Lecture 7: Non-Context-Free Languages

Ryan Bernstein

April 22, 2016

1 Introductory Remarks

- Assignment 2 is due next Thursday, April 28th
- The midterm is also next Thursday, April 28th. We'll have a review day on Tuesday, so bring questions if you've got them.

1.1 Recapitulation

Last time, we looked in more depth at how we *parse* strings to determine whether or not they're elements of a given context-free language. We introduced the idea of a parse tree, which is a tree structure that we can use to visualize the series of productions that we need to use to transform the start symbol of a grammar into some string. We also saw a simplified form of grammar, known as Chomsky normal form. To be in CNF, a grammar must have the following properties:

- Every rule in R is either of the form $A \to BC$ or $A \to a$ — that is, every production yields either two variables or one terminal
- The start symbol never appears on the right-hand side of any rule
- Only the start symbol may produce ϵ

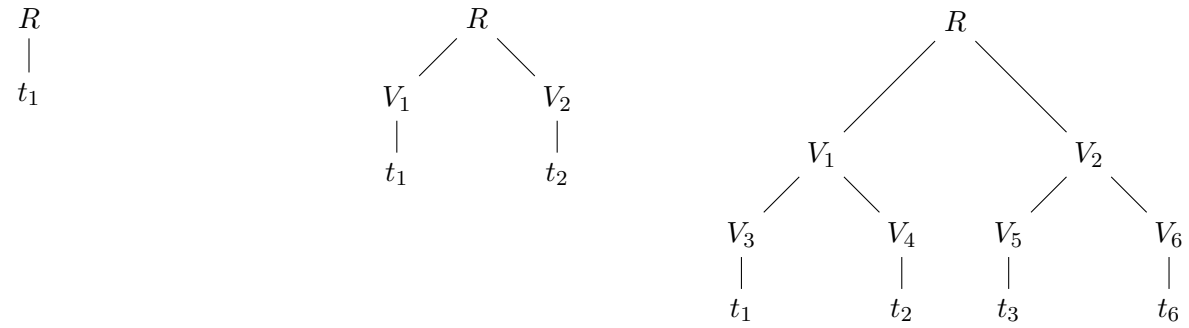
Why is this important? Firstly, any parse tree derived from a CNF grammar is a binary tree. We also know that this tree contains exactly $2|s| - 1$ productions, which means it contains $2|s| - 1$ variables.

Knowing this means that if we generate all possible combinations of length k, we can generate every string in the language of some grammar in k steps. If none of these are equal to s, then $s \notin L(G)$.

2 Breadth and Depth of Parse Trees

Let's talk a bit more about what a parse tree for a CNF grammar looks like. We know that every parse tree built from a CNF grammar is a binary tree, and that every node therefore has either one or two children. Thanks to the requirements of CNF, we also know that any node with one child is producing a terminal, and that this is therefore the end of the path in question.

Let's see what these restrictions mean for path length. We'll now draw binary trees illustrating the maximal number of leaves (i.e. characters in a string) that can be reached from the root node with a depth of d for $d \in [0, 3]$.

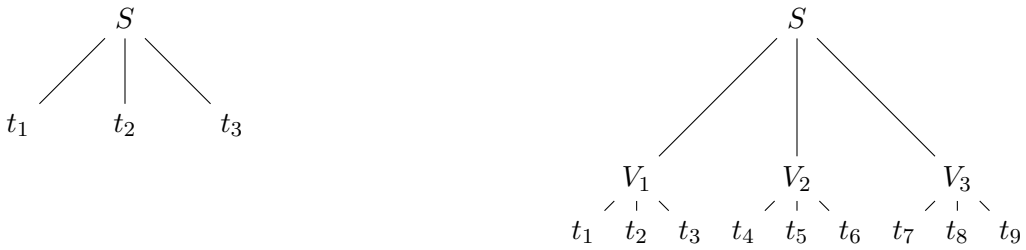


$d = 1$ $d = 2$ $d = 3$

What we're illustrating here is something that you may remember from CS 163: a *complete* binary tree of depth n contains 2^n leaves. Since our parse trees add one additional step to transform "leaf variables" into terminals, we add one to n. A CNF parse tree of depth d is therefore capable of generating strings of at most 2^{d-1} characters.

As it turns out, we can say something similar of *any* parse tree, regardless of whether or not its grammar is in Chomsky normal form. We let b denote the *branching factor* of the tree — that is to say, the maximal number of children that any single node can have. This is the length of the longest string of symbols on the right-hand side of any rule in the grammar. Since grammars that aren't in CNF don't require that rules yield at most one terminal, they don't require the additional step to transform leaf variables to terminals.

To generalize, we can say that a parse tree of depth d is capable of producing strings of length b^d or shorter. The longest string that could be produced in a parse tree of depth 1 would be one in which all b children of the start symbol were variables; the longest string that could be produced by a parse tree of depth 2 would be one in which all b children of the start symbol were variables, each with b terminal children of their own.



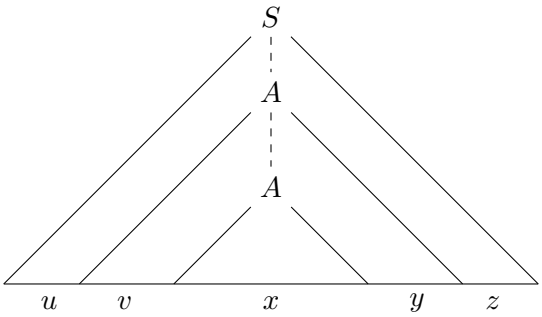
2.1 The Pigeonhole Principle Returns

What does the depth represent in a parse tree? It's the length of the *longest path* between variables in the entire tree. Chomsky normal form gives us a strongly-defined relationship between the length of the longest path in the parse tree and the length of the string being generated.

So if a parse tree with a longest path of depth d can only generate strings of at most b^h, what happens if we need to generate a string with a length longer than $b^{|V|+1}$? We'll *require* some path to be longer than $|V|$. By the pigeonhole principle, this path *must* contain some variable more than once. And since this is a single path, that means that *some variable must be capable of producing itself.*

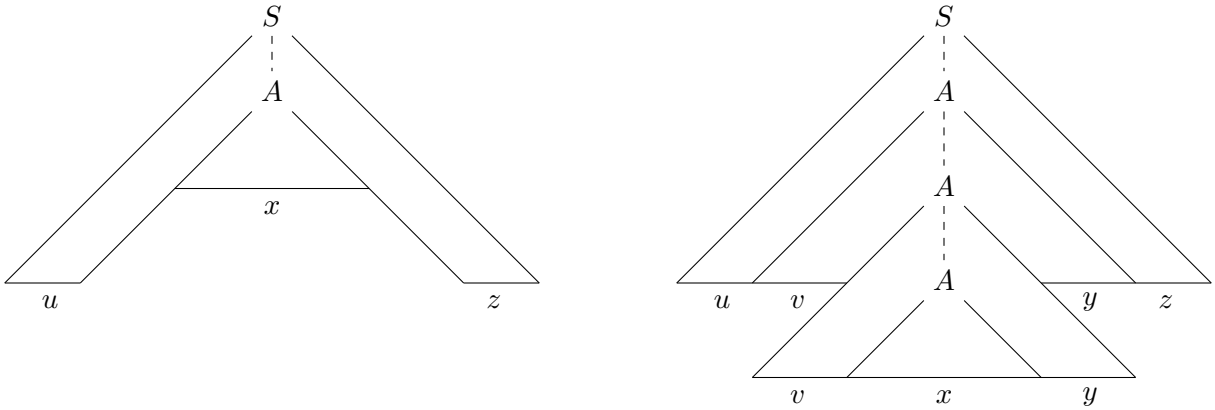
It's at this point that things should start to look distressingly familiar.

Assume that we do have some sufficiently long string s, and that a derivation of s from some grammar G contains a path with more variables than exist in V (remember that we can convert *any* context-free grammar to Chomsky normal form, so this is possible in any context-free language $L(G)$. Then our parse tree for s looks something like this:



In simple terms, some repeated variable A produces something big, which eventually produces another repetition of A. The final such occurrence of A then produces something smaller, which we call x.

Since this is the *same variable*, though, the first occurrence of A has the option of producing x directly. Alternately, the final occurrence of A could produce the same, longer string of variables and terminals as did the first, yielding yet another repetition.



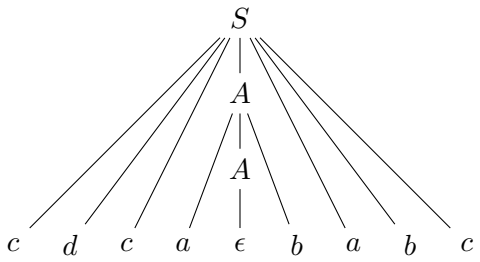
By inductively extending this idea, we see that if the original string $uvxyz$ can be generated by this grammar, so too can uv^ixy^iz for some arbitrary number of repetitions i.

As a concrete example, let's look at the language $L = \{cdca^nb^nabc\}$. We can construct a simple grammar for L as follows:

$$S \rightarrow cdcAabc$$
$$A \rightarrow aAb \mid \epsilon$$

The longest string of symbols on the right-hand side of any rule has length 3, so our branching factor $b = 7$.

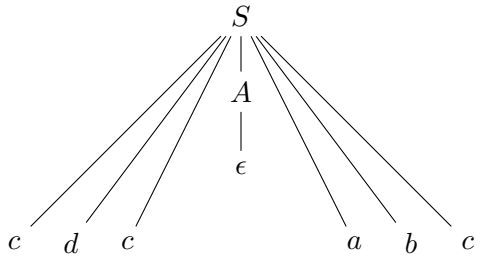
We also know that $|V| = 2$. To *ensure* that we see a repeated symbol, we'd need a string of length $b^{|V|+1} = 7^3 = 343$. Luckily, this is a conservative upper bound. Since all we need is for some symbol to appear multiple times in a single path in our parse tree, we can simply use the string $\{cdcababc\}$:

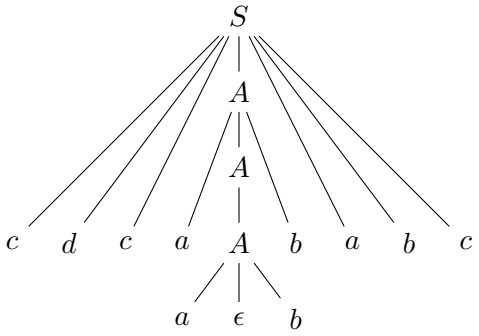


In this case:

- $u = cdc$
- $v = a$
- $x = \epsilon$
- $y = b$
- $z = abc$

By having the first A produce ϵ immediately, we could shorten our string to uv^0xy^0z. Similarly, by having the second A produce aAb and the third produce ϵ, we could produce uv^2xy^2z.





This should be a property that holds true of any context-free language. If we can prove that it does *not* hold for some language L, then we have proven that L is not context-free. This brings us to the context-free pumping lemma. As with the regular language pumping lemma, proving that the context-free *does* hold for some language L is not sufficient to show that L is context-free.

3 The Context-Free Pumping Lemma

For any context-free language L, there exists some length p, known as the pumping length, such that any string $s \in L$ with a length of at least p can be divided into five parts $s = uvxyz$ that satisfy the following

conditions:

1. $uv^ixy^iz \in L$ for any $i \geq 0$
2. $|vy| > 0$
3. $|vxy| \leq p$

Note the following variations that are possible under these constraints:

- While either v or y may be empty, we know that they are not *both* empty.
- Any or all of the substrings u, x, and z may be empty.

3.1 Using the Context-Free Pumping Lemma

Our strategy for choosing strings for the regular language pumping lemma was to choose some s that was restrictive enough that we knew something about the form of y. When pumping 0^n1^n, for example, we chose the string 0^p1^p. Since $|xy| \leq p$ (that is to say, y had to occur within the first p characters), we knew that y had to be composed entirely of zeros.

Because context-free grammars can generate strings from the middle out, though, this is now somewhat more difficult. We know that $|vxy| \leq p$, but this substring is no longer attached to the beginning of s. We must therefore consider vxy as a "sliding window" with a length of at most p and consider all possible locations within s. Often, this means using a proof by cases.

Often, we want to choose some string s with sections of length p or greater. This means that our sliding window can affect at most two adjacent sections at once.

Example 1 Show that $A = \{0^n1^n0^n \mid n \geq 0\}$ is not context-free.

Assume that A is context-free. Then there exists some length p such that any string in A with a length of at least p can be split into five parts $s = uvxyz$ that satisfy the conditions of the context-free pumping lemma.

Let $s = 0^p1^p0^p$. Since $|xyz| \leq p$, we can divide the possibilities for $uvxyz$ into three cases.

Case 1: v and y are both composed entirely of zeros

> We know that v and y are composed entirely of zeros. Since each section of the string has length p and $|vxy| \leq p$, we know that vxy is contained entirely within one of the end sections. It cannot affect both of them.
>
> If we let $k = |vy|$, "pumping up" to $uvvxyyz$ therefore yields a string of the form $0^{p+k}1^p0^p$ or $0^p1^p0^{p+k}$. Neither of these are an element of A.

Case 2: v and y are both composed entirely of ones

> Since v and y are composed entirely of ones, "pumping up" to $uvvxyyz$ causes the center section to be longer than the end sections. If $|vy| = k$, then $uvvxyyz$ is of the form $0^p1^{p+k}0^p$, and is therefore not an element of A.

Case 3: v is composed entirely of one character and y is composed entirely of another character

In this case, we can expand at most two sections. Since vxy has a length of at most p, it's actually not long enough to reach all three sections. Consider the following example, in which we arbitrarily assume that $p = 4$:

$$0000\underbrace{1111}_{vxy}0000$$

Because $|vxy|$ is bounded from above by p, even if it begins on the very last character in the first p zeros, it can't affect all three sections at once.

This means that in the case in which v is composed entirely of one character and y entirely of the other, we'll be expanding (or contracting) two sections at once without affecting the third.

Case 4: v or y contain some combination of zeros and ones

In this case, $uvvxyyz$ will contain some sequence of zeros and ones out of order.

Since $uvxyz \in A$ and $uvvxyyz \notin A$ in all of the above cases, the context-free pumping lemma does not hold, and A is not a context-free language.

Example 2 Show that $B = \{1^{n^2} \mid n \geq 0\}$ is not context-free.

Assume that B is context-free. Then there exists some length p such that any string in B with a length of at least p can be split into five parts $s = uvxyz$ that satisfy the conditions of the context-free pumping lemma.

Let $s = 1^{p^2}$. Since s is composed entirely of ones, we already know for a fact that v and y are also composed entirely of ones.

If we let $k = |vy|$, then $uvvxyyz = 1^{p^2+k}$. We know also that $0 < k \leq p$, which means that $p^2 < p^2 + k \leq p^2 + p$. If we order the strings in B by length, the next element is $1^{(p+1)^2} = 1^{p^2+2p+1}$. Since $p^2 + k < p^2 + 2p + 1$, $uvvxyyz \notin B$.

Since $uvxyz \in B$ and $uvvxyyz \notin B$, the context-free pumping lemma does not hold, and B is not a context-free language.

Example 3 Show that $C = \{ww \mid w \in \{a, b\}^*\}$ is not context-free.

Assume that C is context-free. Then there exists some length p such that any string in C with a length of at least p can be split into five parts $s = uvxyz$ that satisfy the conditions of the context-free pumping lemma.

Let $s = a^p b^p a^p b^p$. We can break the possibilities for vxy into three cases:

Case 1: vy contains only as or only bs

In this case, we're mutating only one of our four sections. If we "pump up" either of the first two sections, then dividing the resulting string in half results in one string that begins with an a and one string that begins with a b. If we "pump up" either of the latter two sections to $i = 2$, then this yields a string that ends with an a and a string that ends with a b.

In neither case can these be identical, so $uv^2xy^2z \notin C$.

Case 2: vy contains as follows by bs

Since we have a window of at most p characters that lies on a boundary between as and bs, we know that our window vxy is affecting only one half of s. "Pumping up" to $i = 2$ causes one half of the string to transition between a and b more often than the other. The result is therefore not in C.

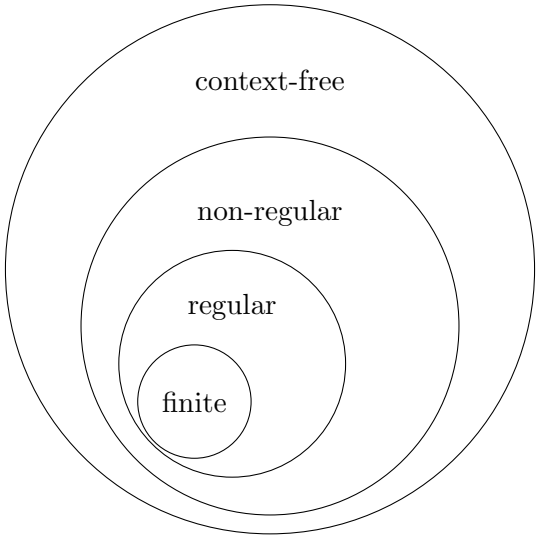
Case 3: vy contains bs followed by as

In this case, our window vxy is on the border between the first and second occurrences of w. Pumping up with $i = 2$ will affect the back half of the first w (the bs) and the front half of the second w (the as). Since there's no possible way for this to mutate both occurrences of w in the same fashion, we know that the result will not be of the form ww.

Since $uvxyz \in C$ and $uv^2xy^2z \notin C$, the context-free pumping lemma does not hold, and C is not a context-free language.

4 Language Identification

We've now added a ring to our Chomsky Hierarchy of Languages:



When we introduced non-regular languages, we referred to the regular languages as "languages that can't count". We addressed this by adding a stack to an NFA to allow counting.

Following from that description, we can say that context-free languages are "languages that can count zero or one thing". This is why the context-free languages are not closed under intersection: the languages A and B each count zero or one things, so if both PDAs utilize their stacks, we may be required to count *two* things. Since we have only one stack, this is not possible using a PDA.

When given some arbitrary unknown language L — say, in an exam situation — we've seen strategies that we can use to *prove* that L belongs in a given ring in this hierarchy. But where do we start? What intuition do we use to determine which proof strategy to attempt in the first place? Typically, I ask myself the following questions:

- Is this language finite? If so, it is regular.

- Does this language require us to count something? If so:
  - Does it require us to count only one thing? If so, it is likely context-free
  - Does it require us to count more than one thing simultaneously? If so, it is likely non-context-free.
- Does this language require us to know where the exact center or midpoint of our string lies? If so, it is likely context-free
- Does this language require us to remember the order of the characters we've seen? If so:
  - Does it require us to replicate that order in reverse? If so, the stack-based nature of a pushdown automaton may make this language context-free
  - Does it require us to replicate that order exactly? Since we can only push to/pop from the top of a stack in a PDA, the language is probably non-context-free.