Lecture 8

Parse trees, ambiguity, and Chomsky normal form

In this lecture we will discuss a few important notions connected with context-free grammars, including *parse trees*, *ambiguity*, and a special form for context-free grammars known as the *Chomsky normal form*.

8.1 Left-most derivations and parse trees

In the previous lecture we covered the definition of *context-free grammars* as well as *derivations* of strings by context-free grammars. Let us consider one of the context-free grammars from the previous lecture:

$$S \to 0\,S\,1\,S \;\big|\; 1\,S\,0\,S \;\big|\; \varepsilon. \tag{8.1}$$

Again we'll call this CFG G, and as we proved last time we have

$$\mathrm{L}(G) = \big\{ w \in \Sigma^* \,:\, |w|_0 = |w|_1 \big\}, \tag{8.2}$$

where $\Sigma = \{0,1\}$ is the binary alphabet and $|w|_0$ and $|w|_1$ denote the number of times the symbols 0 and 1 appear in w, respectively.

Left-most derivations

Here is an example of a derivation of the string 0101:

$$S \Rightarrow 0\,S\,1\,S \Rightarrow 0\,1\,S\,0\,S\,1\,S \Rightarrow 0\,1\,0\,S\,1\,S \Rightarrow 0\,1\,0\,1\,S \Rightarrow 0\,1\,0\,1. \tag{8.3}$$

This is an example of a *left-most derivation*, which means that it is always the left-most variable that gets replaced at each step. For the first step there is only one variable that can possibly be replaced—this is true both in this example and in general. For the second step, however, one could choose to replace either of the occurrences of the variable S, and in the derivation above it is the left-most occurrence that gets replaced. That is, if we underline the variable that gets replaced and the symbols and variables that replace it, we see that this step replaces the left-most occurrence of the variable S:

$$0\,\underline{S}\,1\,S \Rightarrow 0\,\underline{1\,S\,0\,S}\,1\,S. \tag{8.4}$$

The same is true for every other step—always we choose the left-most variable occurrence to replace, and that is why we call this a left-most derivation. The same terminology is used in general, for any context-free grammar.

If you think about it for a moment, you will quickly realize that every string that can be generated by a particular context-free grammar can also be generated by that same grammar using a left-most derivation. This is because there is no "interaction" among multiple variables and/or symbols in any context-free grammar derivation; if we know which rule is used to substitute each variable, then it doesn't matter what order the variable occurrences are substituted, so you might as well always take care of the left-most variable during each step.

We could also define the notion of a *right-most derivation*, in which the right-most variable occurrence is always evaluated first—but there isn't really anything important about right-most derivations that isn't already represented by the notion of a left-most derivation, at least from the viewpoint of this course. For this reason, we won't have any reason to discuss right-most derivations further.

Parse trees

With any derivation of a string by a context-free grammar we may associate a tree, called a *parse tree*, according to the following rules:

- We have one node of the tree for each new occurrence of either a variable, a symbol, or an ε in the derivation, with the root node of the tree corresponds to the start variable. (We only have nodes labelled ε when rules of the form $V \rightarrow \varepsilon$ are applied.)
- Each node corresponding to a symbol or an ε is a leaf node (having no children), while each node corresponding to a variable has one child for each symbol or variable with which it is replaced. The children of each (variable) node are ordered in the same way as the symbols and variables in the rule used to replace that variable.

For example, the derivation (8.3) yields the parse tree illustrated in Figure 8.1.

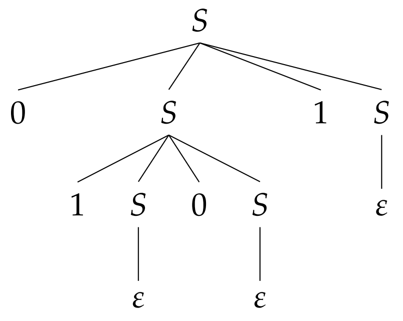

Figure 8.1: The parse tree corresponding to the derivation (8.3) of the string 0101.

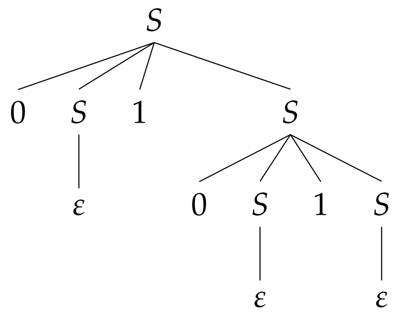

Figure 8.2: A parse tree corresponding to the derivation (8.5) of the string 0101.

There is a one-to-one and onto correspondence between parse trees and left-most derivations, meaning that every parse tree uniquely determines a left-most derivation and each left-most derivation uniquely determines a parse tree.

8.2 Ambiguity

Sometimes a context-free grammar will allow multiple parse trees (or, equivalently, multiple left-most derivations) for some strings in the language that it generates. For example, a different left-most derivation of the string 0101 by the CFG (8.1) from the derivation (8.3) is given by

$$S \Rightarrow 0\,S\,1\,S \Rightarrow 0\,1\,S \Rightarrow 0\,1\,0\,S\,1\,S \Rightarrow 0\,1\,0\,1\,S \Rightarrow 0\,1\,0\,1. \tag{8.5}$$

The parse tree corresponding to this derivation is illustrated in Figure 8.2.

When it is the case, for a given context-free grammar G, that there exists at least one string $w \in \mathrm{L}(G)$ having at least two different parse trees, then the CFG G is said to be *ambiguous*. Note that this is so even if there is just a single string having multiple parse trees—in order to be *unambiguous*, a CFG must have just a single, unique parse tree for each string it generates.

Being unambiguous is generally considered to be a positive attribute of a CFG, and indeed it is a requirement for some applications of context-free grammars.

Designing unambiguous CFGs

In some cases it is possible to come up with an unambiguous context-free grammar that generates the same language as an ambiguous context-free grammar. For example, we can come up with a different context-free grammar for the language

$$\{w \in \{0,1\}^* : |w|_0 = |w|_1\} \tag{8.6}$$

that, unlike the CFG (8.1), is unambiguous. Here is such a CFG:

$$\begin{aligned} S &\rightarrow 0\,X\,1\,S \mid 1\,Y\,0\,S \mid \varepsilon \\ X &\rightarrow 0\,X\,1\,X \mid \varepsilon \\ Y &\rightarrow 1\,Y\,0\,Y \mid \varepsilon \end{aligned} \tag{8.7}$$

We won't take the time to go through a proof that this CFG is unambiguous—but if you think about it for a few moments, it shouldn't be too hard to convince yourself that it is unambiguous. The variable X generates strings having the same number of 0s and 1s, where the number of 1s never exceeds the number of 0s, and the variable Y is similar except the role of the 0s and 1s is reversed. If try to generate a particular string, you'll never have more than one option as to which rule to apply, assuming you restrict your attention to left-most derivations.

Here is another example of how an ambiguous CFG can be modified to make it unambiguous. Let us define an alphabet

$$\Sigma = \{a, b, +, *, (\,,\,)\} \tag{8.8}$$

along with a CFG

$$S \rightarrow S + S \mid S * S \mid (\,S\,) \mid a \mid b \tag{8.9}$$

This grammar generates strings that look like arithmetic expressions in variables a and b, where we allow the operations $*$ and $+$, along with parentheses. For instance, the

$$(a + b) * a + b \tag{8.10}$$

is such an expression, and we can generate it (for instance) as follows:

$$\begin{aligned} S &\Rightarrow S * S \Rightarrow (S) * S \Rightarrow (S + S) * S \Rightarrow (a + S) * S \Rightarrow (a + b) * S \\ &\Rightarrow (a + b) * S + S \Rightarrow (a + b) * a + S \Rightarrow (a + b) * a + b. \end{aligned} \tag{8.11}$$

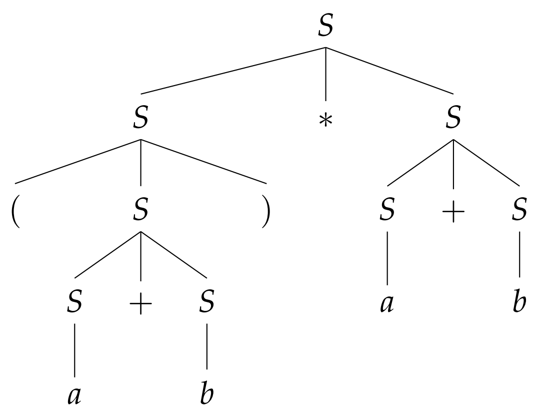

Figure 8.3: Parse tree for $(a+b)*a+b$ corresponding to the derivation (8.11).

This happens to be a left-most derivation, as it is always the left-most variable that is substituted. The parse tree corresponding to this derivation is shown in Figure 8.3.

You can imagine a more complex version of this grammar allowing for other arithmetic operations, variables, and so on, but we will stick to the grammar in (8.9) for the sake of simplicity.

Now, the CFG (8.9) is certainly ambiguous. For instance, a different (left-most) derivation for the same string $(a+b)*a+b$ as before is

$$\begin{aligned} S &\Rightarrow S+S \Rightarrow S*S+S \Rightarrow (S)*S+S \Rightarrow (S+S)*S+S \\ &\Rightarrow (a+S)*S+S \Rightarrow (a+b)*S+S \Rightarrow (a+b)*a+S \\ &\Rightarrow (a+b)*a+b, \end{aligned} \tag{8.12}$$

and the parse tree for this derivation is shown in Figure 8.4. Notice that there is something appealing about the parse tree illustrated in Figure 8.4, which is that it actually carries the meaning of the expression $(a+b)*a+b$, in the sense that the tree structure properly captures the order in which the operations should be applied. In contrast, the first parse tree seems to represent what the expression $(a+b)*a+b$ would evaluate to if we lived in a society where addition was given higher precedence than multiplication.

The ambiguity of the grammar (8.9), along with the fact that parse trees may not represent the meaning of an arithmetic expression in the sense just described, is a problem in some settings. For example, if we were designing a compiler and wanted a part of it to represent arithmetic expressions (presumably allowing much more complicated ones than our grammar from above allows), a CFG along the lines of (8.9) would be completely inadequate.

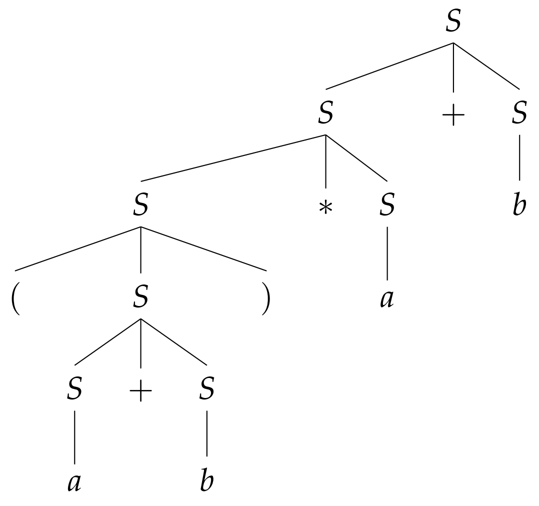

Figure 8.4: Parse tree for $(a + b) * a + b$ corresponding to the derivation (8.12).

We can, however, come up with a new CFG for the same language that is much better—it is unambiguous and it properly captures the meaning of arithmetic expressions. Here it is:

$$\begin{aligned} S &\rightarrow T \mid S + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow I \mid (\, S \,) \\ I &\rightarrow a \mid b \end{aligned} \tag{8.13}$$

For example, the unique parse tree corresponding to the string $(a + b) * a + b$ is as shown in Figure 8.5.

In order to better understand the CFG (8.13), it may help to associate meanings with the different variables. In this CFG, the variable T generates *terms*, the variable F generates *factors*, and the variable I generates *identifiers*. An expression is either a term or a sum of terms, a term is either a factor or a product of factors, and a factor is either an identifier or an entire expression inside of parentheses.

Inherently ambiguous languages

While we have seen that it is sometime possible to come up with an unambiguous CFG that generates the same language as an ambiguous CFG, it is not always possible. There are some context-free languages that can only be generated by ambiguous CFGs. Such languages are called *inherently ambiguous* context-free languages.

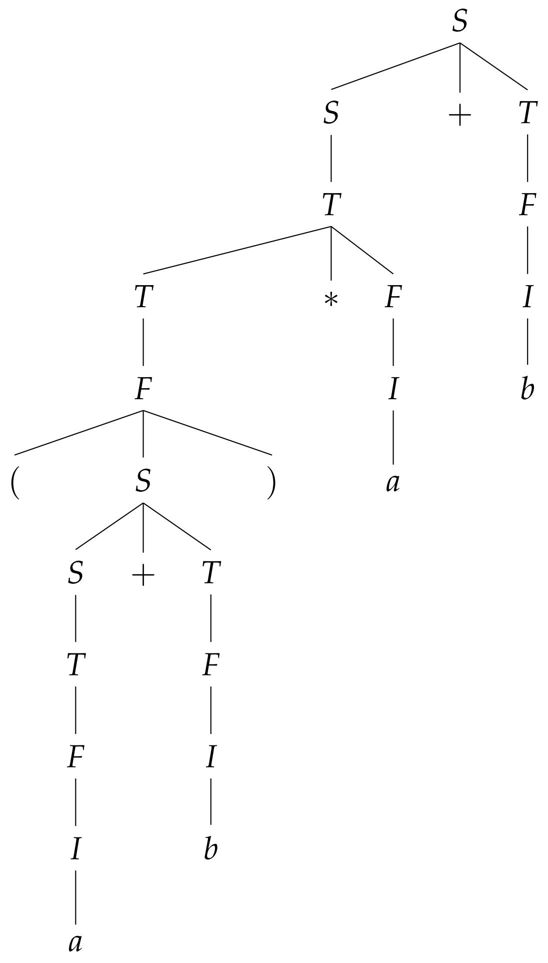

Figure 8.5: Unique parse tree for $(a+b)*a+b$ for the CFG (8.13).

An example of an inherently ambiguous context-free language is this one:

$$\{0^n 1^m 2^k \;:\; n = m \text{ or } m = k\}. \tag{8.14}$$

We will not discuss a proof that this language is inherently ambiguous, but the intuition is that the string $0^n 1^n 2^n$ will always have multiple parse trees for some sufficiently large natural number n.

8.3 Chomsky normal form

Some context-free grammars are strange. For example, the CFG

$$S \to S\,S\,S\,S \;\big|\; \varepsilon \tag{8.15}$$

simply generates the language $\{\varepsilon\}$; but it is obviously ambiguous, and even worse it has infinitely many parse trees (which of course can be arbitrarily large) for the only string ε it generates. While we know we cannot always eliminate ambiguity from CFGs—as some context-free languages are inherently ambiguous—we can at least eliminate the possibility to have infinitely many parse trees for a given string. Perhaps more importantly, for any given CFG G, we can always come up with a new CFG H for which it holds that $\mathrm{L}(H) = \mathrm{L}(G)$, and for which we are guaranteed that every parse tree for a given string $w \in \mathrm{L}(H)$ has the same size and a very simple, binary-tree-like structure.

To be more precise about the specific sort of CFGs and parse trees we're talking about, it is appropriate at this point to define what is called the *Chomsky normal form* for context-free grammars.

Definition 8.1. A context-free grammar G is in *Chomsky normal form* if every rule of G has one of the following three forms:

1. $X \rightarrow YZ$, for variables X, Y, and Z, and where neither Y nor Z is the start variable,
2. $X \rightarrow \sigma$, for a variable X and a symbol σ, or
3. $S \rightarrow \varepsilon$, for S the start variable.

Now, the reason why a CFG in Chomsky normal form is nice is that every parse tree for such a grammar has a simple form: the variable nodes form a binary tree, and for each variable node that doesn't have any variable node children, a single symbol node hangs off. A hypothetical example meant to illustrate the structure we are talking about is given in Figure 8.6. Notice that the start variable always appears exactly once at the root of the tree because it is never allowed on the right-hand side of any rule.

If the rule $S \rightarrow \varepsilon$ is present in a CFG in Chomsky normal form, then we have a special case that doesn't fit exactly into the structure described above. In this case we can have the very simple parse tree shown in Figure 8.7 for ε, and this is the only possible parse tree for this string.

Because of the very special form that a parse tree must take for a CFG G in Chomsky normal form, we have that *every* parse tree for a given string $w \in \mathrm{L}(G)$ must have exactly $2|w| - 1$ variable nodes and $|w|$ leaf nodes (except for the special case $w = \varepsilon$, in which we have one variable node and 1 leaf node). An equivalent statement is that every derivation of a (nonempty) string w by a CFG in Chomsky normal form requires exactly $2|w| - 1$ substitutions.

The following theorem establishes that every context-free language is generated by a CFG in Chomsky normal form.

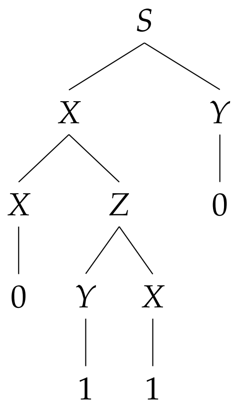

Figure 8.6: A hypothetical example of a parse tree for a CFG in Chomsky normal form.

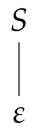

Figure 8.7: The unique parse tree for ε for a CFG in Chomsky normal form, assuming it includes the rule $S \rightarrow \varepsilon$.

Theorem 8.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. There exists a CFG G in Chomsky normal form such that $A = \mathrm{L}(G)$.*

The usual way to prove this theorem is through a construction that converts an arbitrary CFG G into a CFG H in Chomsky normal form for which it holds that $\mathrm{L}(H) = \mathrm{L}(G)$. The conversion is, in fact, quite straightforward—a summary of the steps one may perform to do this conversion for an arbitrary CFG $G = (V, \Sigma, R, S)$ is as follows:

1. Add a new start variable S_0 along with the rule $S_0 \rightarrow S$.

   Doing this will ensure that the start variable S_0 never appears on the right-hand side of any rule.

2. Eliminate ε-rules of the form $X \rightarrow \varepsilon$ and "repair the damage."

   Aside from the special case $S_0 \rightarrow \varepsilon$, there is never any need for rules of the form $X \rightarrow \varepsilon$; you can get the same effect by simply duplicating rules in which X appears on the right-hand side, and directly replacing or not replacing X

with ε in all possible combinations. For example, if we have the CFG

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow (\,S\,)\,S \mid \varepsilon \end{aligned} \tag{8.16}$$

we can easily eliminate the ε-rule $S \rightarrow \varepsilon$ in this way:

$$\begin{aligned} S_0 &\rightarrow S \mid \varepsilon \\ S &\rightarrow (\,S\,)\,S \mid (\,)\,S \mid (\,S\,) \mid (\,) \end{aligned} \tag{8.17}$$

It can get quite messy for larger grammars, and when ε-rules for multiple variables are involved one needs to take care in making the process terminate correctly, but it is always possible to remove all ε-rules (aside from $S_0 \rightarrow \varepsilon$) in this way.

3. Eliminate unit rules, which are rules of the form $X \rightarrow Y$.

   Rules like this are never necessary, and they can be eliminated by simply "hard coding" the substitution $X \rightarrow Y$ into every other (non-unit) rule where X appears on the right-hand side. For instance, carrying on the example from above, we can eliminate the unit rule $S_0 \rightarrow S$ like this:

$$\begin{aligned} S_0 &\rightarrow (\,S\,)\,S \mid (\,)\,S \mid (\,S\,) \mid (\,) \mid \varepsilon \\ S &\rightarrow (\,S\,)\,S \mid (\,)\,S \mid (\,S\,) \mid (\,) \end{aligned} \tag{8.18}$$

4. Introduce a new variable X_σ for each symbol σ.

   Include the rule $X_\sigma \rightarrow \sigma$, and replace every instance of σ appearing on the right-hand side of a rule by X_σ. (Of course you don't have to do this in cases where σ appears all by itself on the right-hand side of a rule.) For the example above, we may use L and R (for left-parenthesis and right-parenthesis) to obtain

$$\begin{aligned} S_0 &\rightarrow L\,S\,R\,S \mid L\,R\,S \mid L\,S\,R \mid L\,R \mid \varepsilon \\ S &\rightarrow L\,S\,R\,S \mid L\,R\,S \mid L\,S\,R \mid L\,R \\ L &\rightarrow (\\ R &\rightarrow) \end{aligned} \tag{8.19}$$

5. Finally, we can split up rules of the form $X \rightarrow Y_1 \cdots Y_m$ using auxiliary variables in a straightforward way.

   For instance, $X \rightarrow Y_1 \cdots Y_m$ can be broken up as

$$\begin{aligned} X &\rightarrow Y_1 X_2 \\ X_2 &\rightarrow Y_2 X_3 \\ &\vdots \\ X_{m-1} &\rightarrow Y_{m-1} Y_m. \end{aligned} \tag{8.20}$$

We need to be sure to use separate auxiliary variables for each rule so that there is no "cross talk" between separate rules. Let's not write this out explicitly for our example because it will be lengthy and hopefully the idea is clear.

The description above is only meant to give you the basic idea of how the construction works and does not constitute a proof of Theorem 8.2. It is possible, however, to be more formal and precise in describing this construction in order to obtain a proper proof of Theorem 8.2.

As you may by now suspect, this conversion of a CFG to Chomsky normal form often produces very large CFGs. The steps are routine but things get messy and it is easy to make a mistake when doing it by hand. I will never ask you to perform this conversion on an actual CFG, but we will make use of the theorem from time to time—when we are proving things about context-free languages it is sometimes extremely helpful to know that we can always assume that a given context-free language is generated by a CFG in Chomsky normal form.

Finally, it must be stressed that the Chomsky normal form says nothing about ambiguity in general—a CFG in Chomsky normal form may or may not be ambiguous, just like we have for arbitrary CFGs. On the other hand, if you start with an unambiguous CFG and perform the conversion described above, the resulting CFG in Chomsky normal form will still be unambiguous.