Automata Theory- Regular and Context Free Grammars

Michael Levet

May 18, 2015

1 Introduction

This tutorial 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. Familiarity with regular languages and finite state automata (`http://www.dreamincode.net/forums/topic/374400-automata-theory-introduction-to-regul` is assumed.

2 II. Context-Free Languages

Recall from my previous tutorial on automata theory that the desired result 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 $\omega \in (N \cup T)^*$ 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.

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:

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

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:

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

3 III. Regular Grammars

This section will introduce the concept of a regular grammar. Recall that regular languages are a subset of context-free languages. The regular grammar is a specific type of context-free grammar, where each regular grammar generates a regular language.

A regular grammar is a context-free grammar $G = (N, T, P, S)$ in which all the production rules are linear. In particular, all rules must be either left-linear or right-linear. In a right-linear grammar, the rules are of the form: $\Gamma \to \epsilon$ or $\Gamma \to a\beta$, where $\Gamma \in N$, $a \in T$, and $\beta \in N \cup \{\epsilon\}$. That is, the head of the production is a single non-terminal symbol, and the body consists of a single terminal symbol and at most one non-terminal symbol. The left-linear grammars simply place the non-terminal before the terminal symbol in the body of the production, so $\Gamma \to \beta a$. The left-linear grammar generates strings from the left to right, while the right-linear grammar generates strings starting at the last character.

Let's look at a couple of examples. The first language is $L = \{\omega : |\omega| \equiv 1 \pmod 2\}$, where the alphabet is $\Sigma = \{0, 1\}$. This language is regular, as a regular expression can be constructed for it. Consider $\Sigma(\Sigma\Sigma)^* = (0 + 1) \cdot [(0 + 1)(0 + 1)]^*$.

The goal is to construct a regular grammar $G = (N, T, P, S)$ for L. Clearly, $T = \{0, 1\}$. Just as in the last section, the first step is to identify the base cases or building blocks for the grammar. As $|\epsilon| = 0$, no rule will terminate on ϵ. Thus, the minimal cases are $0, 1$. As the grammar is regular, a production rule contains exactly one terminal symbol. So at least two non-terminal symbols are needed. Consider $S \to 0A$ and $S \to 1A$ as some production rules. The final step is to use A to ensure an odd number of characters are generated. So really, it suffices to circle around back to S. The production rules, thus, are as follows:

- $S \to 0$
- $S \to 1$
- $S \to 0A$
- $S \to 1A$
- $A \to 0S$
- $A \to 1S$

In the last for rules, swapping the terminal and non-terminal symbols will produce the right-linear regular grammar instead of the left-linear regular grammar. The left and right linear grammars are equivalent.

A second example will now be considered. Let the language $L = \{\omega : \omega \text{ contains at least three } 1s\}$, where the alphabet is $\Sigma = \{0, 1\}$. The regular expression for this language is easy to see: $L = 0^*10^*10^*1(0 + 1)^*$.

The regular grammar for this language follows the regular expression closely. Consider first 0^*1. The rules are $S \to 0S$ and $S \to 1$. However, 0^*1 has to be repeated three times. This requires more non-terminals than just S. Let $N = \{S, A, B, C\}$ with the production rules:

- $S \to 0S$
- $S \to 1A$
- $A \to 0A$
- $A \to 1B$
- $B \to 0B$
- $B \to 1C$
- $C \to 0C$
- $C \to 1C$
- $C \to \epsilon$

Observe that the $S \to A, A \to B$ and $B \to C$ steps handle $(0^*1)^3$ while the production rules with C as the head generate Σ^*.

This grammar can be condensed significantly by using a context-free grammar rather than restricting to a regular grammar. Let $N = \{S, A\}$. Then the context-free representation is:

- $S \to A1A1A1A$
- $A \to 0A$
- $A \to 1A$
- $A \to \epsilon$

4 IV. Regular Grammars to Regular Expressions

This section will introduce converting regular grammars to regular expressions. A common, but ineffective way of approaching this task is to generate some sample strings using the grammar in the hopes of seeing a pattern. The method of judicious guessing is more often than not an exercise in frustration and futility.

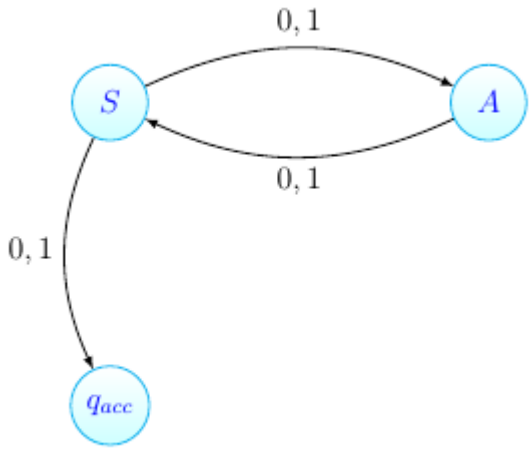
Recall from my previous tutorial that regular languages are accepted by finite state automata. This fact will be used to rewrite the regular grammar as a non-deterministic finite state automaton. This automata will be helpful in garnering a regular expression.

Let's start with an example from above: $L = \{\omega \in \{0, 1\}^* : |\omega| \equiv 1 \pmod 2\}$. Recall that the production rules are:

- $S \to 0$
- $S \to 1$
- $S \to 0A$
- $S \to 1A$
- $A \to 0S$
- $A \to 1S$

Each non-terminal in the grammar will be a state of the finite state automata, with S being the starting state. An accepting halt state will also be added to the finite state automata. In a right-linear grammar, we have rules of the form $\Gamma \to \epsilon$ or $\Gamma \to \alpha\beta$, where $\Gamma \in N$, $\alpha \in T$, and $\beta \in N \cup \{\epsilon\}$. So α represents the character to be parsed by the finite state automata at state Γ and β represents the next state. If $\beta = \epsilon$, then the receiving state is the accepting halt state. Loops and directed cycles represent the Kleene star operation.

So now consider the grammar above. Observe that the rules $S \to 0$ and $S \to 1$ tell the finite state automata to transition to the accepting halt state q_{acc} when a 0 or 1 is read in. Similarly, the rules $S \to 0A$ and $1A$ define the transitions $S \to A$ when either a 0 or 1 is read in. Finally, the rules $A \to 0S$ and $A \to 1S$ define the transition from $A \to S$ when either a 0 or 1 is read in. The finite state automata diagram is below:

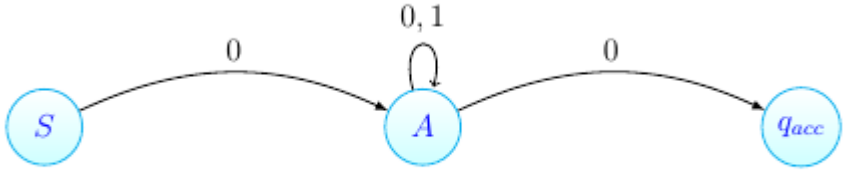


Observe that $0, 1 \in L(G)$. Now any pair of characters read in causes the finite state automata to transition $S \to A \to S$. Only an unpaired third character will allow the finite state automata to transition to q_{acc}. So $L(G) = (0 + 1) \cdot [(0 + 1)(0 + 1)]^*$.

Let's now look at a second example. Consider the grammar given by the following rules:

- $S \to 0A$
- $A \to 0A$
- $A \to 1A$
- $A \to 0$

The finite state automata for this language has three states: $S, A,$ and q_{acc}. The rule $S \to 0A$ defines the transition from $S \to A$ upon reading in a 0. Similarly, $A \to 0A$ and $A \to 1A$ define a loop at state A upon reading in either a 0 or 1. The rule $A \to 0$ ends at a terminal, so $A \to q_{acc}$ must be a transition when parsing a 0. The finite state automata diagram is given below:



Observe that $S \to A$ only when reading in a 0, so any string in the language must start with a 0. At state A, observe that there is a loop when reading in either a 0 or 1, which corresponds to $(0 + 1)^*$. State A can

also transition to q_{acc} after parsing a 0, so any string in the language must end with a 0. Thus, $L(G) = 0(0+1)^*0$.

5 Determining The Language of a Context-Free Grammar

Context-free languages are more complex than regular languages. As a result, it is more difficult to take a context-free grammar and determine a closed-form expression for its language. Regular languages have a simpler machine accepting them- the finite state automata. Context-free languages add a stack to this structure. The stack is useful in matching pairs of data, as was evident in the example earlier regarding the language of balanced parentheses.

Recall the example above where the grammar for $L = \{0^n 1^n : n \geq 0\}$ was provided:

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

Observe in this grammar that each 0 is paired up with a single 1, as evidenced by the rules $S \to 01$ and $S \to 0S1$. In the latter grammar, the S rule recurses. However, S terminates on either ϵ or 01. By this analysis, it follows that $L(G) = L = \{0^n 1^n : n \geq 0\}$.

Now let's look at a more complicated grammar, given by the production rules:

- $S \to aSc$
- $S \to A$
- $A \to bAc$
- $A \to \epsilon$

Let's first consider the language generated by the A rules. Observe that $A \to bAc$ matches a single b character with a single c character, with an A rule in the middle. The A production terminates at $A \to \epsilon$, so $L(A) = \{b^n c^n : n \geq 0\}$.

Now consider the S production rules. Observe that $S \to aSc$ matches each a with c, separated by an S rule. The S production terminates on $S \to A$. Thus, a string from $L(A)$ separates the sequence of a's from the sequence of c's generated by the S rule. So $L(G) = \{a^k \omega c^k : k \geq 0, \omega \in L(A)\} = \{a^k b^n c^{n+k} : n, k \geq 0\}$.