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 *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:

$$S \to 0S1S \,\big|\, 1S0S \,\big|\, \varepsilon. \tag{8.1}$$

Again we will 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:

$$S \Rightarrow 0S1S \Rightarrow 01S0S1S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101. \tag{8.3}$$

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:

$$0\underline{S}1S \Rightarrow 01\underline{S0S}1S. \tag{8.4}$$

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 does not 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 is not really anything important about right-most derivations that is not already represented by the notion of a left-most derivation, at least from the viewpoint of this course. For this reason, we will not 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:

1. 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 corresponding to the start variable. We only have nodes labeled ε when rules of the form $V \to \varepsilon$ are applied.
2. 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, variable, or ε 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.

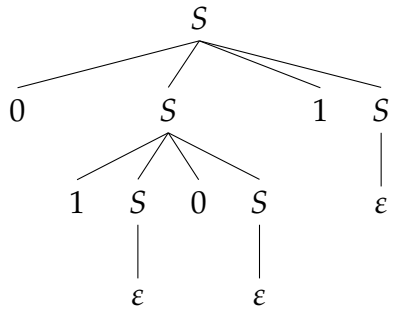

Figure 8.1: The parse tree corresponding to the derivation (8.3) of the string 0101.

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 left-most derivation of the string 0101 by the CFG (8.1) that is different from (8.3) is

$$S \Rightarrow 0S1S \Rightarrow 01S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101. \tag{8.5}$$

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, 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 *every* 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

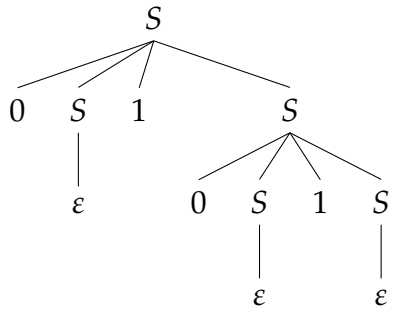

Figure 8.2: The parse tree corresponding to the derivation (8.5) of the string 0101.

L(G) described in (8.2) that, unlike the CFG (8.1), is unambiguous. Here is such a CFG:

$$\begin{aligned} S &\to 0X1S \,\big|\, 1Y0S \,\big|\, \varepsilon \\ X &\to 0X1X \,\big|\, \varepsilon \\ Y &\to 1Y0Y \,\big|\, \varepsilon \end{aligned} \tag{8.6}$$

We will not take the time to go through a proof that this CFG is unambiguous, but if you think about it for a few moments you should be able 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 when you read from left to right, and the variable Y is similar except the role of the 0s and 1s is reversed. If you try to generate a particular string by a left-most derivation with this CFG, you will never have more than one option as to which rule to apply.

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

$$\Sigma = \{a, b, +, *, (,)\} \tag{8.7}$$

along with a CFG

$$S \to S + S \,\big|\, S * S \,\big|\, (\, S\,) \,\big|\, a \,\big|\, b \tag{8.8}$$

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

For example, the string $(a + b) * a + b$ corresponds to such an expression, and one derivation for this string is as follows:

$$\begin{aligned} S \Rightarrow S * S \Rightarrow (S) * S \Rightarrow (S + S) * S \Rightarrow (a + S) * S \Rightarrow (a + b) * S \\ \Rightarrow (a + b) * S + S \Rightarrow (a + b) * a + S \Rightarrow (a + b) * a + b. \end{aligned} \tag{8.9}$$

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

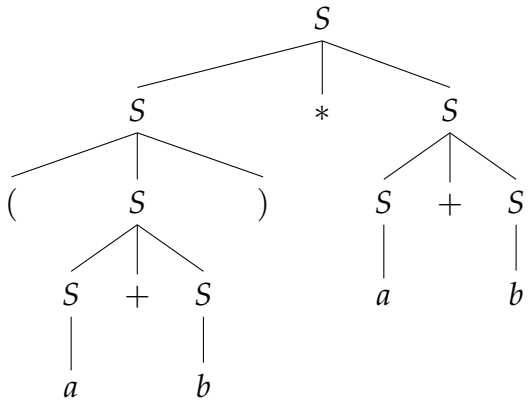

Figure 8.3: Parse tree for $(a+b)*a+b$ corresponding to the derivation (8.9).

Figure 8.3. You can of course 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 (8.8) for the sake of simplicity.

The CFG (8.8) is ambiguous. For instance, a different (left-most) derivation for the same string $(a+b)*a+b$ as before is

$$\begin{aligned} S &\Rightarrow S+S \Rightarrow S*S+S \Rightarrow (S)*S+S \\ &\Rightarrow (S+S)*S+S \Rightarrow (a+S)*S+S \Rightarrow (a+b)*S+S \\ &\Rightarrow (a+b)*a+S \Rightarrow (a+b)*a+b, \end{aligned} \tag{8.10}$$

and the parse tree for this derivation is shown in Figure 8.4.

Notice that the parse tree illustrated in Figure 8.4 is appealing because 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 according to the standard order of precedence for arithmetic operations. In contrast, the parse tree shown in Figure 8.3 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 (8.8), 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 (8.8) would be completely inadequate.

We can, however, come up with a new CFG for the same language that is much better, in the sense that it is unambiguous and properly captures the meaning of

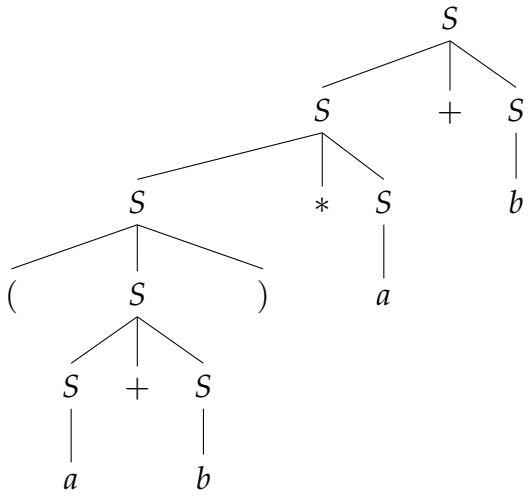

Figure 8.4: Parse tree for $(a+b)*a+b$ corresponding to the derivation (8.10).

arithmetic expressions (given that we give multiplication higher precedence than addition). Here it is:

$$\begin{aligned} S &\to T \mid S+T \\ T &\to F \mid T*F \\ F &\to I \mid (\,S\,) \\ I &\to a \mid b \end{aligned} \tag{8.11}$$

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

To better understand the CFG (8.11), 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. An example of an inherently ambiguous context-free language is this one:

$$\left\{0^n 1^m 2^k \,:\, n=m \text{ or } m=k\right\}. \tag{8.12}$$

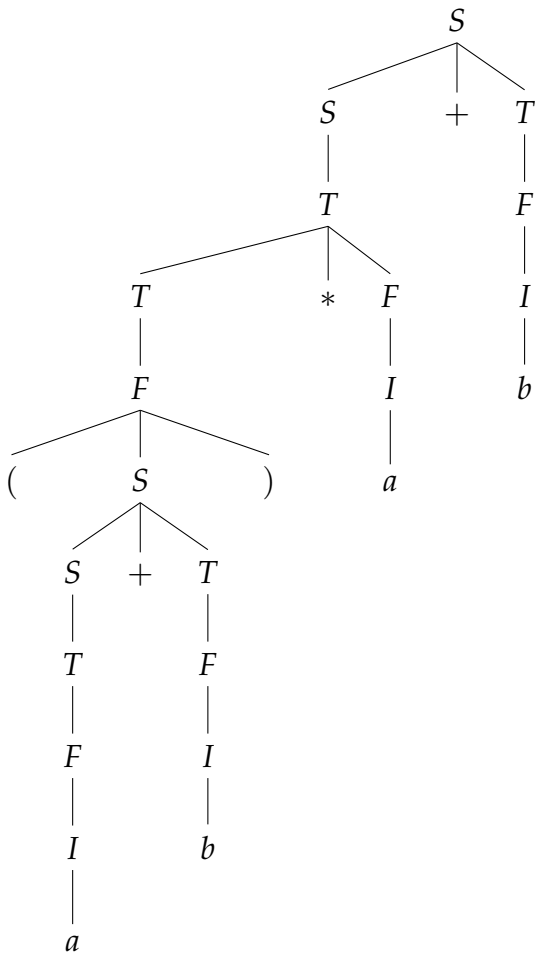

Figure 8.5: Unique parse tree for $(a + b) * a + b$ for the CFG (8.11).

We will not prove that this language is inherently ambiguous, but the intuition is that no matter what CFG you come up with for this language, the string $0^n1^n2^n$ 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

$$S \to SSSS \mid \varepsilon \tag{8.13}$$

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

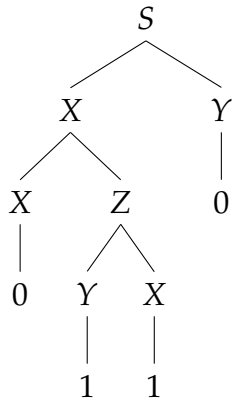

Figure 8.6: A hypothetical example of a parse tree for a CFG in Chomsky normal form.

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 $\mathrm{L}(H) = \mathrm{L}(G)$, 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 are talking about, it is appropriate at this point to define *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. $X \to YZ$, for variables X, Y, and Z, and where neither Y nor Z is the start variable,
2. $X \to a$, for a variable X and a symbol a, or
3. $S \to \varepsilon$, 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 does not have two 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.

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

If the rule $S \to \varepsilon$ is present in a CFG in Chomsky normal form, then we have a special case that does not match 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 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.

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 $\mathrm{L}(H) = \mathrm{L}(G)$. The conversion is, in fact, fairly straightforward—a summary of the steps one may perform to do this conversion for an arbitrary CFG $G = (V, \Sigma, R, S)$ appear below. To illustrate how these steps work, let us start with the following CFG, which generates the balanced parentheses language BAL from the previous lecture:

$$S \to (S)S \mid \varepsilon \tag{8.14}$$

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

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

   Applying this step to the CFG (8.14) yields

   $$\begin{aligned} S_0 &\to S \\ S &\to (S)S \mid \varepsilon \end{aligned} \tag{8.15}$$

2. Introduce a new variable X_a for each symbol $a \in \Sigma$.

   First include the new rule $X_a \to a$. Then, for every other rule in which a appears on the right-hand side, except for the cases when a appears all by itself on the right-hand side, replace each a with X_a.

   Continuing with our example, the CFG (8.15) is transformed into this CFG (where we will use the names L and R rather than the weird-looking variables $X_($ and $X_)$ in the interest of style):

$$\begin{aligned} S_0 &\to S \\ S &\to LSRS \mid \varepsilon \\ L &\to (\\ R &\to) \end{aligned} \tag{8.16}$$

3. Split up rules of the form $X \to Y_1 \cdots Y_m$, whenever $m \geq 3$, using auxiliary variables in a straightforward way.

   In particular, $X \to Y_1 \cdots Y_m$ can be broken up as

$$\begin{aligned} X &\to Y_1 Z_2 \\ Z_2 &\to Y_2 Z_3 \\ &\vdots \\ Z_{m-2} &\to Y_{m-2} Z_{m-1} \\ Z_{m-1} &\to Y_{m-1} Y_m \end{aligned} \tag{8.17}$$

   Note that we must use separate auxiliary variables for each rule so that there is no "cross talk" between different rules—so do not reuse the same auxiliary variables to break up multiple rules.

   Transforming the CFG (8.16) in this way results in the following CFG:

$$\begin{aligned} S_0 &\to S \\ S &\to LZ_2 \mid \varepsilon \\ Z_2 &\to SZ_3 \\ Z_3 &\to RS \\ L &\to (\\ R &\to) \end{aligned} \tag{8.18}$$

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

Aside from the special case $S_0 \to \varepsilon$, there is never any need for rules of the form $X \to \varepsilon$; 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 each possible combination. You might introduce new ε-rules in this way, but they can be handled recursively—and any time a new ε-rule is generated that was already eliminated, it is not added back in.

Transforming the CFG (8.18) in this way results in the following CFG:

$$\begin{aligned} S_0 &\to S \mid \varepsilon \\ S &\to LZ_2 \\ Z_2 &\to SZ_3 \mid Z_3 \\ Z_3 &\to RS \mid R \\ L &\to (\\ R &\to) \end{aligned} \tag{8.19}$$

Note that we do end up with the ε-rule $S_0 \to \varepsilon$, but we do not eliminate this one because $S_0 \to \varepsilon$ is the special case that we allow as an ε-rule.

5. Eliminate unit rules, which are rules of the form $X \to Y$.

   Rules like this are never necessary, and they can be eliminated provided that we also include the rule $X \to w$ in the CFG whenever $Y \to w$ appears as a rule. If you obtain a new unit rule that was already eliminated (or is the unit rule currently being eliminated), it is not added back in.

   Transforming the CFG (8.19) in this way results in the following CFG:

$$\begin{aligned} S_0 &\to LZ_2 \mid \varepsilon \\ S &\to LZ_2 \\ Z_2 &\to SZ_3 \mid RS \mid) \\ Z_3 &\to RS \mid) \\ L &\to (\\ R &\to) \end{aligned} \tag{8.20}$$

   At this point we are finished; this context-free grammar is in Chomsky normal form.

The description above is only meant to give you the basic idea of how the construction works and does not constitute a formal 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.

We will make use of the theorem from time to time. In particular, 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.