Lecture 9

Parse Trees and Chomsky Normal Form

The goal in this lecture will be to investigate and reduce the amount of ambiguity and redundancy present in context-free grammars. We will define parse trees, left-most derivations, ambiguity, and Chomsky normal form. The latter is a fairly "clean" format for context-free grammars, which we will use when proving structural properties of context-free grammars and languages.

9.1 Parse trees and left-most derivations

Consider the context-free grammar $S \to (S)|SS|\epsilon$ over the alphabet $\{(,)\}$, which we've seen generates the strings of balanced parentheses. Let's call this grammar G. Then for a given string in $L(G)$, such as $w = ()()$, we can write a derivation of w in G, which is a sequence of strings $z_1, z_2, \ldots, z_m$ such that $z_1 = S$, $z_m = w$, and $z_i \Rightarrow_G z_{i+1}$ for $i = 1, 2, \ldots, m-1$. For example, we can write the derivation

$$S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G (S)(S) \Rightarrow_G ()(S) \Rightarrow_G ()().$$

However, this derivation is not unique; we can also write

$$S \Rightarrow_G SS \Rightarrow_G S(S) \Rightarrow_G (S)(S) \Rightarrow_G ()(S) \Rightarrow_G ()().$$

Although this derivation seems essentially identical to the one above, it doesn't have the same sequence of strings z_i. The difference is that in the first derivation, when we got to the string SS, we replaced the first S by (S) before replacing the second by (S), and in the second derivation, we made the same substitutions in the opposite order.

Although we got two different derivations, the difference between them seems immaterial: the same variables got substituted by the same strings, just in different order. To formalize this feeling that the difference is immaterial, we define a notion called a *parse tree*. The parse tree for a derivation is a tree whose root is the start variable S. Each node of the tree is labeled by either a variable, an alphabet symbol, or ϵ. A node labeled by a variable has children, and the labels of the child nodes (from left to right) form the string by which that variable was substituted in the derivation. A node labeled by an alphabet symbol or by ϵ is a leaf and has no children. We give an example in Figure 9.1. Observe that the leaves of the parse tree, when read from left to right, form the final string in the derivation.

The two derivations mentioned above feel equivalent because they have the same parse tree: each variable in the derivation gets substituted by the same string in both derivations. The only thing that changed between them is the order in which we wrote down the intermediate steps of the derivation.

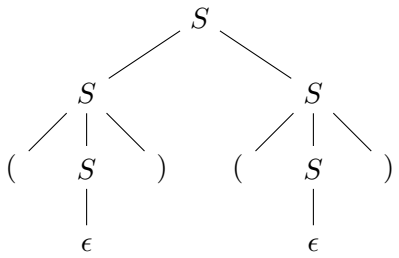

Figure 9.1: The parse tree for the derivation $S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G (S)(S) \Rightarrow_G ()(S) \Rightarrow_G ()()$.

Indeed, for a given parse tree, we can write down many different derivations, which all correspond to that parse tree and which differ only in the order in which the variable substitutions are implemented. Since all these derivations are equivalent, we can choose to write any of them. We will generally choose to write what's called a *left-most derivation*, in which the left-most variable gets substituted at each step. For example, the left-most derivation for the parse tree in Figure 9.1 is

$$S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()().$$

The choice of using the left-most derivation is arbitrary; we could have used the right-most derivation instead, for example (where the right-most variable would be substituted at each step).

Each parse tree has a unique left-most derivation, and each left-most derivation (for a given string in a given CFG) has a unique parse tree. Since writing down left-most derivations is often simpler (or at least more concise) than drawing out a parse tree, we will often use left-most derivations when we wish to analyze a parse tree.

9.2 Ambiguity

Sometimes, a string can be generated by a CFG in multiple non-equivalent ways: there may be multiple derivations for the string that correspond to different parse trees (equivalently, different left-most derivations). A simple example of this arises for the same grammar in the previous section: although we had a left-most derivation

$$S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()()$$

for the string $()()$, we can have another left-most derivation for the same string:

$$S \Rightarrow_G SS \Rightarrow_G SSS \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()().$$

What happened here is that after we reached the string SS, we substituted the left S with another SS, and then we used the rule $S \to \epsilon$ on the leftmost S again. The gave us $SS \Rightarrow_G SSS \Rightarrow_G SS$ through a left-most derivation. By repeating this, we can generate infinitely many different left-most derivations for the same string $()()$, and hence infinitely many different parse trees. We draw the parse tree for the new derivation above in Figure 9.2.

This new, non-equivalent derivation seems wasteful: it wastes space by generating the string $()()$ in a longer way than is necessary. However, it is allowed by our grammar. We could try to eliminate such waste by modifying our grammar. In particular, a variable S that can become both SS and ϵ leads to infinitely many parse trees for any given string in the language.

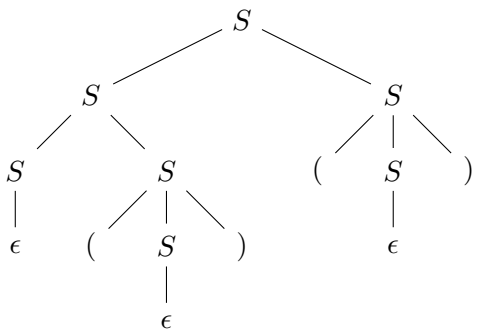

Figure 9.2: The parse tree for the derivation $S \Rightarrow_G SS \Rightarrow_G SSS \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()()$.

A different CFG for the same language of balanced parentheses is $S \to (S)S|\epsilon$. One can show that this one still generates the same language, but that every string in the language has exactly one parse tree (equivalently, exactly one left-most derivation).

We say that a CFG G is *ambiguous* if there is some string $w \in L(G)$ such that there are two different left-most derivations for w in G. That is, we call the grammar ambiguous if there is even a single string generated by G that has more than one parse tree in G. Otherwise, we say G is unambiguous.

Let's now argue that $S \to (S)S|\epsilon$ is an unambiguous context-free grammar for the language B of balanced parentheses. Technically, we first need to prove that this grammar indeed generates exactly the strings of balanced brackets, but we will omit this proof.

Let's proceed to proving that this grammar is unambiguous. Suppose by contradiction that it was ambiguous; this means there is a string of balanced parentheses that has at least two different left-most derivations. Let w be the shortest such string. If $w = \epsilon$, then the only valid derivation of it is $S \Rightarrow_G \epsilon$, since if we use the rule $S \Rightarrow_G (S)S$ we cannot get the empty string. Since w is not the empty string, any derivation of it must start with $S \Rightarrow (S)S$, so its two different left-most derivations start this way. Suppose the first left-most derivation of w has the form $S \Rightarrow_G (S)S \Rightarrow_G^* (w_1)w_2$, and that the second has the form $S \Rightarrow_G (S)S \Rightarrow_G^* (w_3)w_4$. Then $(w_1)w_2 = w = (w_3)w_4$, which necessarily means that $w_1 = w_3$ and $w_2 = w_4$. This means that in the expression $(S)S$, the first S eventually turns into w_1 in both derivations, and the second eventually turns into w_2 in both derivations. Since these left-most derivations are not the same, either the parse tree that generates w_1 from S is different in the two derivations, or else the one that generates w_2 from S is different in the two derivations. However, since w_1 and w_2 are shorter than w, this contradicts our choice of w as the shortest string for which such ambiguity is present.

9.2.1 Can ambiguity always be eliminated?

Given that we managed to modify our context-free grammar $S \to SS|(S)|\epsilon$ to make it the non-ambiguous grammar $S \to (S)S|\epsilon$ while preserving the language it generates, one might wonder whether this is always possible: does every context-free language have an unambiguous context-free grammar?

The answer turns out to be no. Consider the following grammar over the alphabet $\{0, 1, 2\}$.

$$\begin{aligned} S &\to XY \mid WZ \\ X &\to 0X1 \mid \epsilon \\ Y &\to 2Y \mid \epsilon \\ W &\to 0W \mid \epsilon \\ Z &\to 1Z2 \mid \epsilon. \end{aligned}$$

It's not hard to see that X generates all strings of the form 0^n1^n, Y generates all strings of the form 2^m, W generates all strings of the form 0^m, and Z generates all strings of the form 1^n2^n, so S must generate all strings of the form either $0^n1^n2^m$ or $0^m1^n2^n$. If we denote this CFG by G, then

$$L(G) = \{0^a1^b2^c : a, b, c \in \mathbb{N}, \text{ and either } a = b \text{ or } b = c\}.$$

Since we've constructed a context-free grammar for this language, the language is a CFL. However, it turns out that this CFL cannot be generated by an unambiguous CFG. The proof of this fact is a bit tricky, though we might get to it in a few lectures from now, once we've established more properties of context-free grammars and languages.

9.3 Chomsky Normal Form

Although not all ambiguity can be eliminated from context-free grammars, we can at least try to eliminate as much as possible. For example, the context-free grammar $S \to SS|SSS|\epsilon$ only generates the string ϵ, but it does so in infinitely many ways with a lot of needless redundancy.

It turns out that each context-free grammar can be converted into a CFG in "Chomsky normal form", which is a special form for a CFG designed to make it nice (and to eliminate needless ambiguities). As we will see, for a CFG in Chomsky normal form, although a string w might have more than one parse tree, it can only have finitely many different parse trees.

Definition 9.1. *A context-free grammar $G = (V, \Sigma, R, S)$ is said to be in* Chomsky normal form *if every rule in R has one of the following forms:*

1. *$X \to YZ$ where $X, Y, Z \in V$ and neither Y nor Z is the start variable (but we allow Y or Z to be equal to X),*
2. *$X \to a$ where X is a variable and $a \in \Sigma$ is an alphabet symbol, or*
3. *$S \to \epsilon$ where S is the start variable.*

A CFG in Chomsky normal form (CNF) is nice in several ways: first, the empty string ϵ can only occur in a single rule, $S \to \epsilon$ (or else it can not occur at all). Further, after we transition from the start symbol using a rule $S \to XY$, we can never get back to the start symbol from either X or Y, and hence ϵ will never occur in the parse tree. This banishment of ϵ, allowing it to occur only to in the parse tree for the string ϵ itself, is one of the main nice properties of Chomsky normal form, and eliminates a lot of ambiguity.

The parse tree for a string with respect to a CFG in Chomsky normal form is always either the trivial parse tree that connects S directly to ϵ (generating the empty string ϵ), or else it is a binary tree of variables, with each internal node of the tree having either (a) exactly two children, which are labeled by variables and are also internal nodes of the tree, or (b) exactly one child, which is

Figure 9.3: The only parse tree for ϵ in any CFG that is in Chomsky normal form and which can generate ϵ.

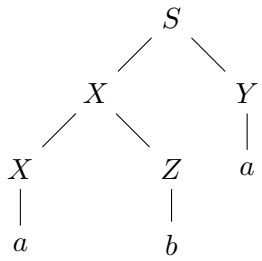

Figure 9.4: An example of a parse tree for the string *aba* in some CFG in Chomsky normal form. Note that each node in the graph has either one or two children, and it has one child if and only if that child is a leaf.

a leaf and is labeled by an alphabet symbol. We draw these two possibilities in Figure 9.3 and Figure 9.4.

We will now argue that every context-free language has a CFG in Chomsky normal form.

Theorem 9.2. *Let A be a context-free language. Then there is a context-free grammar G such that G is in Chomsky normal form and such that $L(G) = A$.*

We won't formally prove this theorem, but we will give an example of how to convert a CFG into Chomsky normal form (the formal proof is not hard, just tedious). The example will be for the grammar $S \to (S)S|\epsilon$ that we've seen. The conversion to Chomsky normal form will be in several steps.

First, we add a new start variable S_0, together with the rule $S_0 \to S$ (as well as all the previous rules). This will ensure that we can never return to the start variable: S_0 does not occur on the right hand side of any rule.

Second, we will introduce a variable for each alphabet symbol. In this case, we will introduce variables L and R, together with rules $L \to ($ and $R \to)$. We can then replace any occurrences of alphabet symbols in the old rules with the variables L and R; we get the grammar

$$\begin{aligned} S_0 &\to S \\ S &\to LSRS \mid \epsilon \\ L &\to (\\ R &\to). \end{aligned}$$

Third, will break up rules in which the right hand side has many variables into intermediate rules using new variables; for example, the rule $X_0 \to X_1X_2 \dots X_n$ will become the set of rules $X_0 \to X_1Z_1$, $Z_1 \to X_2Z_2$, $Z_2 \to X_3Z_3$, ..., $Z_{n-2} \to X_{n-1}X_n$. This will ensure the right hand side

of rules all have at most two variables. Applied to our example grammar, we get

$$\begin{aligned} S_0 &\to S \\ S &\to LZ_1 \mid \epsilon \\ Z_1 &\to SZ_2 \\ Z_2 &\to RS \\ L &\to (\\ R &\to). \end{aligned}$$

Fourth, we eliminate rules with ϵ on the right hand side except for $S_0 \to \epsilon$. To do so, whenever we have a rule $X \to \epsilon$ for some variable X, we delete that rule but add a new rule for each rule where X appeared on the right hand side. For example, if we had the rule $Y \to XZ$ and also $X \to \epsilon$, we will delete the latter, and add the new rule $Y \to Z$, which is what we would get by substituting ϵ for X in the rule $Y \to XZ$. Note that deleting X from the right hand side of rules can introduce new rules with ϵ on the right hand side. We could deal with this recursively, by eliminating those new ϵ-rules as well. To make sure we don't encounter a loop – a situation where we're reintroducing a rule $X \to \epsilon$ that we've already eliminated – we will not introduce a new rule $X \to \epsilon$ if we've already eliminated that same rule. It turns out that this is a safe thing to do (we won't prove that here).

Applying this procedure to the above grammar, we get

$$\begin{aligned} S_0 &\to S \mid \epsilon \\ S &\to LZ_1 \\ Z_1 &\to SZ_2 \mid Z_2 \\ Z_2 &\to RS \mid R \\ L &\to (\\ R &\to). \end{aligned}$$

Finally, we eliminate rules of the form $X \to Y$ with exactly one variable on the right-hand side. We can do this by adding new rules for X for everything Y can turn into; so if we had $X \to Y$ and $Y \to WZ$ before, we will delete the former and add the rule $X \to WZ$ instead. We keep doing this until there are no rules with a single variable on the right hand side. Applying this to the above grammar gives

$$\begin{aligned} S_0 &\to LZ_1 \mid \epsilon \\ S &\to LZ_1 \\ Z_1 &\to SZ_2 \mid RS \mid) \\ Z_2 &\to RS \mid) \\ L &\to (\\ R &\to). \end{aligned}$$

At this point, the grammar is in Chomsky normal form, but still generates the same language as the original context-free grammar. This strategy can be generalized to turn any CFG into Chomsky normal form, though we won't formally prove this.