box where **Noninverted Tree** is.
6. Push the **Start** button again.
7. Which grammar rules are responsible for producing the correct order for the symbols between the [and the]?

Questions to Think About:

1. How many rules are needed if Σ = {a, b, ..., z}? How will our set of nonterminal symbols be affected?
2. How many rules are needed if Σ = {0, 1, ..., 9}?

Try It!

1. Enter **cool[coo]** in the input box.
2. Push the **Start** button.
3. What is the result? Why?
4. Select ***Input > User Control Parse***.
5. Push the **Start** button.
6. Attempt to apply grammar rules until you get stuck. You can use the **Derivation Table** or the **Noninverted Tree** from the combo box. You should end up with something similar to the following.

You have just seen an example of a string that cannot be generated by our non-context-free grammar. It does not consist of a substring followed by a [then a copy and then a final].

In many programming languages that you use, variables must be declared before they are used. This is a form of echoing. The language {**int** w**;** w = 5; | w Σw Σ Σ^*} is not context-free. How do the compiler writers handle this? The grammars already have hundreds of rules; they don't need hundreds more. The compilers use lexical analysis to break programs up into categories of tokens (variable names – identifiers, keywords – such as if, predefined types – such as int, literals – such as decimal numbers, etc.). Variable names are stored in data structures called symbol tables. Semantic actions are used to simplify the grammars. Since the set of languages generated by context-free grammars is equivalent to the set of languages generated by pushdown automata, context-free grammars use stacks implicitly. Remember that stacks are FIFO data structures; we need data structures that support more general access to handle declaration before use with respect to variables. This makes us appreciate the compiler writers; without them, we would have to program in machine code with 0's and 1's.

Congratulations! You have explored unrestricted grammars and have seen some practical examples of non-context-free languages.