COM364 Automata Theory
Lecture Note*5 - Context-Free Grammars

Kurtuluş Küllü

May 2018

Until now, we talked about FA and regular expressions which are equally powerful and can be used to recognize or describe regular languages. But lastly, when we looked at the pumping lemma, we saw that there are languages such as $\{0^n1^n \mid n \in \mathbb{N}\}$ that is beyond the regular languages.

Now we will go beyond the regular languages and discuss context-free grammars (CFGs) and context-free languages (CFLs) that they describe. A CFG is a mechanism to describe a language. The languages that are described with CFGs are CFLs (and they include regular languages). Later, we will discuss pushdown automata (PDA) which is a type of machine recognizing CFLs.

Example: Below you see the first example context-free grammar.

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned}$$

Such a grammar is considered as a collection of *substitution rules* (or *productions*). These include *variables* (*nonterminals*) and *terminals* (alphabet symbols). One of the variables is considered as the *start variable* which is by convention often the variable on the left side of the arrow in the topmost substitution rule.

In the example grammar above, A and B are the variables, 0, 1, and # are the terminals, and A is the start variable. We will often use capital letters or words to indicate variables.

A grammar can be used to generate strings as follows:

1. Start with the start variable.
2. For one of the variables you have, use a rule with that variable on the left and apply the substitution.
3. Repeat 2 until there are no variables.

Example: You can see an example string generation below.

$$\begin{aligned} A &\Rightarrow 0A1 \\ &\Rightarrow 00A11 \\ &\Rightarrow 000A111 \\ &\Rightarrow 000B111 \\ &\Rightarrow 000\#111 \end{aligned}$$

Such a sequence of substitutions is called a *derivation*. Same information can be given with a *parse tree* as in Figure 1. All strings that can be generated from a grammar is the language of that grammar. If we call the grammar G, then its language is shown with $L(G)$. For the example grammar above, its language is $\{0^n\#1^n \mid n \in \mathbb{N}\}$.

Note: We use an extra notation as a shortcut. Instead of writing several rules for the same variable on separate lines, we combine them in a single line with the bar symbol (|). E.g., instead of writing $A \to 0A1$ and $A \to B$ separately, we write $A \to 0A1|B$.

*Based on the book "Introduction to the Theory of Computation" by Michael Sipser.

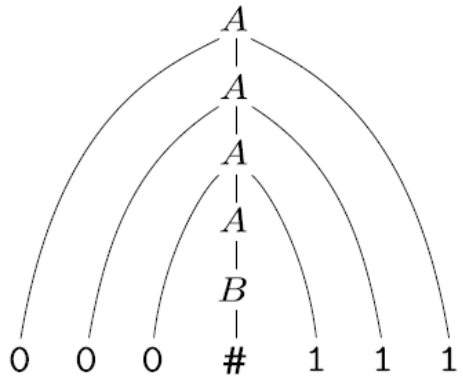

Figure 1: The parse tree for the derivation example (string 000#111).

Formal Definition

A CFG is a 4-tuple (V, Σ, R, S), where

1. V is a finite set of variables,
2. Σ is a finite set of terminals disjoint from V,
3. R is a finite set of rules with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Example: For the example grammar we saw, $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$, and R is the set of rules given above in the first example.

Note: Often, we only write the rules as we did in the first example. The symbols that appear on the left side of arrows are taken to be the variables. All remaining symbols except | are terminals. Start variable is the left side variable of the top rule.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule in the grammar, then we say that uAv *yields* uwv and write this as $uAv \Rightarrow uwv$.

Also, we say that u derives v and we write $u \Rightarrow^* v$, if $u = v$ or if there's a sequence $u_1, u_2, \ldots, u_k$ for $k \geq 0$ such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$. So, the language of a grammar can be expressed as $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Example: $G_3 = (\{S\}, \{a, b\}, R, S)$ where R is $S \to aSb|SS|\varepsilon$.

Note the use of empty string (ε) in this grammar. It allows S to be replaced with empty string during derivations. So, some example strings that can be derived using these rules are $\varepsilon, ab, abab, aabb, aabbab$. Can you describe the language of this grammar? **Hint:** Put (symbol instead of a and) symbol instead of b.

Example: Consider the following grammar G_4.

$$\begin{aligned} \langle \text{EXPR} \rangle &\to \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\to \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\to (\langle \text{EXPR} \rangle) \mid a \end{aligned}$$

The parse trees for strings $a + a \times a$ and $(a + a) \times a$ can be seen in Figure 2.

A very important use of CFGs and parse trees in computers happens during program compilation. A programming language compiler translates strings written in that language to the machine language. While doing this, one of the stages is to correctly understand the meaning of the string. CFGs are used to define programming languages and a part of the compiler (called *parser*) processes given strings as if it is trying to find the parse tree for it.

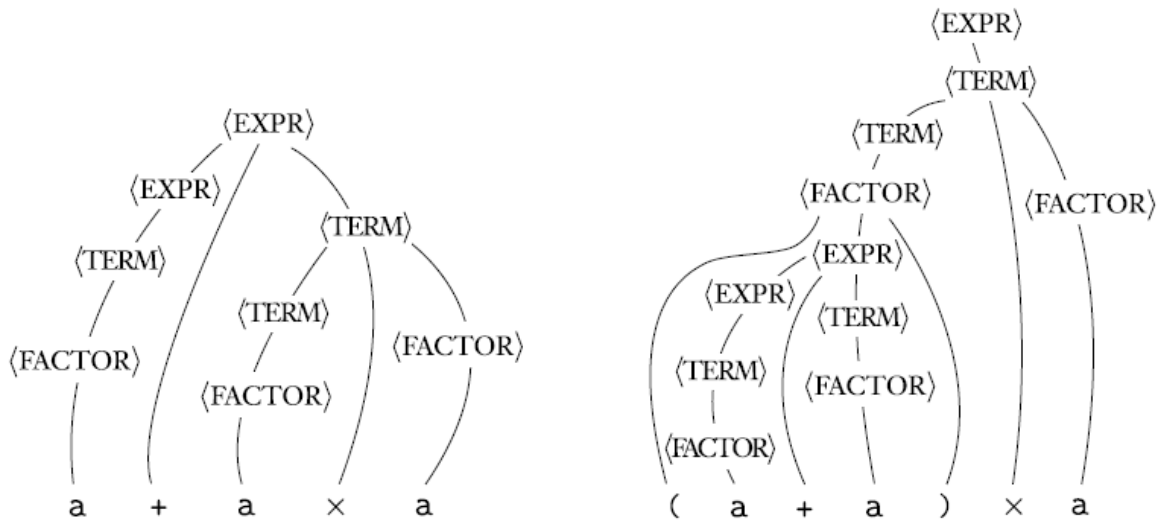

Figure 2: Parse trees for $a + a \times a$ and $(a + a) \times a$ according to CFG G_4.

Designing CFGs

Once again, this process requires creativity and there is no general method to achieve this. But the following techniques are often useful.

- Many CFLs are the union of simpler CFLs. So, breaking down and writing grammars for each part can help. **E.g.,** $\{0^n1^n \mid n \in \mathbb{N}\} \cup \{1^n0^n \mid n \in \mathbb{N}\}$. For the first set, we can write $S_1 \to 0S_11 \mid \varepsilon$ and for the second set, we can write $S_2 \to 1S_20 \mid \varepsilon$. To combine, $S \to S_1 \mid S_2$.
- If the language is regular (do not forget that the set of regular languages is a subset of the set of CFLs), you can design the DFA and find the equivalent grammar by using a variable R_i for each state q_i, writing rules such as $R_i \to aR_j$ if $\delta(q_i, a) = q_j$, and $R_i \to \varepsilon$ if q_i is accepting.
- Many CFLs contain strings with parts that are "linked" in the sense that a machine would need to remember something about the first part when going over the second part. E.g., $\{0^n1^n \mid n \in \mathbb{N}\}$. You can use rules like $R \to uRv$ if u and v are linked somehow and to force the relationship between u and v.
- Strings can have recursive structures (and base cases for these recursive structures). For example, if we have a palindrome, when you remove the first and last symbols, the remaining string in the middle should also be a palindrome. The base case here is that the middle (or the palindrome itself) can be the empty string. You can force this structure in a grammar by defining recursive rules such as $R \to uRu \mid vRv \mid \varepsilon$.

Example: Write a CFG for language $\{w \mid w \text{ is a palindrome}\}$ on alphabet $\Sigma = \{0, 1\}$. (Same language can be defined as $\{w \mid w \in \{0,1\}^* \text{ and } w = w^R\}$ with w^R meaning w reversed.)

$$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$

Example: Write a CFG that generates the language $\{w \mid w$ starts and ends with the same symbol$\}$ on alphabet $\Sigma = \{a, b, c\}$.

$$S \to aMa \mid bMb \mid cMc$$

Ambiguity

Sometimes, a grammar can generate the same string in different ways causing different possible parse trees (and therefore, different meanings). This is not desirable in some applications such as

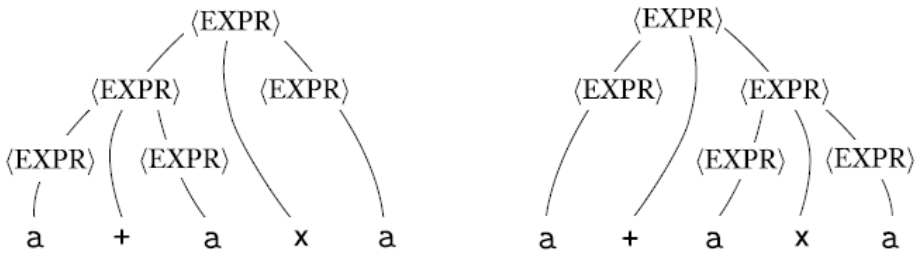

Figure 3: Parse trees for $a + a \times a$ according to CFG G_5 showing that this grammar is ambiguous.

programming language design. If it is possible to generate a string with different parse trees, we say that the grammar is *ambiguous*.

Example: Consider the grammar G_5 given below.

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

This grammar generates exactly the same strings with the grammar G_4 we saw before. However, ambiguity is a major difference between these two grammars. Consider the string $a + a \times a$. There are two possible parse trees you can draw for this string according to G_5. These can be seen in Figure 3. On the other hand, all strings will have a single parse tree using G_4, so that grammar is unambiguous.

Note: Ambiguity is shown with different parse trees for the same string, NOT different derivations. Different derivations can be written for the same parse tree by changing the variable substituted at each step. If we put a standard to this and, for example, choose always the first variable on the left (or right), then this derivation is called a *leftmost* (or *rightmost*) derivation.

Definition: A string w is derived ambiguously in CFG G if it has two or more different leftmost (or rightmost) derivations. G is ambiguous if it generates a string ambiguously.

Note: For some ambiguous CFGs, we can write an equivalent unambiguous one (like the case for G_5 and G_4), but some languages are "inherently ambiguous", meaning that no unambiguous CFG can be written for them. For example, $\{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } (i = j \text{ or } j = k)\}$ is one such language. It is not possible to write an unambiguous grammar for this language.

Chomsky Normal Form (CNF)

Sometimes, it is useful to have a CFG in a simple, well-defined form and *Chomsky Normal Form* (CNF) is one such form. For example, it is mostly useful for a grammar to be in CNF when we speak about algorithms working on grammars.

Definition: A CFG is in CNF if every rule is in the form $A \to BC$ or $A \to a$ where a is any terminal and A, B, C are any variables (except that B and C are not start variables). In addition, we allow the rule $S \to \varepsilon$ only if S is the start variable.

Theorem: Any context-free language has a context-free grammar in Chomsky normal form that generates it.

Proof Idea: Any grammar G can be converted into one in CNF. This is done in several steps by replacing problematic rules with suitable ones without changing the language.

- First, add a new start variable (to ensure that the start variable doesn't appear on the right side of other rules).
- Then, eliminate ε-rules like $A \to \varepsilon$.
- Lastly, work on other rules (unit rules like $A \to B$ and those with more than two variables on the right).

Example: Let G_6 be as follows and convert it to a CFG in CNF.

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

The following changes will convert G_6 to a CFG in CNF.

1. Add new rule $S_0 \to S$, where S_0 is the new start variable.
2. (a) Remove $B \to \varepsilon$ by adding $A \to \varepsilon$ and $S \to a$.
   (b) Remove $A \to \varepsilon$ by adding $S \to SA \mid AS \mid S$.
3. (a) Remove unit rules $S \to S$ (by adding nothing) and $S_0 \to S$ by adding possible S replacements to the S_0 rule as $S_0 \to ASA \mid aB \mid a \mid AS \mid SA$.
   (b) Remove unit rules $A \to B$ (by adding b to the A rule instead of B) and $A \to S$ by adding possible S replacements to A rule as $A \to b \mid ASA \mid aB \mid a \mid AS \mid SA$.
4. Go over all the rules so that we either have only a single terminal or two variables on the right side of each rule. This can simply be achieved by adding new variables and rules. The resulting grammar in CNF that generates the same language as G_6 is given below.

$$S_0 \to AA_1 \mid UB \mid a \mid AS \mid SA$$
$$S \to AA_1 \mid UB \mid a \mid AS \mid SA$$
$$A \to b \mid AA_1 \mid UB \mid a \mid AS \mid SA$$
$$A_1 \to SA$$
$$U \to a$$
$$B \to b$$