MA513: Formal Languages and Automata Theory
Topic: Context-free Grammars (CFG)
Lecture Number 19 Date: September 26, 2011

1 Removing Ambiguity From Grammars

Example 1: Consider the following grammar for expressions:

$$\boxed{\begin{array}{l} E \rightarrow I/E+E/E*E/(E) \\ I \rightarrow a/b/Ia/Ib/I0/I1 \end{array}}$$

Example 2: Consider the sentential form $E+E*E$. It has two derivations from E (see grammar in Example 1):

1. $E \Rightarrow E+E \Rightarrow E+E*E$.
2. $E \Rightarrow E*E \Rightarrow E+E*E$.

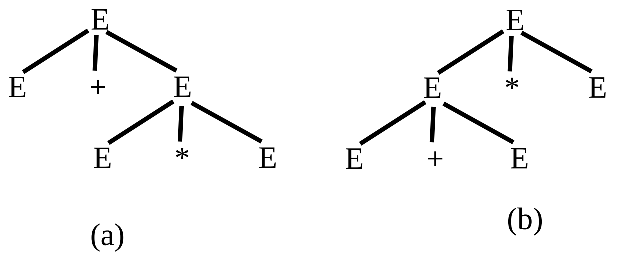

Figure 1: Two parse trees with the same yield $E+E*E$

Example 2 says that the grammar in example 1 is ambiguous grammar. There are two causes of ambiguity in grammar in example 1.

1. The precedence of operators is not respected. While Figure 1(a) properly groups * before + operator, Figure 1(b) is also a valid perse tree and groups the + ahead of the *. We need to force only the structure of Figure 1(a) to be legal in an unambiguous grammar.

2. A sequence of identical operators can group either from the left or from the right. For example, if *'s in Figure 1 were replaced by +'s, we would see two different parse trees for the string $E+E+E$. Since addition and multiplication are associative, it does not matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left, so the structure of Figure 1(b) is the only correct grouping of two + signs.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of **binding strength**. Specially:

1. A **factor** is an expression that can not be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A **term** is an expression that cannot be broken by the + operator. In our example, where + and * are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ can be **broken** if we use left associativity and place $a1*$ to its left. That is, $a1 * a * b$ is grouped $(a1 * a) * b$, which breaks apart the $a * b$. However, placing additive term, such as $a1+$ to its left or $+a1$ to its right cannot break $a*b$. The proper grouping of $a1 + a * b$ is $a1 + (a * b)$, and the proper grouping of $a * b + a1$ is (a*b)+a1.
3. An **expression** will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression for our example is a sum of one or more terms.

From the above discussion, we can write an unambiguous expression grammar as follows:

Table 1: An unambiguous expression grammar

$E \rightarrow T/E + T$
$T \rightarrow F/T * F$
$F \rightarrow I/(E)$
$I \rightarrow a/b/Ia/Ib/I0/I1$

Example 3: The grammar in Table 1 allows only one parse tree for the string $a + a * a$; it is shown in the Figure 2.

The fact that the grammar in Table 1 is unambiguous may be far from obvious. Here are the key observations that explain why no string in the language can have two different parse trees.

- Any string derived from T, a term, must be a sequence of one or more factors, connected by *'s. A factor is either a single identifier or any parenthesized expression.

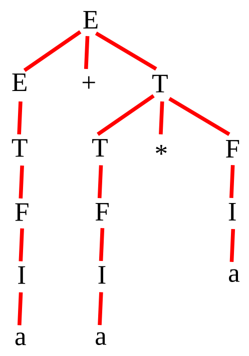

Figure 2: The sole parse tree for $a + a * a$

- Because of the form of the two rules for T, the only parse tree for a sequence of factors is the one that breaks $f_1 * f_2 * \ldots * f_n$, for $n > 1$ into a term $f_1 * f_2 * \ldots * f_{n-1}$ and a factor f_n. The reason is that F can not derive expression like $f_{n-1} * f_n$ without introducing parentheses around them. Thus, it is not possible that when using the production $T \rightarrow T * F$, the F derives anything but the last of the factor. That is, the parse tree for a term can only look like Figure 3.

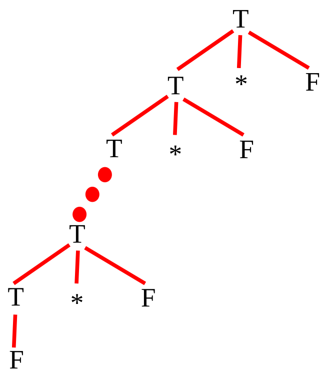

Figure 3: The form of all parse trees for a term

- Likewise an expression is a sequence of terms connected by $+$. When we use the production $E \rightarrow E + T$ to derive $t_1 + t_2 + \ldots, t_n$, then T must derive only t_n, and the E in the body derives $t_1 + t_2 + \ldots t_{n-1}$. The reason, again, is that T can not derive the sum of two or more terms without putting parentheses around them.

2 Leftmost Derivations as a Way to Express Ambiguity

While derivations are not unique, even if the grammar is unambiguous, it turns out that, in an unambiguous grammar, leftmost derivation will be unique, and rightmost derivations will be unique. We shall consider leftmost derivations only, and state the result for rightmost derivations.

As an example, see the two parse trees in Figure 4 that each yield $a + a * a$ (see the grammar in Example 1). If we construct leftmost derivations from them we get the following leftmost derivations from trees (a) and (b), respectively.

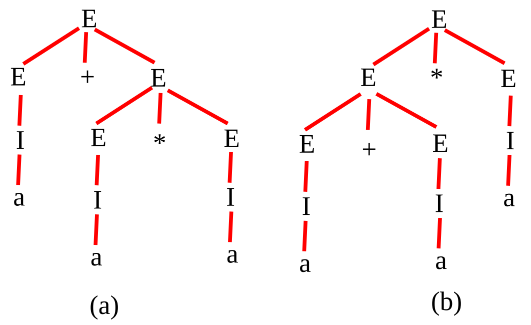

Figure 4: Trees with yield $a + a * a$, demonstrating the ambiguity of expression grammar (see Example 1)

a) $E \Rightarrow_{lm} E + E \Rightarrow_{lm} I + E \Rightarrow_{lm} a + E \Rightarrow_{lm} a + E * E \Rightarrow_{lm} a + I * E \Rightarrow_{lm} a + a * E \Rightarrow_{lm} a + a * I \Rightarrow_{lm} a + a * a$

b) $E \Rightarrow_{lm} E * E \Rightarrow_{lm} E + E * E \Rightarrow_{lm} I + E * E \Rightarrow_{lm} a + E * E \Rightarrow_{lm} a + I * E \Rightarrow_{lm} a + a * E \Rightarrow_{lm} a + a * I \Rightarrow_{lm} a + a * a$

Note that these two leftmost derivations differ. This example does not prove the theorem, demonstrates how the differences in the trees force different steps to be taken in the leftmost derivation.

Theorem: For each grammar $G = (V, T, R, S)$ and string $w \in T^*$, w has two distinct parse trees **if and only if** w has two distinct leftmost derivations from S.

• For proof see Hopcroft, Motwani and Ullman book.

3 Inherent Ambiguity

A context-free language L is said to be **inherently ambiguous** if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language. For example, the language of expressions generated by the grammar in Example 1 is actually unambiguous. Even though that grammar is ambiguous, there is another grammar for the same language that is unambiguous (see the grammar in Table 1).

We shall not prove that there are inherently ambiguous languages. Rather we shall discuss one example of a language that can be proved inherently ambiguous, and we shall explain intuitively why every grammar for the language must be ambiguous. The language L in question is:

$$L = \{a^n b^n c^m d^m | n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n | n \geq 1, m \geq 1\}$$

L is a context-free language. The obvious grammar for L is shown in the Table 2. It uses separate set of rules to generate the two kind of strings in L.

Table 2: A grammar for an inherently ambiguous language

$S \rightarrow AB/C$
$A \rightarrow aAb/ab$
$B \rightarrow cBd/cd$
$C \rightarrow aCd/aDd$
$D \rightarrow bDc/bc$

This grammar is ambiguous. For example, the string *aabbccdd* has the two leftmost derivations:

1. $S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd.$

2. $S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aaDdd \Rightarrow_{lm} aabDdd \Rightarrow_{lm} aabbccdd$

and the two parse trees shown in Figure 5.

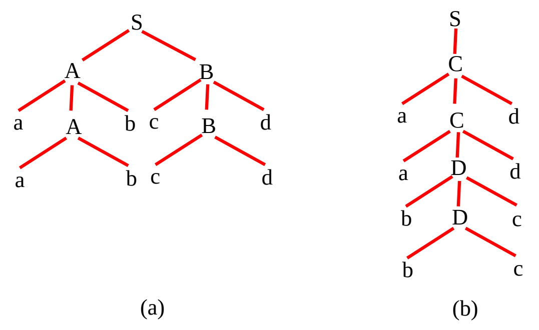

Figure 5: Two parse trees for aabbccdd

The proof that all grammars for L must be ambiguous is complex. However the essence is as follows. We need to argue that all but a finite number of strings whose counts of the four symbols a, b, c and d, are all equal must be generated in two different ways: one in which the a's and b's are generated to be equal and the c's and d's are generated to be equal, and a second way, where the a's and d's are generated to be equal and likewise the b's and c's.