CSE 322 - Introduction to Formal Methods in Computer Science
Introduction to Context-Free Grammars

Dave Bacon
Department of Computer Science & Engineering, University of Washington

In the last few lectures we finished up talking about regular languages and regular expressions. An interesting take home message that you should have taken home with you (and put in a special place on the shelf) is that not all languages are regular. Thus there are language which cannot be recognized by deterministic finite automata. This leads us naturally to ask the question of: well are there other types of machines which will recognize some of these non-regular languages?

To answer this we will begin not with constructing such machines first, but with constructing the languages that ultimately we will show are recognized by some sort of machine. The languages we are going to discuss are called *context-free languages*. These languages are a superset of regular languages. That is all regular languages are context-free. But not necessarily vice versa.

Lets begin with a simple example of a context-free grammar.

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

Okay, so what is this beast, which we will call G? A context free grammar is made up of a collection of *substitution rules*. Each of the lines above is one of these substitution rules. Substitution rules are also sometimes called *productions*. Substitution rules are made up of three parts. The first is a symbol, the second is an arrow, and the third is a string. Sometimes we call the symbol which appears on the left hand side of the substitution rule the *variable*. The variables in the above context-free grammar are A and B. The string consisting of variables and other symbols, which are called *terminals*. Terminals in the above context free grammar include 0, 1, and #. You'll see the reason for this nomenclature in a second. The *terminals* are going to be taken from the alphabet over which the context-free language is constructed. One of the variables in the above specification is also taken to be the *start variable*. By convention, unless otherwise noted, the first variable in the specification will be the start variable. The start variable in the above context-free grammar is A.

To generate strings from a specification of a grammar, you follow a simple procedure:

1. Begin with the start variable. Write it down.

2. Find a variable that has been written down, and a rule which begins with that particular variable. Then replace the variable which you picked with the right hand side of the rule you have also selected. Repeat this process until no variables remain.

Lets do an example for the grammar G above. We start by writing down the start variable A. Now we apply, say, the first rule. This turns the expression into $0A1$. Now there is only one variable to chose, so we choose A. Now let's pick a rule. Suppose we pick the first rule. Then we will end up with $00A11$. Okay, great. Again there is only one variable, A, so we pick it. This time we use the second rule. This turns the expression into $00B11$. Again we only have one variable B. There is only one rule for the B variable, so we apply it and end up with the string $00\#11$. Since there are no variables left, we are done. Thus we say that $00\#11$ is in the language generated by this context free grammar.

The *language of a grammar* is the set of all strings which can be generated by the above process. For example it is easy to see that # is in the grammar, by applying the second rule and then the third rule. A little experimentation will tell you that the language of this context free grammar is $\{0^k\#1^k|k \geq 0\}$. This language is not regular (a quick use of the pumping lemma will verify this.) So we already see that context-free grammars can express languages beyond regular languages.

One convenience that we adopt in denoting a context-free grammar is that for a given variable we denote multiple rules by separating them by a $|$. Thus, the above grammar, which has the two rules $A \rightarrow 0A1$ and $A \rightarrow B$, can be denoted by the line $A \rightarrow 0A1|B$. Think of the $|$ as an "or."

Now one question you might have at this point is a very important one. Why the heck are these things called "context-free"? Good question. Apparently the name comes because the substitutions you are allowed to make do not depend on the context: i.e. on what else is around the variable you are substituting for.

I. FORMAL DEFINITION

Lets get a little more formal. Context-free grammars are going to be specified by a four tuple. Whew. This means we will have to write down less than we did with the DFAs and NFAs and their four tuples. Okay, so this four tuple is (V, Σ, R, S), where these four things are

1. V is the variables. This is a finite set.
2. Σ is the set of terminals. It is a finite set, like V, but is disjoint from V (disjoin means that they share no common elements.)
3. R is a *finite* set of rules. Every rule is made up of a variable and a string of variables and terminals.
4. S is the start variable. It is an element of V.

For example, lets write down the formal definition for the context-free grammar we specified above. $G = (V, \Sigma, R, S)$. $V = \{A, B\}$. $\Sigma = \{0, 1, \#\}$. The set of rules R is made up of $\{A \to 0A1, A \to B, B \to \#\}$. Finally the start variable S is equal to A. So $G = (\{A, B\}, \{0, 1, \#\}, \{A \to 0A1, A \to B, B \to \#\}, A)$.

A few more definitions. If x, y and z are strings of variables and terminals, and $A \to x$ is a rule in the grammar we say that yAz *yields* yxz. We write this as $yAz \Rightarrow yxz$. A further useful word is *derives*. We say that x *derives* y and write this as $x \overset{*}{\Rightarrow} y$ if x is equal to y or can be derived by a sequence of rules, i,e. there exist a sequence $x_1, x_2, \dots, x_k$ such that $x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_k \Rightarrow y$. This is kind of like the δ^* we defined previously. Having defined $\overset{*}{\Rightarrow}$ we can now define the *language of the grammar*. The language of the grammar is the language $\{x \in \Sigma | S \overset{*}{\Rightarrow} x\}$. We often call teh language $L(G)$.

Lets do another example of a context-free grammar. Let $G = (V, \Sigma, R, S)$ be the grammar. Let $V = \{A\}$, $\Sigma = \{x, y, z, +, -, \times, /, (,)\}$. The start state is the only state, $S = A$. The rules are $A \to x|y|z|A+A|A-A|A \times A|A/A|(A)$. What is this language? Well if you play with for a while you will see that it is valid expressions formed from the variables x, y, z. For example, $x + y * z$ is in the language of this grammar.

Finally I'm getting tired of typing context-free grammar so we will denote this by CFG. Further we will denote context-free languages as CFL (not to be confused with the Canadian football league.)

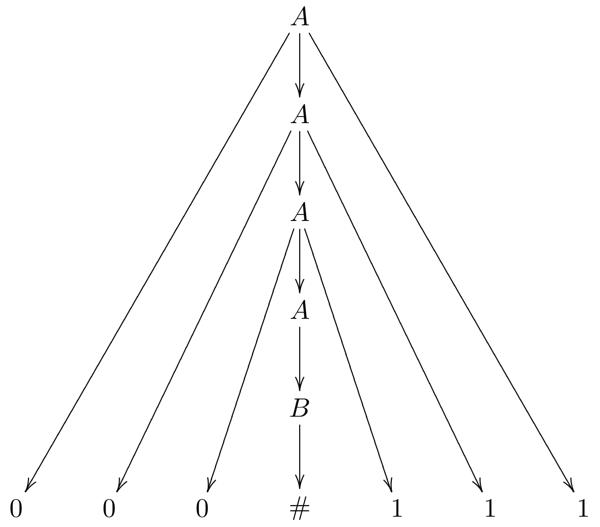
II. CONSTRUCTING CFGS WHICH ACCEPT REGULAR LANGUAGES

Constructing CFGs is a art, a lot like the art of constructing DFAs and NFAs. Indeed it can be a bit harder, because for the later, we had an idea of the computation being carried out, but for CFGs we are constructing a kind of strange set of substitution rules.

Here we will show how to construct a CFG for a language that is regular. Since the language is regular, there exists a DFA which recognizes the language. Suppose this is $M = (Q, \Sigma, \delta, q_0, F)$. The context free grammar we will construct has a variable R_i for every state $q_i \in Q$. The rules consists of $R_i \to aR_j$ if $\delta(q_i, a) = q_j$. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_j with $q_0 = q_j$ the start variable. This CFG has a CFL which is exactly that of the DFA (you should think about why this is true.)

III. PARSE TREES

Recall that above we showed how the CFG G with $A \to 0A1|B$ and $B \to \#$ with start variable A had a derivation $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A1111 \Rightarrow 000B111 \Rightarrow 000\#111$. We can represent this derivation via a *parse tree*. Lets look at the parse tree for this derivation as it will give you a quick idea of what the parse tree is



Note that the leaves of this tree are terminals, and the internal nodes of the tree are variables. Further for any node in the tree which is not a leaf, this node has children which are made up of valid rule substitutions via a rule from the grammar. Further note that parse trees retain an ordering: the children appearing are ordered from left to right and this correspondence is the same as for the rule the node represents.

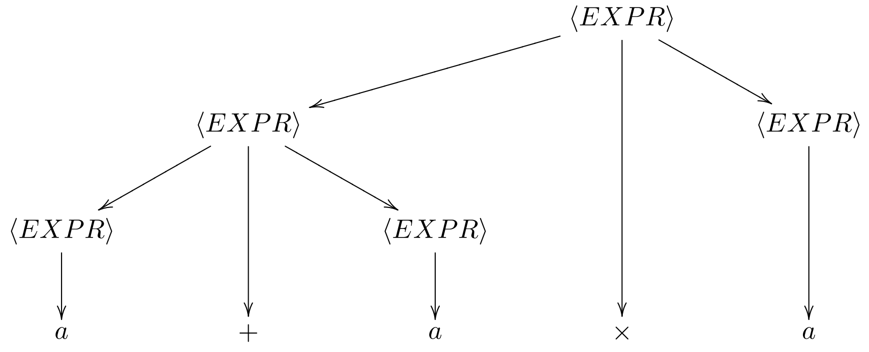
Parse trees are very important in compilers. The reason for this is that the tree structure of the program eases the translation of the program into executables by using the recursive structure of the tree.

IV. AMBIGUITY

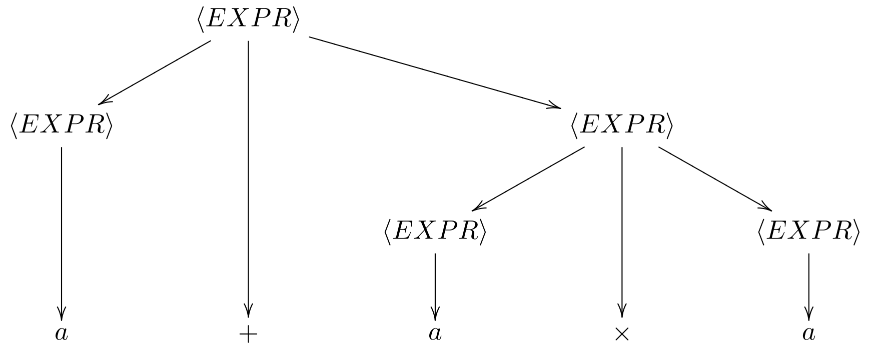
Consider the grammar G:

$$\langle EXPR\rangle \to \langle EXPR\rangle + \langle EXPR\rangle | \langle EXPR\rangle \times \langle EXPR\rangle | (\langle EXPR\rangle) | a \tag{1}$$

Consider the two parse trees for the expression $a + a \times a$:



and



The string $a + a \times a$ is derived *ambiguously* in the grammar. Grammars which generate some string ambiguously are called *ambiguous* grammars.

To formalize the notion of ambiguity, we need to take care of the fact that derivations may differ only in the order in which the variables are replaced. These aren't really ambiguous derivations, since the basic parse tree created from such a derivation is the same. In order to deal with this problem, we introduce the notion of a leftmost derivation (we could have also introduced a rightmost derivation.) A leftmost derivation is one in which at every step in the derivation, the leftmost variable is the one replaced. The leftmost derivation of the first parse tree is $\langle EXPR\rangle \Rightarrow \langle EXPR\rangle \times \langle EXPR\rangle = \langle EXPR\rangle + \langle EXPR\rangle \times \langle EXPR\rangle \Rightarrow a + \langle EXPR\rangle \times \langle EXPR\rangle \Rightarrow a + a \times \langle EXPR\rangle \Rightarrow a + a \times a$, while the left most derivation for the second parse tree is $\langle EXPR\rangle \Rightarrow \langle EXPR\rangle + \langle EXPR\rangle = a + \langle EXPR\rangle \Rightarrow a + \langle EXPR\rangle \times \langle EXPR\rangle \Rightarrow a + a \times \langle EXPR\rangle \Rightarrow a + a \times a$.

It is often possible to take an ambiguous language and find another an unambiguous grammar that generates the same language. However this is not always possible. There are languages that are *inherently ambiguous*. Further you might be interested in whether there is an algorithm for deciding whether a grammar is unambiguous. It turns out, suprisingly perhaps, that there is not algorithm which can determine whether a language is ambiguous. (Note that this is not to say that you cannot determine whether a particular grammar is ambiguous or not: its just that there is no algorithm which takes a description of an arbitrary grammar and computes whether the algorithm is ambiguous or not.)