CSCI 341–Fall 2024: Lecture Notes
Set 8: Context-Free Grammars

Edward Talmage

October 2, 2024

Let us look at a new way to build languages. This is inductive, like regular expressions, but more powerful and thus more complex.

Example: $L = \{w \mid w \text{ is a palindrome}\}$. We know that we cannot give a regular expression for this language, as it is not regular. But can we give an inductive definition for it?

Exercise: What are the base cases for the set of all palindromes? If you have a palindrome, how do you build a bigger one, or conversely, how do you shrink a palindrome down to a smaller one? Can you put those together into an inductive definition? Did you get all palindromes and nothing else?

We can give a *Context-Free Grammar* for L:

$$S \to \varepsilon \mid a \mid aSa$$
$$(\text{for all } a \in \Sigma)$$

Context-Free Grammars (CFGs) are a limited way of writing inductive set definitions for languages. There are several rules, which we will introduce as we need them:

1. Capital letters are variables which we recursively expand.
2. Arrows indicate substitution rules, by which we expand variables.
3. Pipes (|) are OR operators, separating different possible expansions for a variable.

Exercise: What language does the following CFG generate?

$$\begin{aligned} S &\to 0S1 \mid B \\ B &\to \varepsilon \end{aligned}$$

4. Every variable must appear on the left side of a substitution rule.
5. Every derivation (sequence of replacements to generate a string) starts with the variable which appears first in the CFG.

Exercise: What language does the following CFG generate?

$$\begin{aligned} S &\to AB \\ A &\to 0A1 \mid \varepsilon \\ B &\to 1B0 \mid \varepsilon \end{aligned}$$

We can break down a language by looking at the set of strings each variable can produce. Another way of thinking about this is to consider what the language would be if each variable were the start variable. For this example,

- $L(A) = \{0^n 1^n \mid n \in \mathbb{N}\}$

- $L(B) = \{1^n 0^n \mid n \in \mathbb{N}\}$
- $L(S) = L(A) \circ L(B) = \{0^n 1^{n+m} 0^m \mid n, m \in \mathbb{Z}\}$

5. The left hand side of each substitution rule is a single variable alone.
   - This is why we call these "context-free": each replacement happens independently, with no knowledge of where the variable is in the string or what is around it.

Exercise: Give a CFG for $\{w \mid len(w) \equiv 0 \pmod 2\}$ using only one variable. Give another CFG for this language using two variables.

$$S \to \varepsilon \mid S00 \mid S01 \mid S10 \mid S11$$

$$T \to \varepsilon \mid AAS$$
$$A \to 0 \mid 1$$

1 Formal Definition

Definition 1. A *Context-Free Grammar* is a 4-tuple (V, Σ, R, S), where

- V is a finite set, elements are called *variables*.
- Σ is a finite set, disjoint from V, of *terminal symbols*. In other words, Σ is the alphabet.
- R is a finite set of rules, each of the form $A \to w$, where $A \in V, w \in (\Sigma \cup V)^*$.
- $S \in V$ is the start variable.

We also need to define ways to talk about generating strings with a CFG:

- If $w, v \in (\Sigma \cup V)^*$, and $A \in V$, we say that $wAv \Rightarrow u\alpha v$ if $(A \to \alpha) \in R$, and read this as "wAv *yields* $w\alpha v$".
- Similarly, we say that w *derives* v, denoted $w \overset{*}{\Rightarrow} v$, iff $w = v$ or $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_k \Rightarrow v$ for some sequence of $w_1, \dots, w_k$, each in $(\Sigma \cup V)^*$.
- The language generated by variable A is $L(A) := \{w \in \Sigma^* \mid A \overset{*}{\Rightarrow} w\}$.
- The language generated by grammar G is $L(G) := \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.
- A language L is called a *Context-Free Language* (CFL) if there is a CFG which generates it.

Exercise: What language does the following CFG generate: $(\{S\}, \{(,)\}, \{S \to (S), S \to SS, S \to \varepsilon\}, S)$

2 Closure Properties

As with regular languages, closure properties help us, both to generate new CFLs and (later) to show that languages are not context-free.

Theorem 1. *If L_1, L_2 are CFLs over Σ, then $L_1 \cup L_2$, $L_1 \circ L_2$, and L_1^* are CFLs.*

Exercise: How will we go about proving these?

Proof. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be CFGs s.t. $L(G_1) = L_1, L(G_2) = L_2$. Assume WLOG that $V_1 \cap V_2 = \emptyset$.

Exercise: Split into three groups and each group prove one of the three claims of the theorem.

1. $L_1 \cup L_2$: We need to generate everything generated by either G_1 or G_2. We can do this by adding a new start variable which can yield either of the two old start states:

$$G_\cup = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 \mid S_2\}, S), S \notin V_1 \cup V_2$$

   This is a CFG generating $L_1 \cup L_2$, since S can generate any string generated by S_1 or by S_2, so $L_1 \cup L_2$ is a CFL.

2. $L_1 L_2$: To concatenate, generate a string in L_1, then a string in L_2, and concatenate them. To do this in a CFG, we add a new start variable and a rule replacing it with the concatenation of the old start variables:

$$G_\circ = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S), S \notin V_1 \cup V_2$$

   Since S_1 can generate any string in L_1 and S_2 can generate any string in L_2, S generates the concatenation of any string in L_1 with any string in L_2, and thus we have a CFG for $L_1 L_2$, proving it is context-free.

3. L_1^*: We need to be sure we generate the empty string, and any number of strings in L_1, which we then concatenate together. To do this, we again add a new start variable, which can generate ε or any number of copies of S_1 in a row.

$$G_* = (V_1 \cup \{S\}, \Sigma, R_1 \cup \{S \to SS_1 \mid \varepsilon\}, S), S \notin V_1$$

   S can generate any number (including 0) of copies of S_1 by repeatedly applying the first rule, ending with an application of the second rule, and each of those copies generates a string in L_1, so S generates any string formed by number of strings in L_1 concatenated together. Thus, there is a CFG generating L_1^*, so it is context-free.

□

This result suggests that every regular language is context-free.

Theorem 2. *Every regular language is context-free.*

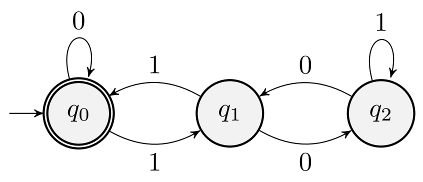
Proof. Let L be any regular language and $M = (Q, \Sigma, \delta, q_0, F)$ a DFA recognizing L. Construct a CFG generating L as $G = (V, \Sigma, R, S)$, with

- $V = Q$
- $R = \{q_i \to aq_j \mid \delta(q_i, a) = q_j\} \cup \{V \to \varepsilon \mid V \in F\}$
- $S = q_0$

Any string which drives M to an accept state will be generated by following the same path in the CFG. □

Exercise: Give a CFG which generates $L = \{w \mid w \text{ is a binary multiple of 3}\}$ by giving a DFA for L and converting it to a CFG.

- DFA:



- CFG:

$$\begin{aligned} q_0 &\to 0q_0 \mid 1q_1 \mid \varepsilon \\ q_1 &\to 0q_2 \mid 1q_0 \\ q_2 &\to 0q_1 \mid 1q_2 \end{aligned}$$

Tips

- If parts of the language you are trying to show is context free are regular, use tools for regular languages, and convert to CFG later.
- Languages with connected, related parts, like 0^n1^n, tend to generate rules like $A \to xAy$, which allows matching counting.

Exercise: Give a CFG for $\{0^n q^{2n} \mid n \le 0\}$

- Recursion in a language tends to lead to recursion in CFGs. Recall the parenthesization example above: $S \to (S) \mid SS \mid \varepsilon$.
  - Example: Suppose you are generating a CFG for a programming language. A block of code can be a statement followed by a block, an `if` containing a block, or a loop containing a block. Each block can similarly be the same, so we will have

$$\begin{aligned} B &\to SB \mid I \mid L \\ I &\to \text{if } B \text{ then } B \\ L &\to \text{while } B \text{ do } B \mid \text{for } B \text{ do } B \end{aligned}$$

  - Fun note: You can see the grammar for the python programming language at `https://docs.python.org/3/reference/grammar.html`. Other language's grammars are also available.

Examples:

1. $a^*b^*c^*$

$$\begin{aligned} S &\to ABC \\ A &\to aA \mid \varepsilon \\ B &\to bB \mid \varepsilon \\ C &\to cC \mid \varepsilon \end{aligned}$$

2. $\{w \mid\ w$ has an even number of 1's$\}$

$$\begin{aligned} S &\to 0S \mid 1A \mid \varepsilon \\ A &\to 0A \mid 1S \end{aligned}$$

3. $\{w \mid\ w$ is parenthesis-balanced$\}$ (alternate)

$$S \to (S)S \mid \varepsilon$$

4. $\{a^ib^j \mid i < j\}$

$$\begin{aligned} S &\to Sb \mid Ab \\ A &\to aAb \mid \varepsilon \end{aligned}$$

Exercises:

Exercise: Give CFGs for the following languages:

1. $\{a^ib^j \mid i < j, i \text{ even}\}$
2. $\{a^ib^jc^k \mid i = j \text{ OR } j = k\}$
3. $\{w \mid w$ has the same number of 0's and 1's$\}$

1. $\{a^i b^j \mid i < j, i \text{ even}\}$

$$\begin{aligned} S &\to Sb \mid Ab \\ A &\to \varepsilon \mid aaAbb \end{aligned}$$

2. $\{a^i b^j c^k \mid i = j \text{ OR } j = k\}$

$$\begin{aligned} S &\to AC \mid B \\ A &\to \varepsilon \mid aAb & L(A) = \{a^i b^i \mid \ i \text{ even}\} \\ B &\to aBc \mid B' \\ B' &\to \varepsilon \mid Bb \\ C &\to \varepsilon \mid cC \end{aligned}$$

3. $\{w \mid w \text{ has the same number of 0's and 1's}\}$

$$\begin{aligned} S &\to \varepsilon \mid 0N \mid 1Z \\ N &\to 0S \mid 1NN & L(N) = \{w \mid \ w \text{ has one more 0 than 1}\} \\ Z &\to 1S \mid 0ZZ & L(Z) = \{w \mid \ w \text{ has one more 1 than 0}\} \end{aligned}$$

3 Ambiguity

3.1 Derivations

Recall our definition that a string w of terminals and variables *derives* a string v, written $w \overset{*}{\Rightarrow} v$, if there is a chain of variable expansions from w to v. If we list these expansions, we call that chain a *derivation*. For example, consider the grammar

$$\begin{aligned} S &\to T \mid U \\ T &\to 0T1 \mid \varepsilon \\ U &\to 1U0 \mid \varepsilon \end{aligned}$$

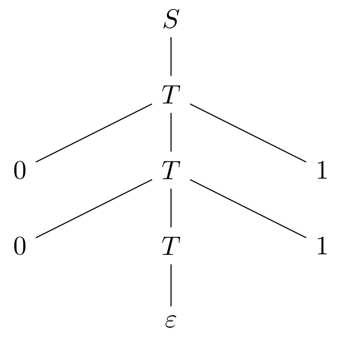
Some sample derivations are

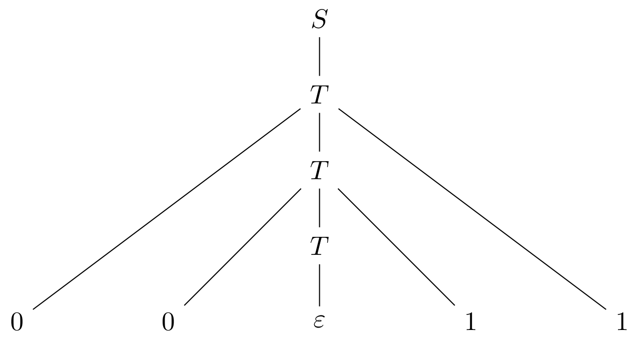
- $S \Rightarrow T \Rightarrow 0T1 \Rightarrow 00T11 \Rightarrow 00\varepsilon 11 \Rightarrow 0011$
- $S \Rightarrow U \Rightarrow \varepsilon$

Exercise: What language does this CFG generate?

3.2 Parse Trees

Another way to represent the generation of a string is a *parse tree*, which is much like a recursion tree for analyzing the runtime of a recursive function. The root is the start variable, and children are the various elements generated by a rule expansion. We continue to expand each variable until we have only terminals, which are leaves. Reading the leaves in left-to-right order gives the generated string. For an example, consider the string we generated above:



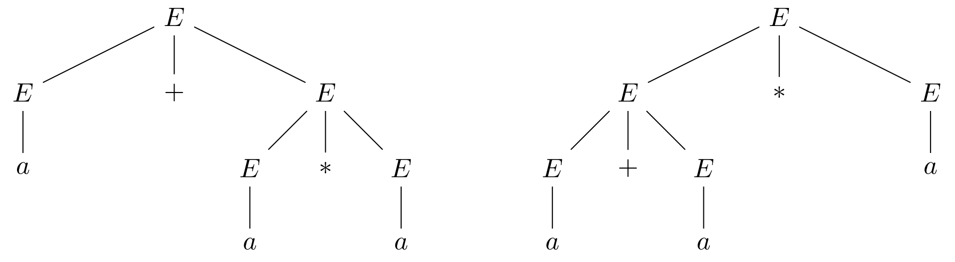


Note that sometimes we will draw all leaves at the same level to make it easier to read the string. This is great, after the fact, but hard to do while generating the tree.

3.3 Ambiguity

Similarly to NFAs, we have so far not worried about *how* a grammar generates a particular string, merely whether or not there is *some* derivation for it. However, some grammars may have different parse trees for the same string, which may or may not convey different meanings. Certain proofs will also be much more difficult if there are multiple equivalent derivations.

Example: Consider the grammar $E \to E + E \mid E * E \mid (E) \mid a$ and the two following parse trees.



While the generated string is the same in both cases, the trees suggest different meanings. The left tree multiplies, then adds, while the right adds then multiplies. Different derivations may not capture this logical distinction, as they capture only order, not structure.

For the rest of the course, we will restrict ourselves to *leftmost derivations*, which always replace the first, or leftmost, variable first. Now, we can formally define what it means for a CFG to be ambiguous.

Definition 2. A CFG G derives a string w *ambiguously* if w has two or more leftmost derivations. Grammar G is ambiguous if it generates any string ambiguously.

Some ambiguous grammars generate languages which also have an unambiguous grammar. Other CFLs are inherently ambiguous, meaning that any grammar generating that language must be ambiguous.

Example: Here is an unambiguous version of the previous grammar for arithmetic expressions:

$$\begin{aligned} E &\to E + E \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid a \end{aligned}$$

Exercise: Give an unambiguous grammar for the language of strings with equal numbers of 0's and 1's.