4CS4-6: Theory of Computation (Context-free languages and Grammars)

Lecture 13: March, 11 & 12, 2020

Prof. K.R. Chowdhary *: Professor of CS*

13.1 Closure properties of Context-free languages

Like closure properties studied earlier for regular languages, the context-free languages are also closed on certain operation. In the following, we prove these properties.

Theorem 13.1 *If L_1 and L_2 are context-free languages then $L_1 \cup L_2$, L_1L_2 and L_1^* are also context-free languages.*

Proof: This can be proved, almost on the similar lines of regular and finite automata discussed earlier. To prove that $L_1 \cup L_2$, L_1L_2, and L^* exists in the form of context-free languages, it is required to find out corresponding CFG for each of these cases.

Let us assume the CFG corresponding to L_1, L_2 are

$$G_1 = (V_1, \Sigma, S_1, P_1),$$
$$G_2 = (V_2, \Sigma, S_2, P_2).$$

Case I (For $L_1 \cup L_2$): Let $G = (V, \Sigma, S, P)$ be union CFG that generates $L_1 \cup L_2$. Let V and P be defined as

$$V = V_1 \cup V_2 \cup \{S\},$$
$$P = P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}.$$

If there is a string $x \in L(G_1)$ then S_1 generates x, i.e., $S_1 \Rightarrow_{G_1}^* x$. Similarly, for $y \in L(G_2)$, $S_2 \Rightarrow_{G_2}^* y$. In the case of x, union CFG first uses the production $S \to S_1$, then continues with the derivation of x in G_1. While in the case of y, the union CFG first uses the production $S \to S_2$ first, then continues with the derivation of y in G_2. Therefore, $L(G_1) \subseteq L(G)$, $L(G_2) \subseteq L(G)$, which can be combined to give $L(G_1) \cup L(G_2) \subseteq L(G)$.

On the other hand, if x is derivable from a production in G, the derivation must start with,

$$S \Rightarrow S_1, \text{ or}$$
$$S \Rightarrow S_2.$$

For $S \Rightarrow S_1$, all subsequent productions will be in G_1. Since $V_1 \cap V_2 = \phi$, therefore, $x \in L(G_1)$. Similar is case with $y \in L(G_2)$. Hence, $\{x\} \cup \{y\} \in L(G_1) \cup L(G_2)$, which is $L(G) \subseteq L(G_1) \cup L(G_2)$.

Case II (For L_1L_2): Let $G = (V, \Sigma, S, P)$ be a CF Grammar, which generates the CF language L_1L_2. Let G_1, G_2 be the CFGs corresponding to L_1, L_2 respectively. Let $V_1 \cap V_2 = \phi$. Various components of G can be specified as follows:

$$V = V_1 \cup V_2 \cup \{S\},$$
$$P = P_1 \cup P_2 \cup \{S \to S_1S_2\}.$$

Let $x_1 \in L(G_1)$, $x_2 \in L(G_2)$, and $x = x_1x_2 \in L(G_1)L(G_2)$. Then $x \in L(G)$ can be derived as follows:

$$S \Rightarrow S_1S_2 \Rightarrow^*_{G_1} x_1S_2 \Rightarrow^*_{G_2} x_1x_2 = x,$$

where, $S_1S_2 \Rightarrow x_1S_2$ is derivation in G_1, and $x_1S_2 \Rightarrow^* x_1x_2$ is a derivation in G_2. On the other hand if x can be derived from S_1S_2 and therefore $x = x_1x_2$, where x_1 can be derived from S_1 in G and x_2 can be derived from S_2 in G. Since $V_1 \cap V_2 = \phi$, the above S_1x_1 derivable from G implies it derivable from G_1. Similarly x_2 derivable from G implies that it is derivable from G_2. Hence $x = x_1x_2 \in L(G_1)L(G_2)$.

Case III (For L_1^*): For this it is required to construct $G = (V, \Sigma, S, P)$, which generates L_1^*. Let $V = V_1 \cup \{S\}$ and $S \notin V_1$.

The language L_1^* will contain strings of the form $x = x_1x_2 \dots x_n$, where $x_i \in L$. Since x_i can be derived from S_1, for G it is only required to get these strings derived from S. Therefore, a production in G in terms of production of G_1 can be specified as,

$$S \to S_1S \mid \varepsilon$$

Hence, the production in G can be expressed as,

$$P = P_1 \cup \{S \to S_1S \mid \varepsilon\}.$$

For x to be member of $L(G)$, $x = \varepsilon$ or x can be derived from some string $x_i^* \in L(G_1)$. In the later case, the only productions in G are those that exist in G_1. Therefore, $x \in L(G_1)^*$. This proves the theorem. ■

13.2 Ambiguous Grammars and Languages

One special desirable feature in a grammar for a programming language is freedom from *ambiguity*. A CFG G is said to be ambiguous if and only if some string in $L(G)$ has two distinct left-most derivations. A CFG G is said to be *inherently ambiguous* if and only if

every equivalent CFG is ambiguous. A CFG is said to be *structurally ambiguous* if and only if some string in $L(G)$ has two distinct derivation-trees that are the same except for non-terminal labels.

There is extensive interest in ambiguity of natural languages as well as context-free languages. Some of the known results are:

1. it is *recursively unsolvable* whether an arbitrary grammar for a language is ambiguous;
2. there exists *inherently ambiguous languages*, for example,

$$\{a^i b^j c^i d^k \mid i, j, k \geq 1\} \cup \{a^i b^j c^k d^j \mid i, j, k \geq 1\},$$

3. no regular language is inherently ambiguous.

The above considerations motivate the study of ambiguity in context-free languages. Given a grammar $G = (\Sigma, V, S, P)$, a word w is said to be ambiguously derivable if there exists at least two derivations of w from S whose associated generation trees are different. A grammar G is said to be ambiguous if there exists an ambiguously derivable word in $L(G)$, and is otherwise unambiguous.

Definition 13.2 Unambiguous. *A CFG is unambiguous if for every $w \in L(G)$ there is a unique parse-tree in G with yield w.*

Definition 13.3 Inherently ambiguous. *A language L is said to be inherently ambiguous if there is no unambiguous grammar for L.*

Following are additional terms related to ambiguity.

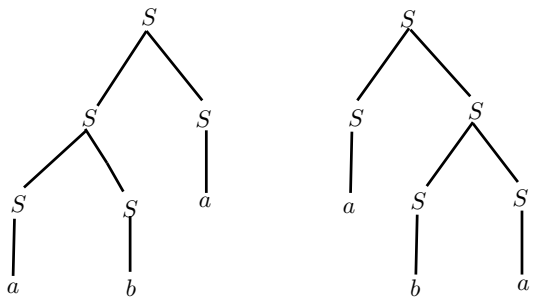
- The degree of ambiguity of a sentence is the number of its distinct derivation-trees.
- A sentence is unambiguous if its degree of ambiguity is unity.
- A grammar is unambiguous if each of its sentences is unambiguous.
- A grammar has bounded ambiguity if there is a bound b on the degree of ambiguity of any sentence of the grammar.
- A grammar is reduced if every non-terminal appears in derivation of some sentence.

If T_1 and T_2 are two distinct generation trees, then their associated leftmost derivations are distinct; the converse also holds. Thus a word is ambiguously derivable if and only if it has two leftmost derivations. It is well established that there is no decision procedure for determining whether an arbitrary grammar is ambiguous. Also there is no decision procedure (i.e., it is unsolvable) for determining if an arbitrary language is inherently ambiguous. In other words, there is no algorithm for determining of an arbitrary language whether it is generated by some unambiguous grammar. Since parse-tree is a semantics expression of generated language string, the two different parse-trees represent two different meanings of the generated string.

Given a grammar $G = (V, \Sigma, P, S)$, a word w is said to be unambiguously derivable if there exists two derivations of w from S, whose associated generation trees are different. A grammar G is said to be ambiguous if there exists an ambiguously derivable word in $L(G)$, and is otherwise unambiguous. A language L is said to be inherently ambiguous if there is no unambiguous grammar for L.

Example 13.4 *Show that grammar with productions $\{S \to a, S \to b, S \to SS\}$, is ambiguous.*

We show following two derivation-trees 13.1(a), (b) for the same sentence aba, which, as per the definition, concludes that this grammar is ambiguous.



(a)

$S \Rightarrow SS$
$\Rightarrow SSS$
$\Rightarrow aSS$
$\Rightarrow abS$
$\Rightarrow aba$

(b)

$S \Rightarrow SS$
$\Rightarrow aS$
$\Rightarrow aSS$
$\Rightarrow abS$
$\Rightarrow aba$

Figure 13.1: (a) Parse-tree I for sentence aba, (b) Parse-tree II for the same sentence.

Example 13.5 *Let us consider the grammar for arithmetic expressions with productions given below.*

$$E \to E + E \mid E * E \mid (E) \mid a.$$

For the sake of simplicity only two operations: '+' and '$*$' have been considered for this purpose as these are sufficient to express the ambiguity presence in the generated arithmetic expressions. Let us generate the string $a + a * a$. The corresponding derivations and derivations trees are shown in Fig. 13.2(a), (b).

The Fig. 13.2(a) shows that 2nd and 3rd term in the expression are to be multiplied and then the result is to be added into the first term. Whereas the derivation-tree in Fig. 13.2 shows, first and second terms of the expression are to be added together first and then the result of this is to be multiplied with the third term. Obviously, the two derivations represent different meanings of the expression $a + a * a$. Therefore, this grammar is ambiguous and the corresponding language is ambiguous too. □

An ambiguity in ALGOL 60, for example, is

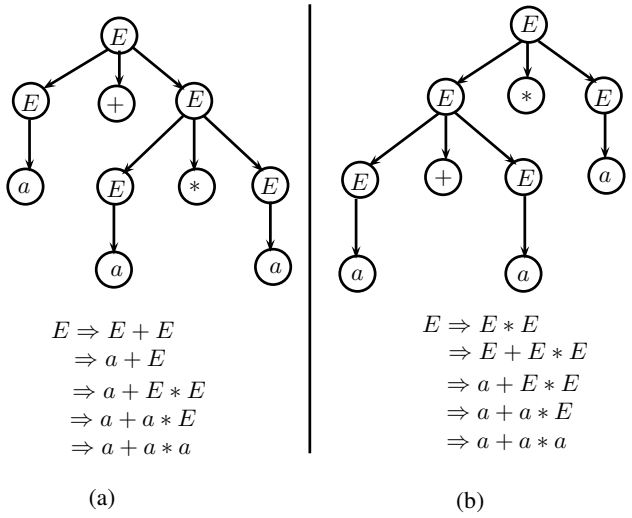


Figure 13.2: (a) Derivation and parse-tree for $a + a * a$, (b) Derivation and parse-tree for the same expression.

$$\text{if } \beta_1 \text{ then } \alpha \text{ if } \beta_2 \text{ then } \Sigma_1 \text{ else } \Sigma_2,$$

where β_1, β_2 represents any Boolean expressions,α a statement, and Σ_1, Σ_2 are unconditional statements.

Ambiguous grammars represent different parse-trees for the same expression. The different parse-trees always represent different meanings. Therefore, it is necessary to remove the ambiguity in the CFGs. One way to represent the ambiguity in a language expression is to consider the precedence of operators. Thus, in $a+a*a$, the part $a*a$ needs to be processed before the '+' operator. Hence, in the parse-tree answer $a * a$ is computed first. However, when there are identical operators they are grouped from left to right. For example, $a+a+a$ may be grouped as $((a + a) + a)$ or $(a + (a + a))$. In two cases the parse-tree are different, but they will evaluate the same result.

Naturally, we are interested to have some algorithm to remove all kinds of ambiguities in a CFG. However, it is *recursively unsolvable* whether an arbitrary grammar for a language is ambiguous. In other words, there cannot be an algorithm to determine if an arbitrary language is generated by some ambiguous grammar or not.

In fact, there are CFLs that have ambiguous CFGs, but removal of ambiguity for such grammar is impossible. It is not always true that ambiguity can be removed from a grammar. However, an ambiguous grammar can always be converted into unambiguous grammar by some changes in it. Sometimes, ambiguous grammars can be made unambiguous by adding some non-terminal symbols through process called disambiguation so that for a given expression there is one parse-tree and one derivation only.

Example 13.6 *Disambiguate the CFG* $G = (V, \Sigma, S, P)$*, where,*

$$V = \{E\}$$
$$\Sigma = \{(,), +, *, a\}$$
$$S = \{E\}$$
$$P = \{E \to E + E, E \to E * E, E \to (E), E \to a\}.$$

This grammar is ambiguous as it generates two difference parse-trees for the arithmetic expression $a + a \to a$. Following modifications can be made in G to disambiguate it.

$$V = \{E, T, F\},$$
$$P = \{E \to E + T \mid T, T \to T * F \mid F, F \to a \mid (E)\}$$

The Σ and S remain unchanged.

The parse-tree for $a + a * a$ are shown in Fig. 13.3.

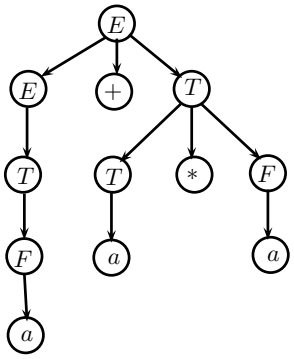


Figure 13.3: Parse-Tree for an unambiguous language

In the rewritten grammar, for any expression containing "+" and "$*$", the parse-tree will situate "$*$" deeper in the tree, i.e., closer to the terminals, than the "+". Thus, in effect, forcing the evaluation of "$*$" before the evaluation of "+".

We note that no other derivation and parse-tree are possible for this string. Hence, the modified grammar is unambiguous.

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \\ &\Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \\ &\Rightarrow a + a * a. \end{aligned}$$

Similarly, if $a * a + a$ is expression, the derivation is as follows:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow T * F + T \Rightarrow F * F + T \\ &\Rightarrow a * F + T \Rightarrow a * F + T \Rightarrow a * a + T \Rightarrow a * a + a. \end{aligned}$$