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 #4

Push-down Automata

States are almost irrelevant in Pushdown Automata; what is key is the stack. Stacks can count arbitrarily high (by putting a mark on the stack each time you see something and taking the mark off when the thing you saw is cancelled out). However, they can only count one thing at a time. To keep track of a small, finite amount of associated information (e.g. whether the number on the stack is supposed to be positive or negative) you can use states. So you can represent objects by the ordered pair (s, stk), where s is a state (and has a finite number of values) and stk is the stack (and has a possibly infinite number of values).

Nobody uses PDAs in the real world: they are important merely from a conceptual point of view. Therefore, don't spend too much time trying to figure out how to get a PDA to accept a language you want it to accept. Instead, express the language as a grammar (that *is* a useful skill to cultivate) and use the construction given in class to convert the grammar to a PDA.

Prof. Boneh mentioned PDAs must be non-deterministic to be equivalent to CFGs, but he never gave an example of where this is needed. The palindrome language is the typical one given. For a PDA accepting the palindrome language (that is, reads the same forwards and backwards), we need two states: one for when we read the first half of the string — when we push what we read onto the stack — and one for when we read the second half of the string — when we pop the stack and check it against the word. But we don't know where the middle of the string is as we're reading it; we have to non-deterministically "guess" when we're at the midpoint and should change states. We can implement this by running the PDA $|w|$ times, changing states at a different time each run.

The last-in first-out quality of the stack is why we could recognize $\{ww^r\}$ (the palindrome language). This should hint to you that $\{ww\}$ (the double-word language) is not context-free.

Instantaneous Description

Nothing complicated here. This is what is meant by the real-world (as opposed to technical) use of the word "state": an ID encapsulates the state of the world. For DFAs the ID was merely what state you were in and where you were in in the input string. For PDAs it's also the contents of the stack. (For real computers it would be the contents of the main memory and disk, the status of the various peripherals, perhaps what applications are running, etc. As the list gets longer, it becomes more apparent why we work with idealized versions of computers.)

The CFG Pumping Lemma

Ugh. You will not encounter an uglier theorem this course. Luckily it's much less important than the pumping lemma for regular languages, since it doesn't really introduce any new ideas. A reminder of the theorem: 1) adversary gives you m, 2) you pick w, 3) adversary divides w into five (!) parts $uvxyz$ such that $|vxy| < m$ but vy is not empty, 4) you pump uv^ixy^iz to get a string not in the language, 5) you gloat that the language isn't context free after all.

The way to think of the difference between this and the regular-language form: for RLs, we concentrate on what happens at the beginning, while for CFLs we concentrate on what happens in the *middle*.

Recall how the (regular) pumping lemma was proved: some states happen, then some states forming a loop happen (these are the ones you pump), and then random things you don't care about happen. For the context-free pumping lemma, we have to deal with the extra power of CFGs, namely that a rule can add stuff to both the beginning *and* the end of a string, not just the beginning. So we think of letters as working inside out: First some rules are followed, yielding the x in the middle of $uvxyz$. Then some rules are followed in a cycle (that is, we expand A, and then later we expand A again), yielding the vxy (the v and y we just added are what we pump). Then some other stuff we don't care about happens, yielding the u and z parts that frame w.

Likewise, when trying to pump context-free languages, you should think about pumping from the middle, not from the beginning. That is, instead of picking ws where the beginning is long, pick ws where the middle is long, if you can. Having the beginning long doesn't do anything, since the adversary can swallow up the beginning in u, which can be arbitrarily long.

Doing Proofs

As we approach the midterm, proof discussions may have a few more concrete applications than they did earlier in the course. Two issues I will discuss here:

Proving machine correctness: There are two issues involved here: proving that a machine (or grammar, or regular expression) produces all strings in the language, and proving that it produces only strings in the language. The first part is usually the easier: you give a series of derivations for your machine that generates a string in the grammar. Usually, you will use induction to show that if a few basic strings can be generated, then the rest of the strings in the language can as well through application of the same rules. For instance, if you have a grammar for $a^n b^n$, you would show how to generate ab, and then use induction to show you can generate any string $a^i b^i$, for all i. To prove that your machine doesn't generate extraneous strings, you must categorize all the types of "wrong" strings (e.g. for the even palindrome language, it would be strings that have an odd length and strings whose i^{th} character differ from their $n-i^{th}$ for each i). Then you must argue that your machine cannot produce strings of these illegal forms.

Proving by induction: If an object is finite but arbitrarily large, induction is appropriate. If an object is infinite, induction won't cut it: intuitively you can induct as high as you want, but not forever. So far, the only infinite things we have studied have been languages; rules in grammars, strings, and machines may all be as large as you like, but are always finite.

Hints on Problem Set 3

(1) is just as easy as you think. Make sure you read the description of the language right: this isn't the palindrome language!

(2) Much easier than the gcd pumping problem. You're going to have to deal with many cases though: vxy is before the #, vxy is after the #, vxy includes the # (and then, is it in the x part or in a pumped part?) Each case in itself is fairly easy, though if you pick the wrong w it may seem hopeless. Remember, "substring" means "$w_1 \dots w_n$ occur consecutively in x."

(3) You can (and should) do this problem without finding L. Hint: this problem uses your knowledge of closure properties much more than your knowledge of CFGs. Remember the closure properties we know about CFGs: closed under union, concatenation, repetition, kleene star, but not closed under intersection (remember the counterexample?).

(4) DFAs have states, and PDAs have states and stacks. Where have we tried to combine state sets before? Why do you think we run into problems when we try to combine two PDAs?