Name: ________________________________

Student ID: ________________________________

CS181 Winter 2012 - Final Exam

Due Friday, March 16, 2012, at 4:00pm, in Royce 362

- This exam is open-book and open-notes, but any materials not used in this course are prohibited, including any material found on the internet. Collaboration is, of course, also prohibited. **Please avoid temptation by not working on the examination while you are in the presence of any other student who has taken or is currently taking CS 181.** If you have any questions about the exam, **ask the TA or Professor Sahai. Do not ask other students.** You are allowed to use any theorem shown in class or in the textbook, as long as you clearly cite it.
- We suggest that you spend approximately 12 hours (not necessarily contiguous) to take this exam. Start early so that you have time to understand and think about the problems. It must be turned in by 4:00 P.M on Friday, March 16, 2012. Please turn it personally to the TA from 2-4pm at Royce 362. You may also turn it during the TA office hours on Thursday, March 15, 2012 from 11-1 at BH3714. If you need an alternate slot, email us to schedule one.
- Place your name and UID on every page of your solutions. **Please use separate pages for each question. All problems require clear and well written explanations.**
- For each part (except for the extra credit), 20% of the points will be given if instead of an answer, your write "I don't know".

Honor Code Agreement: I understand this exam is open-book and open-notes, but any materials not used in this course are strictly prohibited. I also understand that this exam is to be taken individually without any outside help (except possibly from the professor or the TA) within the time limits set forth. I agree to adhere to the course honor code and if I am unsure of any rules of the honor code, I will ask for clarification from the professor or the TA.

Signature: ________________________________

Question	**Points**	
1		45
2		45
3		40
4		45
EC		40
Total		

Name: ______________________________

Student ID: ______________________________

1. **2-Counter Automata.** A 2-counter automata is a DFA augumented with two counters. At the start of computation, the counters begin at 0. A 2-counter automata may consult the value of its of counters before making a state transition. At each transition, the automata may also add 1, subtract 1 or do nothing to one or both counters. Formally, a 2-counter automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where all components except δ are identical to a DFA. The transition function δ is defined as $\delta : Q \times \mathbb{Z} \times \mathbb{Z} \times \Sigma \to Q \times \{+, -, 0\} \times \{+, -, 0\}$ which defines transitions from a particular state, counter values and input alphabet and describes the modifications to the counters. As an example, $\delta(q_5, -2, 5, a) = (q_{10}, -, 0)$ would be a transition which the automata would make if it is in state q_5 and the first counter is at -2 and the second counter is at 5 and the input symbol is a. The automata moves to the state q_{10} and subtracts 1 from the first counter, and makes no change to the second counter.

   Note that in a 2-counter automata, the set of states Q is finite. Note also that a 2-counter automata is **deterministic**.

   (a) (**10 pts.**) Show that there exists a 2-counter automata that accepts the language $L_1 = \{0^n 1^n 2^n \mid n \geq 0\}$. Provide a rigorous construction.

   (b) Let $L = \{w\#w^R \mid w \in \{0,1\}^*\}$. This problem shows that L is not accepted by any 2-counter automata.

      i. (**1 pt.**) How many different strings of the form $w\#w^R$ exist whose length is $2n+1$?

      ii. (**9 pts.**) Consider a 2-counter automata with $|Q| = \ell$. Define the *configuration* of a 2-counter automata as the tuple (q, a, b) where $q \in Q$ and $a \in \mathbb{Z}$ is the value of the first counter and $b \in \mathbb{Z}$ is the value of the second counter. How many different configurations could a 2-counter automata be in after consuming n input characters from the alphabet $\{0,1\}$?

      iii. (**15 pts.**) Let M be any 2-counter automata. Using parts (i), (ii) and the pigeon hole principle, show that there must exist an integer $n > 0$, there must exist two *distinct* binary strings $x, y \in \{0,1\}^n$, and there must exist a configuration (q, a, b) such that: $M(x)$ and $M(y)$ both reach the same configuration (q, a, b) after processing their input.

      iv. (**10 pts.**) Use the previous part to conclude that no 2-counter automata accepts L.

Name: ____________________

Student ID: ____________________

2. **Oracle Machines and Diagonalization.** An *oracle* for a language $\mathcal{L}$ is a magical deterministic device which can, in a single computational step, read an input w and return the answer to the question "is $w \in \mathcal{L}$?". An *oracle machine* is a Turing Machine with the ability to query an oracle. (The textbook also discusses oracle machines in section 6.3)

   This magical power is more formally described by stating that an oracle Turing Machine has a special tape called the oracle tape and a special state q_{oracle}. Whenever the machine enters the state q_{oracle}, let w be the string found on the oracle tape. This entire tape is then magically erased and replaced with the string 1 if $w \in \mathcal{L}$ or 0 if $w \notin \mathcal{L}$. A Turing Machine M with access to an oracle to decide a language $\mathcal{L}$ is denoted by $M^{\mathcal{L}}$. Such an oracle-machine is formally described by an 8-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}, q_{oracle})$.

   As an example, consider the following language $\mathcal{L} = HALT_{TM} = \{\langle M, x\rangle \mid M \text{ halts on input } x\}$. Using an oracle for $\mathcal{L}$ one can construct a *oracle Turing Machine* $M^{\mathcal{L}}$ which **decides** A_{TM} as follows:

   > On input w:
   > - Let $w = \langle N, x\rangle$ be the input.
   > - Write $\langle N, x\rangle$ to the oracle tape and enter the state q_{oracle}.
   > - If the oracle tape contains a 1, then we know that N halts on x. Run N on x and accept if N accepts and reject if N rejects.
   > - If the oracle tape contains a 0, then we know that N does not halt on x. Reject the input w.

   (a) (**10 pts.**) Show that every Turing Recognizable language (with respect to *ordinary* Turing Machines) is *decidable* by some $HALT_{TM}$-oracle Turing Machine.

   (b) (**1 pt.**) Let $\mathcal{L}$ be any fixed language. Show that the number of $\mathcal{L}$-oracle Turing Machines is countably infinite.

   (c) (**19 pts.**) Construct a specific language L_u which is not recognizable by any $HALT_{TM}$-oracle Turing Machine.

   (d) (**15 pts.**) Construct a specific language $\mathcal{G}$ such that some $\mathcal{G}$-oracle Turing Machine decides the language L_u that you constructed above.

Name: ____________________

Student ID: ____________________

3. **Turing Cards.** Alice and Bob are playing the card game, *Turing Cards*. Turing Cards is a game played with a given finite deck of cards. Let $\Lambda = \{1, 2, 3, \ldots, \ell\}$ denote the various types of cards in play. The players have an inexhaustible supply of each type of card.

   The game has two stages. First Bob places the card 1 on the table. Then he declares a finite number of replacement rules. A *replacement rule* is a sequence of up to three cards which can be replaced by any other sequence of up to three cards. For example, a rule might be replace the card 1 with the sequence of cards 634 (written $1 \Rightarrow 634$). As an example, Bob might declare the rules:

   $R = \{1 \Rightarrow 634, 634 \Rightarrow 931,\ 31 \Rightarrow 613,\ 1 \Rightarrow 55,\ 965 \Rightarrow 2\}$. The last rule says that one can replace the sequence of cards 965 with the card 2.

   Let R be the set of rules. Formally R is a finite subset of $(\Lambda \cup \Lambda^2 \cup \Lambda^3) \times (\Lambda \cup \Lambda^2 \cup \Lambda^3)$

   Once Bob sets up the rules, it is Alice's turn to play. The second stage of the game consists of Alice repeatedly replacing any valid sequence of cards based on the replacement rules set by Bob. For example, given the rules above, Alice could change the initial card "1" in the following manner.

   $$\mathbf{1} \rightarrow \mathbf{634} \rightarrow 9\mathbf{31} \rightarrow 96\mathbf{1}3 \rightarrow \mathbf{965}53 \rightarrow 253$$

   Above, the boldfaced cards are the ones corresponding to the left-hand-side of the rule that is about to be applied in the next move of Alice.

   Alice wins the game if she makes the card 2 appear on the table. Bob wins the game otherwise. We are interested in studying whether Alice can win the game given a particular set of rules declared by Bob. Let $ALICE$ be the language

   $$ALICE = \{(\Lambda, R) \mid \text{ Alice has a winning strategy with cards } \Lambda \text{ and rules } R\}$$

   - (**40 pts.**) Show that the language $ALICE$ is undecidable.

Name: ______________________________

Student ID: ______________________________

4. **Recursion Theorem.** Recall that $E_{TM} = \{\langle M \rangle \mid L(M) \text{is the empty set}\}$ is not Turing Recognizable. We would like to use the recursion theorem to prove this. Suppose not, i.e say E_{TM} is Turing Recognizable,

   (a) (**25 pts.**) Let R be a purported *recognizer* for E_{TM}. Use the recursion theorem and construct a Turing Machine M on which R behaves incorrectly. More precisely, find a Turing Machine M such that *either*:

   - $L(M)$ is the empty set, but R does not accept $\langle M \rangle$, OR:
   - $L(M)$ is not the empty set, but R accepts $\langle M \rangle$.

   (b) (**20 pts.**) Let $ALL_{TM} = \{\langle M \rangle \mid L(M) = \Sigma^*\}$. Just as in part (a), we will show that ALL_{TM} is not Turing Recognizable using the recursion theorem.

   Suppose not and say ALL_{TM} is Turing Recognizable, then let S be a purported *recognizer* for ALL_{TM}. Use the recursion theorem and construct a Turing Machine N on which S behaves incorrectly. More precisely, find a Turing Machine N such that *either*:

   - $L(N) = \Sigma^*$ but S does not accept $\langle N \rangle$, OR:
   - $L(N) \neq \Sigma^*$ but S accepts $\langle N \rangle$.

Name: ________________________________

Student ID: ________________________________

5. m-**Love (Extra Credit Problem).** Let $\Sigma = \{0, 1\}$, and let m be any positive number. Two strings w, w' are defined to be in m-Love, if one can obtain w' from w by inserting or deleting km-0's and ℓm-1's (at any positions) for any arbitary integers k, ℓ. In other words, given a string w, you can make a string that is in m-love with w by inserting or deleting any-multiple-of-m number of 0's, and any-multiple-of-m number of 1's from w.

   For example the strings 10110, 111 are in 2-Love, since 111 can be formed by removing 2 0's from 10110. As another example, the set of strings $\{101100, 000, 111, 000000, \epsilon, 001100110101\}$ are **all** in 3-Love with each other.

   For any language L, define m-Love(L) as the set of strings which are in m-Love with some string in L. Formally,

$$m\text{-Love}(L) = \{x \in \{0, 1\}^* \mid \exists w \in L : w \text{ and } x \text{ are in } m\text{-Love}\}$$

   - (**40 pts.**) Let L be *any* language (possibly undecidable!). Show that 511-Love(L) is a regular language.

Final

March 16, 2012

Score on this exam: 88/220 (40%)
Rank : 45/87

High: 190 (86%)
Low: 24 (11%)
Median: 89 (40%)
Mean: 100 (45%)

1 2-Counter Automata.

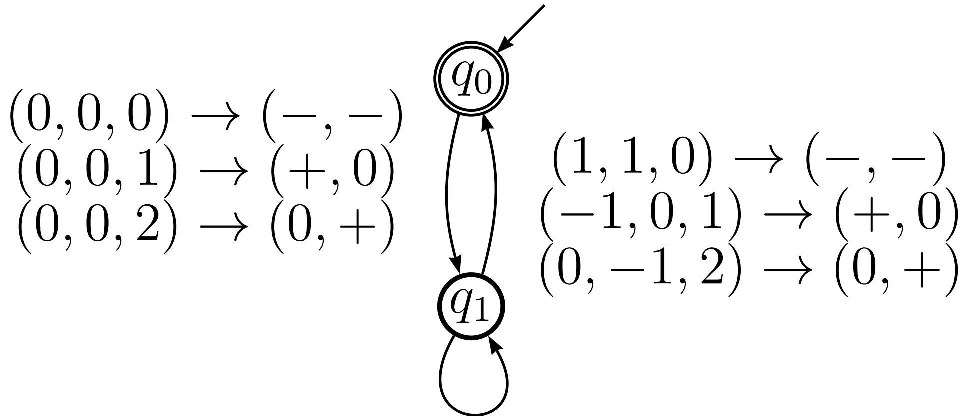
1.1 (a)

We create a 2CA for this language as follows. We recall from class that a PDA can decide the language $L_{eq} = \{0^n 1^n | n \geq 0\}$. The PDA pushed to the stack when reading a 1 and popped from the stack when reading a 0; it had an empty stack when the number of 0's equaled the number of 1's. A PDA is an NFA with a stack, so the idea then is to create a DFA that uses each counter as a stack.

One counter will ensure we have the same amount of 1's and 0's, and the other will ensure we have the same amount of 2's and 0's. Each time we read a 1, we can add to the first counter, and each time we read a 2, we can add to the other counter. But each time we read a 0, we should subtract from both counters. The only way that we will have the same number of 0's, 1's, and 2's is if both counters are empty at the end of the input; this will show that the amount of 0's equals the amount of 1's, and the amount of 0's equals the amount of 2's. Then by the transitive property, the amount of 0's equals the amount of 1's equals the amount of 2's.

We manage our input by having one state q_0 that the machine is in when the counters are zero, and another state q_1 for when the counters aren't zero. We remain in q_1 on any input except when the counters are both almost zero, in which case we move back to q_0 when reading an input that will zero out the counters. In addition we make q_0 (our initial state) an accepting state because the empty string ϵ is part of the language, too (when $n = 0$). The formal construction for the 2CA M_1 is:

$$
\begin{aligned}
M_1 &= (Q, \Sigma, \delta, q_0, F) \\
Q &= \{q_0, q_1\} \\
\Sigma &= \{0, 1\} \\
\delta &= \text{as drawn.} \\
q_0 &= q_0 \\
F &= \{q_0\}
\end{aligned}
$$



1.2 (b)

1.2.1 (i)

If a string $w\#w^R$ has length $2n+1$, then $|w| = \frac{2n+1-1}{2} = n$. Every character in w can be 0 or 1, so there are 2^n ways to create a string w. This means there are 2^n ways to create a string ww^R, since w^R depends on w directly. Adding a # to the middle does not change the amount of ways we can create w, either, since # is not part of w. So, there are 2^n different strings of the form $w\#w^R$ whose length is $2n+1$.

1.2.2 (ii)

The number of configurations is the product of the number of ways each element in the tuple could be assigned. q could have up to l different assignments, since there are l states total. As for a and b, both are bounded by $\pm n$, since after an input is read, we can (at the maximum) increase both counters each time or decrease both counters each time. So, the amount of different values the counters may each take is $2n+1$. This means after n steps, the total amount of configurations (q, a, b) is $l(2n+1)(2n+1) = l(2n+1)^2$.

1.2.3 (iii)

We see that if a machine 2CA M runs for n steps, it has parsed n characters of the input string, and the amount of configurations it could be in is $l(2n+1)^2$, and . However, an input string of length n can have 2^n configurations. Suppose we have 2 distinct strings x and y, both of length greater than or equal to n. By the Pigeonhole Principle, the amount of strings possible to be processed by the machine in n steps is greater than the amount of possible configurations for the machine to be in. So, $M(x)$ and $M(y)$ can reach the same configuration as long as the input string's length is greater than n. In addition, n must be greater than 0; if the strings have length 0, then $2^n \geq l(2n+1)^2$ potentially.

1.2.4 (iv)

The input to L is a string $s = w\#w^R$. If $|w| = n$, then there are 2^n ways to construct such a string. The machine, however, can only be in up to $l(2n+1)^2$ configurations, so by the Pigeonhole Principle, one configuration is repeated at least once. Since some configuration is visited more than once, than we can isolate two distinct characters c_i and c_j $(i \neq j, j > i)$ upon which the machine is in the same configuration. We can then pump all characters in the string $c_i c_{i+1} c_{i+2} \ldots c_{j-1}$ within s to create a new string s', and still produce a string that is accepted by L. This is because repeating those characters will still cause us to reach the same configuration before processing the repeated portion of the string, it will still cause the machine enter the same configuration once c_j is processed each time, and will still cause the machine to proceed as usual from the character c_j and onwards. This property holds for all 2CA's.

However, there is no part of the string $w\#w^R$, where $|w| = n$, that can be repeated to produce a string still in the language. Repeating the entire first portion w or the entire last portion w^R would cause the part before the # to not be equal to the reverse of the part after the #. Repeating any portion containing the # would result in a string with too many #'s. So, there is no way to pump a portion of any string in L. This means no 2CA can accept L.

2 Oracle Machines and Diagonalization.

2.1 (a)

The problem is solved as follows. Given a recognizer, we know that the machine may accept an input w, reject it, or loop indefinitely. A decider, however, will accept some input w, or reject it, but never loop forever. So to turn a recognizer into a decider, we need to reject when the machine loops infinitely on an input. We can use the $HALT_{TM}$-oracle Turing Machine example (denoted $H^{\mathcal{L}}$) to detect when an endless loop occurs, and reject the input that caused it. If the recognizer is rejected, we can reject the input overall; this is because an input that causes a TM to loop forever will never be in the language of the TM. If the recognizer is accepted, the input will never cause an infinite loop, so we can simply run our recognizer on the input and return its result (accept or reject).

So, given a Turing Recognizable language L_{RE} and its recognizer M_L, we can create a corresponding decider $D_L^{\mathcal{L}}$ as follows:

$D_L^{\mathcal{L}} =$ "On input string w:

1. Let $w = \langle M_L, w \rangle$ be the input.
2. Write $\langle M_L, w \rangle$ to the oracle tape and enter the state q_{oracle}.
3. If the oracle tape contains a 0, then we know M does not halt on w. *Reject* the input w.
4. If the oracle tape contains a 1, then we know M halts on w. So, run M_L on w.
5. If M_L accepted w, *accept*.
6. If M_L rejected w, *reject*.

2.2 (b)

We show that the number of $\mathcal{L}$-oracle TM's is countably infinite by creating a bijection from a set we know to be countably infinite to the set of all $\mathcal{L}$-oracle TM's. We can encode each 8-tuple of an $\mathcal{L}$-oracle TM as a string over the alphabet Σ. Then we create a bijection by mapping each element of the ordered set $\Sigma^* = \{\epsilon, 0, 1, 00, 10, \ldots\}$ to each element of the ordered set of encoded $\mathcal{L}$-oracle TM's. Thus, the number of $\mathcal{L}$-oracle TM's is countably infinite.

Σ^*	encoded $\mathcal{L}$-oracle TM's
ϵ	$\langle M_1^{\mathcal{L}} \rangle$
0	$\langle M_2^{\mathcal{L}} \rangle$
1	$\langle M_3^{\mathcal{L}} \rangle$
00	$\langle M_4^{\mathcal{L}} \rangle$
01	$\langle M_5^{\mathcal{L}} \rangle$
$\vdots$	$\vdots$

Since Σ^* is countably infinite, and we have a bijection from Σ^* to the set of all $\mathcal{L}$-oracle TM's, then the set of all $\mathcal{L}$-oracle TM's is countably infinite, too. Another way to think about it is that the set of encoded $\mathcal{L}$-oracle TM's can be no larger than Σ^* since every encoded $\mathcal{L}$-oracle TM is in Σ^*, and not every element of Σ^* can be an encoded into a valid $\mathcal{L}$-oracle TM. In both methods we are mapping (in a one-to-one and onto way) from a subset of Σ^* to all of Σ^*.

2.3 (c)

I adapted the solution to this problem from my solution to Problem 1 from Homework 4. We create a specific language L_u which is not recognizable by a $HALT_{TM}$-oracle TM by Cantor's Diagonalization. Given all the languages of $\mathcal{L}$-oracle TM's, we can create a language not in this set by defining L_u such that for each step in the language, L_u performs the opposite step of a language in our set. We denote the set of all languages of $\mathcal{L}$-oracle TM's as $\mathbb{L}^{\mathcal{L}} = \{L_1^{\mathcal{L}}, L_2^{\mathcal{L}}, L_3^{\mathcal{L}}, \ldots\}$:

$\mathbb{L}^{\mathcal{L}}$ \ Σ^*	ϵ	0	1	00	01	10	11	...
$L_1^{\mathcal{L}}$	0	1	L	1	0	0	L	...
$L_2^{\mathcal{L}}$	1	1	0	L	0	1	0	...
$L_3^{\mathcal{L}}$	L	0	0	0	L	1	1	...
$L_4^{\mathcal{L}}$	0	L	1	L	0	1	0	...
$L_5^{\mathcal{L}}$	0	L	0	1	L	0	L	...
$L_6^{\mathcal{L}}$	L	0	1	L	0	1	1	...
$L_7^{\mathcal{L}}$	1	L	0	1	1	0	L	...
...	...	...	...	...	...	...	...	...

$$\begin{aligned} L_u &= \overline{0}\,\overline{1}\,\overline{0}\,\overline{\text{L}}\,\overline{\text{L}}\,\overline{1}\,\overline{\text{L}}\ldots \\ &= 1\,0\,1\,1\,1\,0\,1\ldots \end{aligned}$$

Now, $L_u \neq L_i^{\mathcal{L}}\ \forall L_i^{\mathcal{L}} \in \mathbb{L}^{\mathcal{L}}$, so $L_u \notin \mathbb{L}^{\mathcal{L}}$. This means L_u is recognized by no $\mathcal{L}$-oracle TM.

2.4 (d)

We let $\mathcal{G} = L_u$. Then a $\mathcal{G}$-oracle TM will trivially decide L_u.

3 Turing Cards.

In retrospect, this solution is almost completely wrong because Turing cards are a context-sensitive grammar.

We can consider the rules that Bob declares as a special CFG $G = (V, \Sigma, R', S)$:

- First we set $\Sigma = \Lambda$ and let $S = S$.
- We note that in Bob's rules, the terminals and nonterminals are the same set. To reconcile this, we first set $V = \{S\} \cup \Lambda'$, where $\Lambda' = \{1', 2', 3', \ldots, \ell'\}$. In other words Λ' is a copy of Λ but with each character $x \in \Lambda$ replaced with a corresponding x'. Now $\Sigma \cap V = \emptyset$.
- Then we change Bob's rules R to create a distinction between terminals and nonterminals. Let R' at first be an empty set of rules. For every rule $r \in R$, we replace all characters $x \in \Lambda$ in the rule with their corresponding counterparts in $x' \in \Lambda'$. This creates a new rule r'. We add this new rule to R'. However, we still do not have a way to terminate. So, we create a set of rules $\{x' \Rightarrow x | x' \in \Lambda', x \in \Lambda\} = \{1' \Rightarrow 1, 2' \Rightarrow 2, \ldots, \ell' \rightarrow \ell\}$. This lets us transition from nonterminals to terminals. We add this set to R', too. Finally we add the rule $S \Rightarrow 1'$ to R'. Now, our set of rules R' lets us emulate Bob's rules in a manner consistent with CFG's; we start with the variable $1'$ (which is equivalent to starting with a 1 in Bob's rules), then transition to various other variables according to Bob's rules where each character x was replaced with x', and can then transition from x' back to x at any time to get non-terminals.

Now $ALICE$ is simply a question of whether or not G produces a string that contains a 2. This is equivalent to seeing if this grammar is decided by a PDA whose language contains a string that contains a 2. So to prove $ALICE$ is undecidable, we perform a reduction from A_{TM} to a machine PDA-2 = $\{\langle P \rangle | P \text{ is a PDA where } L(P) \text{ contains a string with a 2 in it}\}$.

We perform the reduction as follows. Suppose we have a decider M_A (input is $\langle M, x \rangle$) for A_{TM} and a decider M_P for PDA-2. We create a PDA P that that takes as input a modified configuration of a run of M on x. A typical configuration will be $C_1 C_2 C_3 C_4 \ldots C_n$. However P will expect an input of the form $C_1 C_2^R C_3 C_4^R \ldots C_n \# C_1 C_2^R C_3 C_4^R \ldots C_n \#$.

P will verify that the configuration of M on x is indeed a valid configuration. We structure P using the proof from HW 6 #6 as a guideline. First we note that each configuration C_i is written as in the hint for that problem; that is, "write the configuration $(3, q_0, 11100 \sqcup 0)$ as $11q_0100 \sqcup 0$, and the configuration $(4, q_5, \sqcup \sqcup \sqcup\sqcup)$ as $\sqcup \sqcup \sqcup q_5 \sqcup$" for example. We need to check 3 things to ensure a valid configuration: that C_1 is a valid start state, that $C_i \vdash C_{i+1}$, and that C_n is a valid accepting state.

- To ensure that C_i is a valid configuration, we simply check that $C_i = q_0 \sqcup \sqcup \ldots \sqcup$. The PDA can have states that do simple string comparison to verify this.
- To ensure that each $C_i \vdash C_{i+1}$, we reverse every other configuration. This is because a PDA can push the string C_i onto the stack, and check that it is equal to the reverse of C_{i+1} character-by-character as it pops elements off of the stack. Up to 3 symbols next to the head may be different because valid configuration sequences can have a difference of up to 3 symbols from configuration to configuration (depending on the value of i). However, if we only pass the PDA the configuration $C_1 C_2^R C_3 C_4^R \ldots C_n$, then after verifying $C_1 \vdash C_2$, the input C_2 is gone and we can not verify that $C_2 \vdash C_3$. The solution is to pass a second copy of the configuration as input. That way, we can check the $C_i \vdash C_{i+1}$ for even i's on the first instance of the configuration sequence, and that $C_i \vdash C_{i+1}$ for odd i's on the second instance of the configuration sequence.
- To ensure that C_n is a configuration valid for our purposes, we must check that C_n is an accepting configuration. This is done by ensuring that only one state, q_{halt}, appears in C_n.

If the configurations are valid, then this means M accepted x. However our PDA must check that we accepted a string with 2 in it. We can do this check simultaneously with ensuring that C_1 is valid,

$C_i \vdash C_{i+1}$, and that C_n is valid; we simply check to see if at least one C_i $\forall i \in \{2, 3, \ldots, n\}$ has a 2 or $2'$ on the tape. If there is, then Alice has found a 2 with Bob's rules.

Finally, we pass P to M_P. So if M accepted x, then M_P will receive a PDA that has a string with a 2 in its language. Otherwise it will not. In this way we have created the decider M_A for A_{TM}. However, since A_{TM} is undecidable, PDA-2 is undecidable, and since PDA-2 is equivalent to $ALICE$, then $ALICE$ is undecidable, too.

4 Recursion Theorem.

The method behind using the recursion theorem to prove unrecognizability is to first declare a machine M, assume that we have the description of the machine $\langle M \rangle$ within M, and then call the recognizer on $\langle M \rangle$ inside of M. Based on the results of the recognizer, we will then perform an action that is contradictory to what the results of the recognizer imply the machine should do. In this way, we create a machine that contradicts (and invalidates) the recognizer.

4.1 (a)

Let R be the recognizer for E_{TM}. We create a machine M that R will fail on as follows:

M = "On input string w:
1. Let $N = \langle M \rangle$, the description of M.
2. Run R on N.
3. If R accepted, then *accept* w.

If R accepted $\langle M \rangle$, then that means the language of M is the empty set. This means M doesn't accept any strings. But above we constructed M to indeed accept a string w (but only when the recognizer claims M doesn't accept any strings). This is a contradiction. So, R fails to produce the right answer on the machine M, and thus R cannot exist, which means E_{TM} is not recognizable.

4.2 (b)

Let S be the recognizer for ALL_{TM}. We create a machine M that S will fail on as follows:

M = "On input string w:
1. Let $N = \langle M \rangle$, the description of M.
2. Run S on N.
3. If S accepted, then *reject* w.

If S accepted $\langle M \rangle$, then that means the language of $M = \Sigma^*$. This means M accepts all strings. But above we constructed M to indeed reject a string w (but only when the recognizer claims M accepts every string). This is a contradiction. So, S fails to produce the right answer on the machine M, and thus S cannot exist, which means ALL_{TM} is not recognizable.

5 m-Love (Extra Credit Problem).

To prove 511-Love is regular, we can create a regular expression for it. This means that given a language L, we can apply a regular expression to it to isolate the strings that are in 511-Love with each other. If ϵ is in 511-Love of a given language, then the regular expression is probably of the form $(R)^*$ where R is another unknown, regular expression. In this way we can detect ϵ. The regular expression must also detect multiples of 511 1's and 511 0's in arbitrary positions. So it will contain the terms $\left(\bigcup_{511} 0^*10^*\right)^*$ and $\left(\bigcup_{511} 1^*01^*\right)^*$. This coupled with the above assumption is enough to detect 511-Love. The regular expression for 511-Love is $\left(\left(\bigcup_{511} 0^*10^*\right)^* \cup \left(\bigcup_{511} 1^*01^*\right)^*\right)^*$.