Problem Set 2 **Yafim Landa**
18.404 Fall 2010 **July 10, 2011**

Problem 1

Let G be a CFG in Chomsky normal form and assume that G has k variables.

Claim 1. *If $L(G) \neq \emptyset$ then there is a string $w \in L(G)$ where $|w| < 2^k$.*

Proof. Recall from problem set 1 that if a grammar G written in Chomsky normal form then any derivation of a string w has $2|w| - 1$ steps. Let's try to make a string whose length is $< 2^k$ starting from a longer string. We'll do this by looking at its derivation.

Assume that $|w| \geq 2^k$. Then any derivation has length $\geq 2 * 2^k - 1 = 2^{k+1} - 1 > k + 1$. Note that the derivation must end with a literal. By the pigeonhole principle, if we have k variables and the path length is $> k + 1$, then there must be a repeating variable. Cut from the later appearance of this variable to its earlier appearance, and repeat this procedure until $|w| < 2^k$. We have now reduced a long path to a shorter path (much like the pumping lemma), arriving at our initial claim. □

Problem Set 2 **Yafim Landa**
18.404 Fall 2010 **July 10, 2011**

Problem 2

Claim 2. $C = \{w\#w^R\#w | w \in \{0,1\}^*\}$ *is not a CFL.*

Proof. Assume that C is a CFL. Then, by the pumping lemma we can find a string w whose length is greater than the pumping length p that can be split into $w = uvxyz$.

Take the string $w = 0^p1^p\#1^p0^p\#0^p1^p$. This string is more than p characters long, so it could be split into $w = uvxyz$, where $u = 0^p1^p$, $v = \#$, $x = 1^{p-2}$, $y = 1$, $z = 10^p\#0^p1^p\#$.

Pump i down to 0. This yields the string $0^p1^p1^{p-1}0^p\#0^p1^p$, which is not in the language. Therefore, the conditions of the pumping lemma do not hold and we have a contradiction, and so C is not a CFL. □

Claim 3. *A left-reset Turing machine recognizes the same class of languages as a Turing machine.*

Proof. A left-reset turing machine (LRTM) behaves like a Turing machine, except that it has a RESET move instead of a LEFT move. We will show how to simulate a Turing machine's LEFT move with an LRTM. Therefore, if we ignore an LRTM's RESET move and use our simulated LEFT move, an LRTM simulates a TM, and so the two must recognize the same class of languages. The idea is to replace the LEFT operation with a SHIFT-TAPE-RIGHT operation.

Please note that since the machine can reset to the left, there must be a left edge of the tape. Let's call this position 0. We begin with the head at some position k, and want to write symbol σ at this position, and move the head to position $k-1$.

To do so, we will use the following algorithm. We will mark the current symbol with a dot and RESET. We can now shift all of the symbols to the right as follows. We remember the 1st symbol, and replace it with a #. Then we remember the second symbol, and replace it with the first symbol. We then remember the third symbol, and replace it with the second symbol, and so on: at each step, we remember the current symbol, and overwrite it with the previous (remembered) symbol, and move right. [A minor clarifying detail: we can remember symbols just by adding $|\Sigma|$ states to our state machine.] Now, when we get to a dotted symbol, we write the symbol in memory, only with a dot. We then remove the dot from the next symbol. When we've rewritten, we reset and move to the symbol with the dot. Please note that when we reset, we need to ignore any starting #s. We now need to perform this operation *again* in order to move to the left again. We now have the head on a dotted character two positions left of the starting character. Undot this character and move right, and we're done. There is a corner case where we begin at the left edge of the tape. This is handled by dotting this character, RESETing, and observing that we're still on a dotted character. We then RESET and remove the dot. This concludes the LEFT-equivalent procedure using RESET.
Here's an example (underlined is the position of the head):

$abc\underline{d}e$ the starting 4th position
$\underline{a}bc\dot{d}e$ we've dotted the current character and RESET

$\#ab\dot{c}de\_$ perform the replacing procedure and end with a dotted 3rd character
$\#ab\underline{\dot{c}}de$ move to it

$\#\#a\underline{\dot{b}}cde$ do this whole thing again and end up at the 2nd character
$\#\#ab\underline{c}de$ move right and restore the second character

□

Problem Set 2 **Yafim Landa**
18.404 Fall 2010 **July 10, 2011**

Problem 4

Claim 4. *A language is decidable $\iff$ some enumerator enumerates the language in lexicographical order.*

Proof. Let us first prove that a language is decidable if some enumerator enumerates the language in lexicographical order.

We have an enumerator E that enumerates a language L in lexicographical order (meaning that shorter strings are enumerated before longer strings). We want to use E to create a Turing machine M that is a decider for L.

Our TM M receives an input, w, which must then be decided whether it is in L. M must first find the length $|w|$ of w. Look at the output of E. Compare each string e that the enumerator outputs with w: if they're the same, accept. Otherwise, if $|e| > |w|$, reject, or keep reading the outputs of E otherwise. To avoid looping forever if the enumerator only prints strings that are shorter than $|w|$, We must also modify the enumerator to have a state that it goes into when it is done, so that the TM M could check it.

Now, let us prove that an enumerator enumerates a language in lexicographical order if that language is decidable.

We can construct such an enumerator E using a Turing machine M that decides our language L. E ignores the input, and uses M to filter out the accepted strings. For each $i = 1, 2, ...,$ the Turing machine generates a lexicographically sorted list of strings $\in \Sigma^*$ and runs decider M on each of the inputs. E then prints the strings that M accepted. □

Claim 5. *C is Turing-recognizable $\iff$ a decidable language D exists such that $C = \{x | \exists y(< x, y > \in D)\}$*

Proof. First let's prove that if there exists a decidable language D (with decider TM_D), then we can construct a recognizer (Turing machine TM_C) for C as follows. For a given input string x, try all possible values for y, and have TM_C use TM_D to decide on $< x, y >$. If it accepts, then TM_C accepts. Otherwise, it continues on with the next value of y. This is our recognizer.

Now let's prove that if C is Turing-recognizable, then there is a language D such that $C = \{x | \exists y(< x, y > \in D)\}$. We shall do so by constructing D as follows. Consider y to be the sequence of transition function calls that must be made for the TM for language C (TM_C) to accept x. D can decide on $< x, y >$ by running TM_C on x using the sequence of transition function calls y. If this is a valid sequence and x is accepted, then D accepts. Otherwise, D rejects. We have just constructed D, thus completing the proof. □

Problem Set 2 **Yafim Landa**
18.404 Fall 2010 **July 10, 2011**

Problem 6

A variable A in CFG G is *usable* if it appears in some derivation of some string $w \in G$.

We want to formulate as a language the problem of testing whether some variable A in G is usable, and then show that this language is decidable.

Let us invoke the Church-Turing thesis and write a simple algorithm to check whether A is usable. The grammar defines a directed graph, where the variables and literals are nodes, and the productions are the directed edges. We run a breadth-first graph-search algorithm to search for A. This algorithm (and the associated TM) terminates after it has searched the connected component that includes the start variable. Note that it will not loop, since a graph-search algorithm marks nodes and does not return to them twice. If A was found (i.e. it was marked by the algorithm), accept. Otherwise, reject. The language for this TM would be the graph description (the nodes and the edges), and the TM would perform a search on the graph similar to the way it is described in Example 3.23, only with directed edges instead of undirected edges.

Problem 7

We construct a push-down automaton that accepts D to show that D is a context-free language. The push-down automaton goes through its input in state "Collecting" w and pushes the symbols it sees onto its stack. It then nondeterministically branches to a "Matched" state where it pops symbols off of its stack. It stays in the "Matched" state unless it reads a symbol off the tape that does not correspond to the symbol on the stack, which is when it moves to an "Unmatched" state. If the stack is empty by the time it reaches the end of the string, and it is in the "Unmatched" state, then it accepts. Otherwise, it rejects.