Online library: digital copies

Copyright Notice

Staff and students of the University of London are reminded that copyright subsists in this extract and the work from which it was taken. This Digital Copy has been made under the terms of a CLA licence which allows Course Users to:

- access and download a copy;
- print out a copy.

This Digital Copy and any digital or printed copy supplied under the terms of this Licence are for use in connection with this Course of Study. They should not be downloaded or printed by anyone other than a student enrolled on the named course. A student enrolled on this course may retain such copies after the end of the course, but strictly for their own personal use only.

All copies (including electronic copies) shall include this Copyright Notice and shall be destroyed and/or deleted if and when required by the University.

Except as provided for by copyright law, no further copying, storage or distribution (including by e-mail) is permitted without the consent of the copyright holder.

The author (which term includes artists and other visual creators) has moral rights in the work and neither staff nor students may cause, or permit, the distortion, mutilation or other modification of the work, or any other derogatory treatment of it, which would be prejudicial to the honour or reputation of the author.

Name of Designated Person authorising scanning:

Sandra Tury, Associate Director: Online Library Services, University of London Worldwide

Course of Study:

Extract title:

Title author:

Name of Publisher:

Publication year, Volume, Issue:

Page extent:

Source title:

ISBN/ISSN:

UNIVERSITY OF LONDON

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT PROF (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)> ]>
```

Figure 5.16: A DTD for courses

5.4 Ambiguity in Grammars and Languages

As we have seen, applications of CFG's often rely on the grammar to provide the structure of files. For instance, we saw in Section 5.3 how grammars can be used to put structure on programs and documents. The tacit assumption was that a grammar uniquely determines a structure for each string in its language. However, we shall see that not every grammar does provide unique structures.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

5.4.1 Ambiguous Grammars

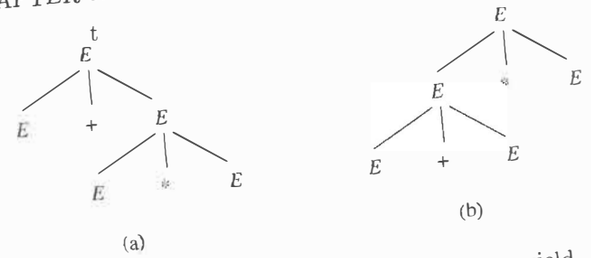
Let us return to our running example: the expression grammar of Fig. 5.2. This grammar lets us generate expressions with any sequence of $*$ and $+$ operators, and the productions $E \to E + E \mid E * E$ allow us to generate these expressions in any order we choose.

Example 5.25: For instance, consider the sentential form $E + E * E$. It has two derivations from E:

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

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$. Figure 5.17 shows the two parse trees, which we should note are distinct trees.

The difference between these two derivations is significant. As far as the structure of the expressions is concerned, derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression, while derivation (2) adds the first two expressions and multiplies the result by the third. In more concrete terms, the first derivation suggests that $1 + 2 * 3$



(a) (b)

Figure 5.17: Two parse trees with the same yield

should be grouped $1 + (2 * 3) = 7$, while the second derivation suggests the same expression should be grouped $(1 + 2) * 3 = 9$. Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions.

Since the grammar of Fig. 5.2 gives two different structures to any string of terminals that is derived by replacing the three expressions in $E + E * E$ by identifiers, we see that this grammar is not a good one for providing unique structure. In particular, while it can give strings the correct grouping as arithmetic expressions, it also gives them incorrect groupings. To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings. □

On the other hand, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar. The following is an example.

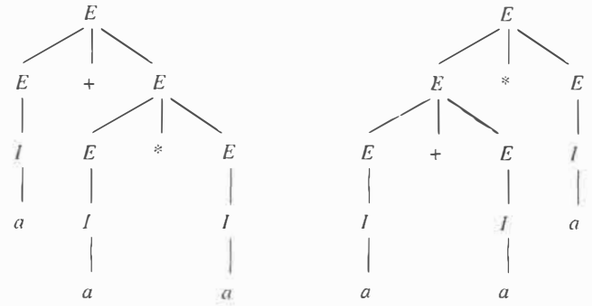
Example 5.26: Using the same expression grammar, we find that the string $a + b$ has many different derivations. Two examples are:

1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

However, there is no real difference between the structures provided by these derivations; they each say that a and b are identifiers, and that their values are to be added. In fact, both of these derivations produce the same parse tree if the construction of Theorems 5.18 and 5.12 are applied. □

The two examples above suggest that it is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees. Thus, we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

For instance, Example 5.25 almost demonstrated the ambiguity of the grammar of Fig. 5.2. We have only to show that the trees of Fig. 5.17 can be completed to have terminal yields. Figure 5.18 is an example of that completion.



(a) (b)

Figure 5.18: Trees with yield $a + a * a$, demonstrating the ambiguity of our expression grammar

5.4.2 Removing Ambiguity From Grammars

In an ideal world, we would be able to give you an algorithm to remove ambiguity from CFG's, much as we were able to show an algorithm in Section 4.4 to remove unnecessary states of a finite automaton. However, the surprising fact is, as we shall show in Section 9.5.2, that there is no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place. Moreover, we shall see in Section 5.4.4 that there are context-free languages that have nothing but ambiguous CFG's; for these languages, removal of ambiguity is impossible.

Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity. The problem with the expression grammar of Fig. 5.2 is typical, and we shall explore the elimination of its ambiguity as an important illustration.

First, let us note that there are two causes of ambiguity in the grammar of Fig. 5.2:

1. The precedence of operators is not respected. While Fig. 5.17(a) properly groups the $*$ before the $+$ operator, Fig 5.17(b) is also a valid parse tree and groups the $+$ ahead of the $*$. We need to force only the structure of Fig. 5.17(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 the $*$'s in Fig. 5.17 were replaced by $+$'s, we would see two different parse trees for the string $E + E + E$. Since addition and multiplication are associative, it doesn't 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 Fig. 5.17(b) is the only correct grouping of two $+$-signs.

Ambiguity Resolution in YACC

If the expression grammar we have been using is ambiguous, we might wonder whether the sample YACC program of Fig. 5.11 is realistic. True, the underlying grammar is ambiguous, but much of the power of the YACC parser-generator comes from providing the user with simple mechanisms for resolving most of the common causes of ambiguity. For the expression grammar, it is sufficient to insist that:

a) $*$ takes precedence over $+$. That is, $*$'s must be grouped before adjacent $+$'s on either side. This rule tells us to use derivation (1) in Example 5.25, rather than derivation (2)e

b) Both $*$ and $+$ are left-associative. That is, group sequences of expressions, all of which are connected by $*$, from the left, and do the same for sequences connected by $+$.

YACC allows us to state the precedence of operators by listing them in order, from lowest to highest precedence. Technically, the precedence of an operator applies to the use of any production of which that operator is the rightmost terminal in the body. We can also declare operators to be left- or right-associative with the keywords `%left` and `%right`. For instance, to declare that $+$ and $*$ were both left associative, with $*$ taking precedence over $+$, we would put ahead of the grammar of Fig. 5.11 the statements:

```
%left '+'
%left '*'
```

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." Specifically:

1. A *factor* is an expression that cannot 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 an 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.

$$
\begin{array}{lcl}
I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
F & \rightarrow & I \mid (E) \\
T & \rightarrow & F \mid T * F \\
E & \rightarrow & T \mid E + T
\end{array}
$$

Figure 5.19: An unambiguous expression grammar

Example 5.27: Figure 5.19 shows an unambiguous grammar that generates the same language as the grammar of Fig. 5.2. Think of F, T, and E as the variables whose languages are the factors, terms, and expressions, as defined above. For instance, this grammar allows only one parse tree for the string $a + a * a$; it is shown in Fig. 5.20.

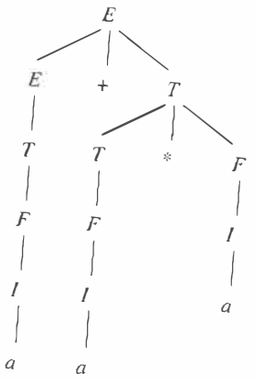


Figure 5.20: The sole parse tree for $a + a * a$

The fact that this grammar 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, as we have defined it, and as follows from the productions for F in Fig. 5.19, is either a single identifier or any parenthesized expression.

- Because of the form of the two productions for T, the only parse tree for a sequence of factors is the one that breaks $f_1 * f_2 * \cdots * f_n$, for $n > 1$ into a term $f_1 * f_2 * \cdots * f_{n-1}$ and a factor f_n. The reason is that F cannot derive expressions like $f_{n-1} * f_n$ without introducing parentheses around them. Thus, it is not possible that when using the production $T \to T * F$, the F derives anything but the last of the factors. That is, the parse tree for a term can only look like Fig. 5.21.

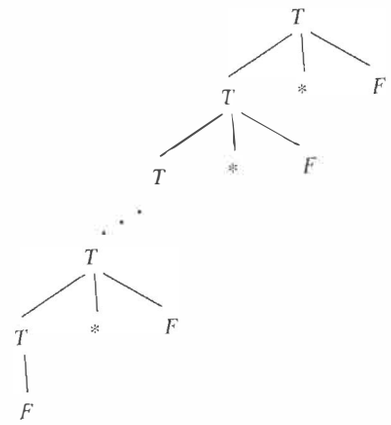


Figure 5.21: The form of all parse trees for a term

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

□

5.4.3 Leftmost Derivations as a Way to Express Ambiguity

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

Example 5.28 As an example, notice the two parse trees of Fig. 5.18 that each yield $E + E * E$. If we construct leftmost derivations from them we get the following leftmost derivations from trees (a) and (b), respectively:

a) $E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} I+E \underset{lm}{\Rightarrow} a+E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*a$

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

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

Theorem 5.29 For each grammar $G = (V, T, P, S)$ and string w in T^*, w has two distinct parse trees if and only if w has two distinct leftmost derivations from S.

PROOF: (Only-if) If we examine the construction of a leftmost derivation from a parse tree in the proof of Theorem 5.14, we see that wherever the two parse trees first have a node at which different productions are used, the leftmost derivations constructed will also use different productions and thus be different derivations.

(If) While we have not previously given a direct construction of a parse tree from a leftmost derivation, the idea is not hard. Start constructing a tree with only the root, labeled S. Examine the derivation one step at a time. At each step, a variable will be replaced, and this variable will correspond to the leftmost node in the tree being constructed that has no children but that has a variable as its label. From the production used at this step of the leftmost derivation, determine what the children of this node should be. If there are two distinct derivations, then at the first step where the derivations differ, the nodes being constructed will get different lists of children, and this difference guarantees that the parse trees are distinct. □

5.4.4 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. We saw, for example, that the language of expressions generated by the grammar of Fig. 5.2 is actually unambiguous. Even though that grammar is ambiguous, there is another grammar for the same language that is unambiguous — the grammar of Fig. 5.19.

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:

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

That is, L consists of strings in $\mathbf{a^+b^+c^+d^+}$ such that either:

1. There are as many a's as b's and as many c's as d's, or
2. There are as many a's as d's and as many b's as c's.

L is a context-free language. The obvious grammar for L is shown in Fig. 5.22. It uses separate sets of productions to generate the two kinds of strings in L.

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

Figure 5.22: A grammar for an inherently ambiguous language

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

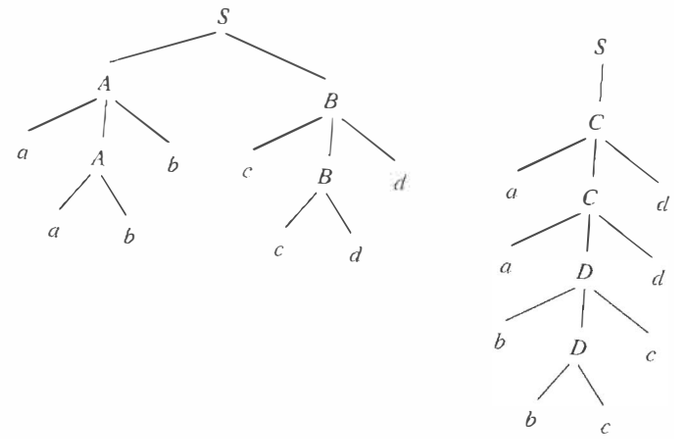
1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

and the two parse trees shown in Fig. 5.23.

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 the 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.

For instance, the only way to generate strings where the a's and b's have the same number is with a variable like A in the grammar of Fig. 5.22. There are variations, of course, but these variations do not change the basic picture. For instance:

- Some small strings can be avoided, say by changing the basis production $A \rightarrow ab$ to $A \rightarrow aaabbb$, for instance.
- We could arrange that A shares its job with some other variables, e.g., by using variables A_1 and A_2, with A_1 generating the odd numbers of a's and A_2 generating the even numbers, as: $A_1 \rightarrow aA_2b \mid ab$; $A_2 \rightarrow aA_1b$.



(a) (b)

Figure 5.23: Two parse trees for $aabbccdd$

- We could also arrange that the numbers of a's and b's generated by A are not exactly equal, but off by some finite number. For instance, we could start with a production like $S \rightarrow AbB$ and then use $A \rightarrow aAb \mid a$ to generate one more a than b's.

However, we cannot avoid some mechanism for generating a's in a way that matches the count of b's.

Likewise, we can argue that there must be a variable like B that generates matching c's and d's. Also, variables that play the roles of C (generate matching a's and d's) and D (generate matching b's and c's) must be available in the grammar. The argument, when formalized, proves that no matter what modifications we make to the basic grammar, it will generate at least *some* of the strings of the form $a^n b^n c^n d^n$ in the two ways that the grammar of Fig. 5.22 does.

5.4.5 Exercises for Section 5.4

*** Exercise 5.4.1:** Consider the grammar

$$S \rightarrow aS \mid aSbS \mid \epsilon$$

This grammar is ambiguous. Show in particular that the string aab has two:

a) Parse trees.

b) Leftmost derivations.

c) Rightmost derivations.

! **Exercise 5.4.2:** Prove that the grammar of Exercise 5.4.1 generates all and only the strings of a's and b's such that every prefix has at least as many a's as b's.

*! **Exercise 5.4.3:** Find an unambiguous grammar for the language of Exercise 5.4.1.

!! **Exercise 5.4.4:** Some strings of a's and b's have a unique parse tree in the grammar of Exercise 5.4.1. Give an efficient test to tell whether a given string is one of these. The test "try all parse trees to see how many yield the given string" is not adequately efficient.

! **Exercise 5.4.5:** This question concerns the grammar from Exercise 5.1.2, which we reproduce here:

$$\begin{array}{lcl} S & \to & A1B \\ A & \to & 0A \mid \epsilon \\ B & \to & 0B \mid 1B \mid \epsilon \end{array}$$

a) Show that this grammar is unambiguous.

b) Find a grammar for the same language that *is* ambiguous, and demonstrate its ambiguity.

*! **Exercise 5.4.6:** Is your grammar from Exercise 5.1.5 unambiguous? If not, redesign it to be unambiguous.

Exercise 5.4.7: The following grammar generates *prefix* expressions with operands x and y and binary operators $+$, $-$, and $*$:

$$E \to +EE \mid *EE \mid -EE \mid x \mid y$$

a) Find leftmost and rightmost derivations, and a derivation tree for the string `+*-xyxy`.

! b) Prove that this grammar is unambiguous.

5.5 Summary of Chapter 5

- *Context-Free Grammars*: A CFG is a way of describing languages by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols, and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.

- *Derivations and Languages*: Beginning with the start symbol, we derive terminal strings by repeatedly replacing a variable by the body of some production with that variable in the head. The language of the CFG is the set of terminal strings we can so derive; it is called a context-free language.