CS 4510 : Automata and Complexity
Ambiguity, ε Rules, Unit Rules

Subrahmanyam Kalyanasundaram

January 28, 2010

1 Ambiguity

Definition 1 (Leftmost Derivation)**.** A *leftmost derivation* of a string w in a grammar G is when at every step of the derivation, the leftmost variable is the one which is replaced.

Consider the following CFG.

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \mathtt{a}$$

The set of variables is $V = \{\langle \text{EXPR} \rangle\}$ and the set of terminals is $\Sigma = \{\mathtt{a}, +, \times, (,)\}$. Below we see two different ways that the grammar can generate the string $\mathtt{a} + \mathtt{a} \times \mathtt{a}$. These correspond to the figure 2.6 in page 106 in the text book. First

- $\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle^{(1)} \times \langle \text{EXPR} \rangle^{(2)}$
- $\langle \text{EXPR} \rangle^{(1)} \to \langle \text{EXPR} \rangle^{(3)} + \langle \text{EXPR} \rangle^{(4)}$
- $\langle \text{EXPR} \rangle^{(3)} \to \mathtt{a}$
- $\langle \text{EXPR} \rangle^{(4)} \to \mathtt{a}$
- $\langle \text{EXPR} \rangle^{(2)} \to \mathtt{a}$

Notice that I am using superscript (1), (2) etc. to just differentiate between the different variables in different lines. Now the second derivation.

- $\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle^{(1)} + \langle \text{EXPR} \rangle^{(2)}$
- $\langle \text{EXPR} \rangle^{(1)} \to \mathtt{a}$
- $\langle \text{EXPR} \rangle^{(2)} \to \langle \text{EXPR} \rangle^{(3)} \times \langle \text{EXPR} \rangle^{(4)}$
- $\langle \text{EXPR} \rangle^{(3)} \to \mathtt{a}$
- $\langle \text{EXPR} \rangle^{(4)} \to \mathtt{a}$

Both of these are leftmost derivations (since we are interpreting the leftmost variable first in all cases), and notice that they really correspond to two different derivations. They look like they were supposed to mean two different things, but they have ended up being the same because the grammar was not very well created. The first one seems to mean $(\mathtt{a} + \mathtt{a}) \times \mathtt{a}$ while the second one seems to mean $\mathtt{a} + (\mathtt{a} \times \mathtt{a})$. The two derivations are not the result of changing the order of one to get the other. Hence they are ambiguous.

Definition 2 (Ambiguity)**.** A string w is derived *ambiguously* in a context free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

Certain languages can be created only using ambiguous grammars. These languages are called *inherently ambiguous*.

Please read example 2.4 (page 103) in the book to see how the above grammar issue is fixed by using more variables and adding more structure to the language, to get the same language, but with a grammar with no ambiguities.

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid \texttt{a} \end{aligned}$$

2 Loops, Unit Rules and ε Rules

In this section, I hope to give a small motivation for the need for having no ε rules, or unit rules. The computer, or machine has to parse and interpret the rules to check if a string is in a language or not. So how would a machine do it? Suppose the rules generating the language are the following.

$$\begin{aligned} S &\rightarrow 0S1 \mid 1S0S1 \mid T \\ T &\rightarrow S \mid \varepsilon \end{aligned}$$

The machine has to check if a given string, say $x = 101100$ is in the language or not. One way to do it would be to keep constructing all the strings that we could from the grammar. The machine checks if the string x is generated. If the string is generated, the machine can conclude that it is in the language. But if it is not in the language, how long should it run before it decides otherwise? The machine could stop once it starts getting strings which are longer than $|x| = 6$. Now, this would be a problem if the language had empty rules. The machine cannot enforce a stop rule, because the string could get shorter in length because of the empty rules. This is why empty rules are undesirable.

Another issue is looping. We know about infinite loops in programs. In the above grammar, we have two rules $S \rightarrow T$ and $T \rightarrow S$. The machine runs the risk of running into an infinite loop with these two rules. This is why unit rules are undesirable. Without any increase in the length of the resulting string, the machine might go on indefinitely interpreting the above rules.

The Chomsky normal form gives us a grammar which has neither empty rules nor unit rules. This makes it possible to create an algorithm, with a guaranteed stop rule, that can determine if a string is in the given grammar or not. We shall see the Cocke-Younger-Kasami algorithm which does this, in the future classes.