CFL Supplement

Michael Levet

June 4, 2017

Today, we will introduce the notion of grammars to generate languages. Grammars provide a recursive set of rules used to generate strings. The recursive structure allows for effective parsing mechanisms. Grammars are particularly useful in programming and markup language design.

1 Context-Free Languages

Recall that the goal of automata theory is to formalize the notions of an algorithm and computation machines. This is perhaps the most intuitive way to introduce context-free languages. Context-Free Languages are those languages accepted by a machine called a pushdown automaton. Conceptually, a pushdown automaton starts with a finite state automaton then adds a stack. So now the computation machine has memory aside from the current state and character. Observe as well that all regular languages are context free. This is easy to see, as a pushdown automaton can accept a regular language simply by ignoring the stack.

Some common examples of context-free languages are $\{0^n1^n : n \geq 0\}$ and the language of balanced parentheses. Examples of strings with balanced parentheses include (()) and ()(), while (() is unbalanced. These languages will be analyzed in greater detail later.

Formally, a context-free language is exactly the set of strings generated by a context-free grammar. The term "context-free grammar" is often times abused to denote the language itself. The context-free grammar will be formally introduced and examined. The pushdown automaton will be covered more thoroughly in a later tutorial.

Context-Free Grammar: A Context-Free Grammar is a four-tuple (N, T, P, S) where:

- N is the set of non-terminal symbols. Each non-terminal symbol represents a set of strings- exactly those strings which can be reached by it. Note that non-terminals may reach other non-terminals.
- T is the set of terminal symbols, which is equivocally the alphabet for the language.
- P is the set of productions or rules. Each production represents the recursive definition of the language. A production consists of a non-terminal symbol as the head, followed by the production symbol $\rightarrow$. The string $\omega \in (N + T)^*$ on the right-hand side of the production symbol is known as the body.
- S is the start symbol. The context-free grammar is generated starting at S and following the productions until only terminals remain.

Example 1. Consider the example above with $L = \{0^n1^n : n \geq 0\}$. Let's construct a context-free grammar to generate the language L. Let $G = (N, T, P, S)$ be the grammar. The terminal characters are clearly $T = \{0, 1\}$. As grammars define languages recursively, the goal is to build L from the ground up. So what are the base cases? They are $\epsilon, 01$. Now using these building blocks, how is 0011 constructed? The only answer is to stick a 01 in the middle of another 01, giving 0(01)1. More generally, 0^n1^n is constructed by nesting n 01 terms. So the grammar can be constructed with the single non-terminal symbol S and the production rules:

$$S \rightarrow \epsilon$$
$$S \rightarrow 01$$
$$S \rightarrow 0S1$$

More succinctly, we write:

$$S \to 0S1|01|\epsilon$$

Example 2. Now consider the language of balanced parentheses. We seek to build a grammar $G = (N, T, P, S)$ to generate this language. The terminal symbols are clearly $T = \{(,)\}$. Just like in the last example, it is important to start from the bottom up. So what are the building blocks for this language? They are $\epsilon, ()$. Now there are two cases to consider. The first is similar to the example for $L = \{0^n 1^n : n \geq 0\}$, where parentheses can be nested. The other case is when a pair of balanced parentheses are right next to each other: $()()$. A single non-terminal is required, so $N = \{S\}$, and the production rules simply deal with the cases mentioned above:

$$S \to \epsilon$$
$$S \to (S)$$
$$S \to SS$$

Definition 1 (Yields Relation)**.** Let $G(N, T, P, S)$ be a grammar, and let $\alpha, \beta \in (N + T)^*$. We say that the string α *yields* β, denoted $\alpha \implies {}^*\beta$, if it is possible to obtain β starting by using the productions in P finitely many times. A *derivation* of β (from α) is the sequence of productions used to produce β from α. A *leftmost* (resp. *rightmost*) derivation is where at each stage, we replace the leftmost (resp. rightmost) non-terminal.

Example 3. Consider again the grammar:

$$S \to AB|\epsilon$$
$$A \to aAb|aAbb|\epsilon$$
$$B \to bB|\epsilon$$

A leftmost derivation of the string ab is given by:

$$S \implies AB \implies aAbB \implies abB \implies ab$$

Similarly, a rightmost derivation of the string ab is given by:

$$S \implies AB \implies A \implies aAb \implies ab$$

Definition 2 (Language of Grammar)**.** Let $G(N, T, P, S)$ be a grammar. The *language* of G, denoted:

$$L(G) = \{w \in T^* : S \implies {}^*w\}$$

Example 4. Ask students for the language of the following grammar. (**Answer:** $L = \{a^i b^j c^k : i + j = k\}$):

$$S \to aSc|B$$
$$B \to bBc|\epsilon$$

Example 5. Ask students for the language of the following grammar. (**Answer:** $L = \{a^i b^j : i \leq 2j\}$):

$$S \to AB|\epsilon$$
$$A \to aAb|aAbb|\epsilon$$
$$B \to bB|\epsilon$$

1.1 Parse Trees and Ambiguity

The root is labeled by the start symbol. Each interior node is labeled by a variable in V. Each leaf is labeled by either a variable, terminal, or ϵ. If the leaf is ϵ, it must be the sole child of its parent.

Given an interior node with children $X_1, \ldots, X_k$, the production is $A \to X_1 \ldots X_k$.

Consider $E \to E + E|E * E|x$. Give two parse trees for this.

Definition 3 (Ambiguous Grammar)**.** A grammar G is said to be ambiguous if there exists a string $\omega \in L(G)$ with more than one left-most (right-most, respectively) derivation $S \implies {}^*\omega$.

Remark: There are grammars that are inherently ambiguous; that is, every grammar is ambiguous. Deciding if a grammar is ambiguous is impossible (this is undecidable). So removing ambiguity in general is not always feasible.

Example 6. Consider:

$$E \to E + E | E * E | x$$

We have two derivations for $x * x + x$.

$$E \implies E + E \implies E * E + E \implies x * E + E \implies x * x + E \implies x * x + x$$
$$E \implies E * E \implies x * E \implies x * E + E \implies x * x + E \implies x * x + x$$

1.2 Pushdown Automata

Definition 4 (Pushdown Automaton)**.** A pushdown automaton (PDA) is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- Q is the finite set of states
- Σ is the finite input alphabet
- Γ is the finite stack alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$
- q_0 The start state
- Z_0, the start symbol of the stack
- F the set of accepting states.

We pop from the stack, then push back at each transition.

Example 7. $L = \{\omega\omega^R : \omega \in \{0,1\}^*\}$. Our PDA is:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{Z_0, 0, 1\}$
- Z_0
- $F = \{q_2\}$
- For each $a \in \{0, 1\}$, we have $\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$. (Push the initial character onto the stack).
- For each $a \in \Gamma$, define $\delta(q_0, \epsilon, a) = \{(q_1, a)\}$. (Guess where the split is).
- $\delta(q_1, a, a) = \{(q_1, \epsilon)\}$ for each $a \in \Gamma$. (Process ω^R).
- $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$ (After popping everything, move to accept state)

Example 8. $L = \{a^i b^j c^k : i + j = k\}$. Our PDA is:

- $\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$ (Read in all the a's)
- $\delta(q_0, a, a) = \{(q_0, aa)\}$
- $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$ (If no a's, check for b's)
- $\delta(q_0, \epsilon, a) = \{(q_1, a)\}$
- $\delta(q_1, b, Z_0) = \{(q_1, bZ_0)\}$ (Read in b's)
- $\delta(q_1, b, a) = \{(q_1, ba)\}$, $\delta(q_1, b, b) = \{(q_1, bb)\}$
- $\delta(q_1, \epsilon, x) = \{(q_2, x)\}$ for each $x \in \Gamma$ (Finish reading b's in)
- $\delta(q_2, c, x) = \{(q_2, \epsilon)\}$ for each $x \in \{a, b\}$ (pop from the stack for each c)
- $\delta(q_2, \epsilon, Z_0) = \{(q_3, Z_0)\}$ (Finish popping and accept)

Representing as diagrams. Labels input, pop/push.