Lecture 6: Context-Free Languages, Continued

Ryan Bernstein

1 Introductory Remarks

- Assignment 1 is due today
- The midterm will be Thursday of week 5 (April 28th)
- Assignment 2 should be posted tonight, and due the day of the midterm.

1.1 Assignment 1 Solutions

1.2 Recapitulation

Last time, we introduced the class of context-free languages, which are a superset of regular languages. We also introduced two constructions that we could use to describe them. The first was a pushdown automaton, which we created by adding a stack to an NFA. We saw that building PDAs for non-trivial context-free languages quickly grows unmanageably complex.

It's fortunate, then, that we have another means of representing context-free languages: context-free grammars. Grammars are composed of variables, terminal symbols, and "rules" or "productions" that transform variables into strings of variables and terminals.

2 Parsing

We've been checking our work with context-free grammars by generating test strings in a very ad-hoc manner. Now, we'll look at these derivations in more detail.

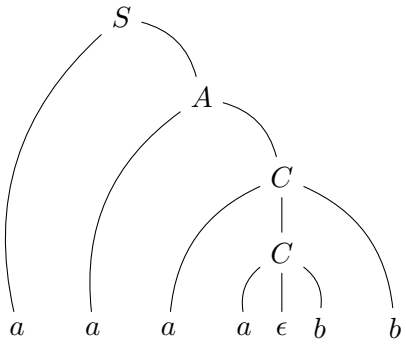
Last time, we represented the language $\{a^i b^j \mid i \neq j\}$ using the following grammar:

$$\begin{aligned} S &\to aA \mid Bb \\ A &\to aA \mid C \\ B &\to Bb \mid C \\ C &\to aCb \mid \epsilon \end{aligned}$$

We'd generate the string $s = aaaabb$ using a series of derivations like the following:

$$S \Rightarrow aA \Rightarrow aaA \Rightarrow aaC \Rightarrow aaaCb \Rightarrow aaaab$$

Representing long chains of productions like this can get unwieldy quickly, especially with less friendly grammars. To address this, we add another dimension and represent productions in a tree structure known as a *parse tree*. A parse tree for the same string would look like this:



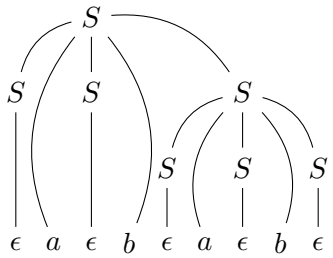
This makes the relationship between each symbol the and variable that produced it immediately clear.

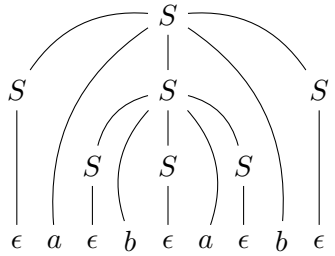
2.1 Ambiguity

Consider another language that we looked at, $B = \{w \in \{a, b\}^* \mid w$ contains an equal number of as and bs$\}$. We represented with a one-line grammar like the following:

$$S \to SaSbS \mid SbSaS \mid \epsilon$$

Since the string $s = abab$ contains an equal number of as and bs, it is a member of B. This time, though, there are actual multiple ways that we can derive this string:





Any grammar that can generate two differently-structured parse trees for the same string in this manner is called an *ambiguous grammar*. Ambiguity is a property of a grammar, not a language. There are languages that contains strings that can *only* be derived ambiguously, and these are called *inherently ambiguous* languages. That, however, is outside of the scope of this class.

2.1.1 The Significance of Ambiguity

All of this brings to mind an important question: who cares? The short answer is "not us". For the purposes of this class, it's not really important whether or not a context-free grammar is ambiguous. As for the long answer, this *will* be important in classes like CS 321, and we'll see why.

Consider the grammar that we used last week to generate all strings of well-formed arithmetic expressions using $\times$ and $+$:

$$E \to E + E \mid E \times E \mid (E) \mid [0 - 9]$$

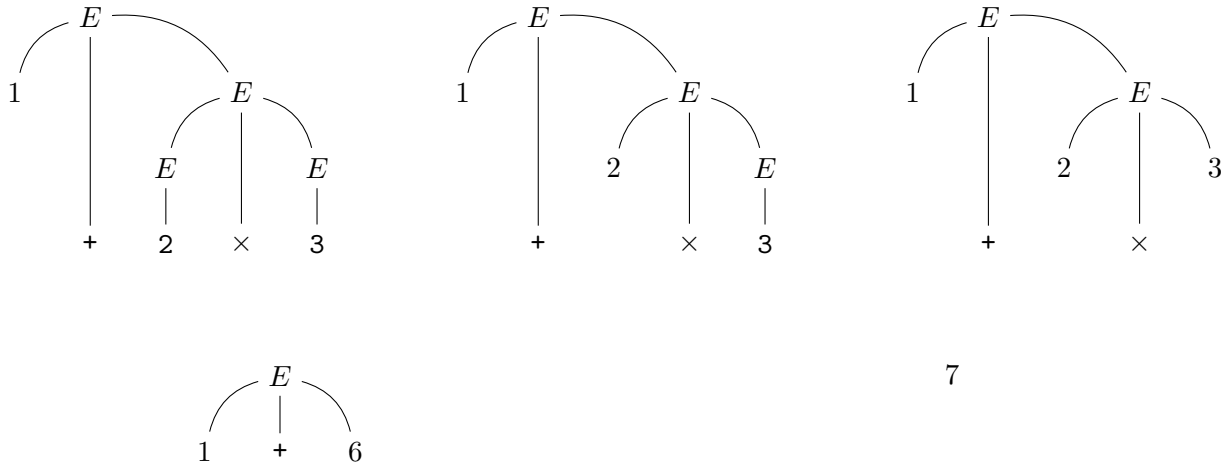
We can use this grammar to generate multiple parse trees for the string `1 + 2 × 3`:

Compilers and interpreters parse expressions like these, but they are responsible for much more than deciding whether or not their inputs are in some language. An interpreter might take an expression like this and compute a numerical result. It does this by taking the parse tree for the expression it reads and turning it into something called an *abstract syntax tree*, which maps each node to some language construct. In the above trees, such nodes might represent:

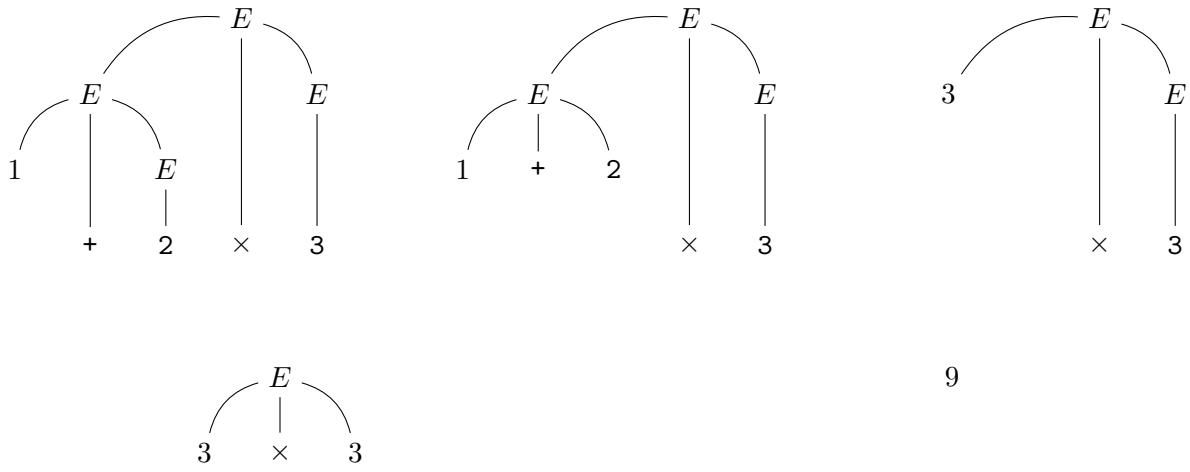
- Constants like `1`, `2`, and `3`, each of which has some associated value
- Additions, which have two subexpressions as children
- Products, which have two subexpressions as children.

We can recursively compute the result of the entire expression by starting at the root, computing the result of any and all subexpressions, and then adding or multiplying the child results as appropriate.

Evaluating the result of the first parse tree then consists of the following computational steps:



The evaluation of the second parse tree looks much different:



9

This is the significance of ambiguity: if the structure of the parse tree has some semantic meaning, as it does when we compile or interpret programs, the ability to generate multiple parse trees from the same programs means that our *programs* are ambiguous, as well as our grammar.

This is a problem with ambiguous grammars, so disambiguating the grammar is actually the solution as well. Things like operator precedence and associativity are actually codified directly in a language's grammar.

3 Closure Properties of Context-Free Languages

In many cases, context-free grammars are also easier to use in constructive proofs than are pushdown automata. We'll illustrate this by quickly proving that context-free languages are closed under the regular operations.

3.1 Union

We want to show that context-free languages are closed under union — that is to say, if A and B are context-free languages, then $A \cup B$ is also context-free.

We know that if A and B are context-free languages, then there are context-free grammars $G_A = (V_A, \Sigma_A, R_A, S_A)$ and $G_B = (V_B, \Sigma_B, R_B, S_B)$ that describe them.

Constructing a new grammar, G_U, to decide $A \cup B$, is simple. We just add a new start variable, S_U, and two productions from it:

1. $S_U \to S_A$
2. $S_U \to S_B$

Then $G_U = (V_A \cup V_B, \Sigma_A \cup \Sigma_B, R_A \cup R_B \cup \{S_U \to S_A, S_U \to S_B\}, S_U)$.

Since we can generate any string in A starting with S_A and any string in B starting with S_B. A new start variable that can generate either of these should be able to generate any string in *either* language, which is any string in $A \cup B$.

3.2 Concatenation

Concatenation is similarly easy. Instead of adding productions from our new start variable to S_A *or* S_B, we can simply add a single new rule $S \to S_A S_B$.

3.3 Kleene Star

Here, we need to generate zero or more strings in some context-free language A. We can do this by generating zero or more repetitions of S_A by adding the following new rule:

$$S \to SS_A \mid \epsilon$$

4 Chomsky Normal Form

While context-free grammars are often easier to create than automata, there is one thing that seems strangely difficult: finding out whether a string can actually be generated by a grammar, or equivalently, whether or not it's part of the language in question. Automata allow us to start with a given string and "test" it by running it through the machine. With grammars, though, we have to start with the grammar and attempt to generate the string. While automata have a very mechanical model of computation, we have yet to see an algorithm that we can apply to a given grammar and string to determine the string's membership in the language of that grammar.

Part of this stems from the loose structure of parse trees. Nodes can have arbitrary *arity* (number of children), which means that the size of the tree and the length of the generated string may not be strongly correlated. This makes determining when we've generated all strings of length $|s|$ or shorter a difficult proposition.

We'll now introduce a simplified form of context-free grammar that makes it possible to determine whether or not a string s is in some grammar G by exhaustively generating all strings of length $|s|$. This is called Chomsky normal form, or CNF.

A grammar is in Chomsky normal form if and only if:

- Every rule in the grammar is of the form $A \to BC$ or $A \to a$ — that is to say, every variable produces either 1) two variables or 2) one terminal.
- The start variable never appears on the right-hand side of any rule
- No variable but the start variable may generate ϵ.

4.1 Conversion to Chomsky Normal Form

We can convert any arbitrary grammar to CNF using the following series of steps. The order is important, since if we ordered the steps differently, the results of later steps might invalidate the results of earlier ones.

Eliminate rules that produce the start variable.

Ensure that the start variable never appears on the right-hand side of a rule by adding a new start variable S_0 and a new rule $S_0 \to S$

Eliminate ϵ-rules

Assume some rule A (that is *not* S_0) produces ϵ. We can eliminate this rule without changing the language of the grammar as follows:

1. Delete the rule $A \to \epsilon$
2. For any rule $B \to uAv$ (where u and v are arbitrary-length strings of terminals and nonterminals), add a new rule $B \to uv$ (i.e. remove the occurrence of A). If A appears multiple times on the right-hand side of some rule, this will not be sufficient. Repeat this step for any and all occurrences of A — that is to say, if there is a rule $B \to uAvAw$, add rules $B \to uvAw$, $B \to uAvw$, and $B \to uvw$.

Eliminate unit rules

For any rule of the form $A \to B$, we add a rule $A \to u$ for every string u of terminals and nonterminals that B is capable of producing (unless u is another unit rule that we've already seen).

Convert remaining rules to the proper form

We want to ensure that every rule produces either 1) two variables or 2) one terminal.

1. For each rule $R = A \to u_1 u_2 ... u_k$ where $k \geq 3$, replace R with a series of rules $A \to u_1 A_1, A_1 \to u_2 A_2, A_2 \to u_3 A_3, ..., A_{k-2} \to u_{k-1} A_{k-1}$.
2. All rules should now produce at most two symbols. If the right-hand side of a rule contains a terminal u, replace u with a new variable U and add a new rule $U \to u$.

Example Convert the following grammar for $A = \{a^i b^j \mid i > j\}$ to CNF:

$$\begin{aligned} &S \to aA \\ &A \to aA \mid CC \qquad\qquad\qquad\qquad \to aCb \mid \epsilon \end{aligned}$$

First, we add a new start variable:

$$\begin{aligned} S_0 &\to S \\ S &\to aA \\ A &\to aA \mid C \\ C &\to aCb \mid \epsilon \end{aligned}$$

Now, we have to remove the ϵ produced by C. Our first step will replace any C on the right-hand side of

a production with ϵ, which bubbles the transition up to A:

$$\begin{aligned} S_0 &\to S \\ S &\to aA \\ A &\to aA \mid C \mid \epsilon \\ C &\to aCb \mid ab \end{aligned}$$

Since we still have an ϵ-rule, we repeat this step:

$$\begin{aligned} S_0 &\to S \\ S &\to aA \mid a \\ A &\to aA \mid a \mid C \\ C &\to aCb \mid ab \end{aligned}$$

Now, we need to eliminate the unit rules. We have two: $S_0 \to S$ and $A \to C$.

$$\begin{aligned} S_0 &\to aA \mid a \\ S &\to aA \mid a \\ A &\to aA \mid a \mid aCb \mid ab \\ C &\to aCb \mid ab \end{aligned}$$

Since S does not appear on the right-hand side of any rule and is therefore unreachable, we'll ignore it going forward. All that remains now is to ensure that the remaining rules produce either one terminal or two variables. First, we add a new rule $D \to Cb$ to use in the right-hand side of our rules that generate more than three symbols:

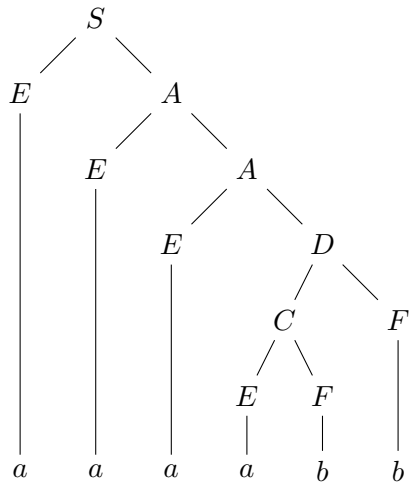
$$\begin{aligned} S_0 &\to aA \mid a \\ A &\to aA \mid a \mid aD \mid ab \\ C &\to aD \mid ab \\ D &\to Cb \end{aligned}$$

Now, we'll add rules $E \to a$ and $F \to b$ to use in any mixed rule that generates a variable and a terminal.

$$\begin{aligned} S_0 &\to EA \mid a \\ A &\to EA \mid a \mid ED \mid EF \\ C &\to ED \mid EF \\ D &\to CF \\ E &\to a \\ F &\to b \end{aligned}$$

4.2 Parsing with CNF Grammars

This has been a lot of work, and we ended up with a larger grammar. What did this gain us? Let's look again at a parse tree for the same string *aaaabb* that we looked at at the beginning of this lecture:



By fixing the arity of each node in the tree, putting a grammar into Chomsky normal form ensures that any parse tree using that grammar is actually a *binary* tree. This makes it much easier to operate on algorithmically, as we'll see after the midterm.