Craig Silverstein
Gates 470
Phone: (72)3-1787
Office Hours: Wed 4:30–5:30, Fri 1–3, Gates 470
Section: Monday 2:15–3:15, 380-380W

Computer Science 154
School of Engineering
Stanford University

Section Handout #3

The Pumping Lemma

The Pumping Lemma formalizes the idea that finite automata "can't count." Or rather, that they can only count up to a finite, predetermined number (that number is the maximum length of the x part of the string). Furthermore, it formalizes the fact that if a finite machine accepts an infinite number of strings, it does so because of loops in the machine (that's the n part of the y^n of the string).

Here's the way to think about the Pumping Lemma. You're trying to prove the language isn't regular, and the adversary, being adversarial, is trying to prove it is. To do this, the adversary claims he has a machine that accepts your language. He won't show you the machine, but he'll tell you what it does on a given input. You then catch him in a trap, showing him his machine must also accepts some other string not in the language. Sucks to be him![1]

So here's the give-and-take that goes on. There are a lot of parts, but taken one at a time it's not so bad:

1. **The adversary tells you k, the limit on $|xy|$.** This happens first, so you can pick your string knowing how big $|xy|$ is going to be. In reality, k is related to the number of states in the adversary's machine.

2. **You pick a string w to "pump."** By telling you k, the adversary has told you the size of his machine. You can use this fact to pick a string so big that it must use one of the machine's loops. (This implies your language has to be infinite. Why?)

3. **The adversary divides up w.** He divides it up into 3 parts based on how w acts in the machine: x, which has no loops, y, which ends at the same state x ends at and thus completes the first loop, and z, everything after the first loop. The adversary *won't* tell us what x, y, and z are, but since he told us k we know the maximum size for xy and we know y is not empty.

4. **We show that, for any division of w, we can "pump" y to get a string not in L.** This is the crux of the proof. There are only so many ways the adversary can have divvied up w (on the order of $k^2/2$ – $|xy|$ can take on k possible values, and given that, $|y|$ can take on $|xy| - 1$ possible values). We don't know which one he picked, but we know he picked one. So if we can show that for *all* possible divisions, we can pump y to get a string not in the language, then we've shown his machine is a fraud.

This last step may seem like a lot of work: you have to try to pump k^2 different strings. But usually one general method works for most or all strings. Consider the following, silly pumping proof that $\{a^n b^n\}$ is not regular:

> Given k, pick $w = a^{k/2}b^{k/2}$. Then the adversary can divide up w in one of three ways:

[1] Adversaries can also be female.

1. x is all a's and y is all a's. (So $|xy| < k/2$.) In this case, $xy^{100}z$ will have many more a's than b's and thus isn't in L.
2. x is all a's and y is $a^i b^j$, $i, j > 0$. (So $|xy| > k/2$ but $|x| < k/2$). Then xy^2z will look like $a \cdots aa^i b^j a^i b^j b \cdots b$. This isn't of the form $a^n b^n$ at all since a's follow b's.
3. x is $a^{k/2} b^j$ and y is all b's, $j \geq 0$. (So $|xy| > k/2$ and $|x| \geq k/2$.) Then $xy^{100}z$ will have many ¿ more b's than a's and thus isn't in L.

No matter how the adversary divides z, we can pump to get a string not in L.

Note that this argument won't work for the language $\{a^n b^m : m > n\}$ because then the third way of pumping is ok. However, if we choose w more intelligently, we can show that $\{a^n b^m : m > n\}$ is not regular. There's a take-home lesson here: **You can't use the Pumping Lemma to prove a language is regular**.

Context-Free Grammars

Short intro: The Context-free Grammar is more powerful than the regular expression: it can, for instance, express the language $a^n b^n$. ("Context-free," by the way, means that the left-hand side can only have one symbol, so a rule is applied without context around the left-hand side). CFGs still can't "count," but they can "match." This means they can handle $a^n b^n$, which only requires the letters match in number, and even $\{a^i b^j c^k : i + j = k\}$, since it's still a job of matching the a's and b's to the c's. Likewise $\{a^i b^j : i \leq j\}$ is a piece of cake, since it's just $a^n b^n b*$. But CFGs can't handle $\{a^p : p \text{ prime}\}$, which requires division.

A CFG is formally expressed — much like DFAs and NDFAs — by a tuple with alphabets, start conditions, and transition rules (functions). Informally, we just list the transition rules, letting the alphabets and start symbol be determined by context (as it were): the alphabet is all the symbols we see, while the start symbol is the left-hand letter on the first rule.

The way you use these things is you have a string, and you want to figure out what rules to apply to get this string. It's not like FAs where a machine reads the string and takes actions accordingly. The problem is that, at any given time, many possible rules may be applicable (even if you follow the left-nonterminal-first maxim) and you may not know which one is the right one until much later. Sometimes, two different rules will both lead to your string! (Grammars that allow this are *ambiguous*. We computer scientists hate ambiguity.)

Hints on Homework 2

(1) Note that even though all states are accepting states, the language is *not* Σ^*, because many arrows are omitted.

(2) I shouldn't tell you this, but you would be much better off converting the RE $((01 \cup 10) \cup 11)^*$ instead of the one given in the problem set. What's the difference? Ah, there's the rub!

(3) gcd is "greatest common denominator." The gcd is 1 if no number smaller than i and j divides both i and j. This is a simple application of the Pumping Lemma.

(4) If you're willing to spend exponential time, this problem is quite easy. Try to find an algorithm that's linear in the size of the output word. Treat the DFA not as a machine but as a directed acyclic graph. How much time does your algorithm take, in terms of $|x|$, $|\Sigma|$, and $|Q|$?

(5) The key fact about DFAs used here is that they are "memory-less": if you are in state s looking at the last k characters of x, whether the string is accepted depends only on s and $x_{n-k+1} \ldots x_n$. It *doesn't* depend on the previous characters of x at all. Thus, if you need to distinguish between two strings that have the same last k characters (accept one but reject the other), they had better end up in different states after reading $n - k + 1$ characters.