Problem 1, CS21 Midterm, Matt Lim

(a) We will use the CFL pumping lemma to show that L_1 is not context free.

Choose $w = a^p b^p c^p d^p$. $w = uvxyz$, with $|vy| > 0$ and $|vxy| \leq p$. Since $|vy| > 0$ and $|vxy| \leq p$, we get the following possibilities for w.

1. vxy could be completely contained in one symbol. Thus, pumping on v and y would add to the number of one symbol, while the number of the other symbols would stay the same. This would produce a string not in the language. For example, say we start with $w = a^p b^p c^p d^p$, where $i = j = p$. If v and y were contained solely in a, pumping up would give us $w = a^k b^p c^p d^p$, where $k \neq p$. But for w to be in the language, there must be the same number of a's and c's. So w is not in the language. Similar arguments can be made for if vxy were contained solely in b, c, or d.
2. v and y could contain two adjacent symbols (never three, because $|vxy| \leq p$). For example, v could contain a's and y could contain b's, or v could contain a's and b's and y could contain just b's. Thus, pumping up would add to the number of two adjacent symbols; that is, the number of a's and b's, b's and c's, or c's or d's could be increased. Then we have that pumping up produces a string not in the language, because the language requires an equal number of a's and c's and an equal number of b's and d's. For example, say vxy is contained in the a's and b's. Then pumping up would result in a string where the number of a's and the number of c's are unequal, a string which is not in the language.

Then we have that any way that u, v, x, y, z are chosen, we can pump on the string $w = a^p b^p c^p d^p$ to get a string not in the language. Thus, by the pumping lemma, $L_1 = \{a^i b^j c^i d^j : i, j \geq 0\}$ is not context free. Note that since regular languages are included in context free languages, L_1 is also not regular.

(b) L_2 is context free but not regular. We will first prove that it is not regular by using the pumping lemma.

Assume L_2 is regular. Then there exists a pumping length p. We will choose the string $w = a^p b^p c^p d^p$. We can write this string as $w = xyz$, with $|y| > 0$ and $|xy| \leq p$. Since $|xy| \leq p$, then y must be contained in the first p a's (i.e y can only be in the a's). So, choose any y in the first p a's. Pumping up on y gives us an uneven number of a's and d's. That is, we get $w = a^k b^p c^p d^p$, where $k \neq p$. But then w is not in L_2 when we pump $i > 1$ times. But the lemma states that for every $i \geq 0$, $xy^i z \in L_2$. Thus we have that L_2 is not regular.

Now we will prove that L_2 is context free using the following statement: L is recognized by a NPDA implies L is described by a CFG. We will construct a NPDA that recognizes L_2, thus proving L_2 is context free.

Look at the NPDA constructed below. Let k be any symbol. As we can see, L_2 is recognized by our NPDA. Thus, L_2 is context free.

Thus, since we earlier proved that L_2 is not regular, we have that L_2 is context free but not regular.

(c) L_3 is a regular language. We will prove this by constructing an NFA that recognizes L_3. This, since we have that NFAs and FAs are equivalent and that the regular languages are those languages recognized by a FA, suffices to prove that L_3 is regular. Given the size of this NFA, a formal definition or drawing would be unruly. So I will simply describe our NFA in detail. The NFA will work as follows:

From the starting state, there will be 1002 paths to take. We will have the start state be an accept state to accept the empty string.

- The first path will go to a reject state if it reads a symbol that is not a.
- The next 1000 paths will be to accept all strings of the form $a^n b^n c^n$, where $1 \leq n < 1001$. So, the first of these paths will accept $a^1 b^1 c^1$, the second will accept $a^2 b^2 c^2$, and so on. It is easy to picture how this is done, but I will shortly explain it. Basically, on each path, we will have exactly n states for each symbol a, b, c, and transition from the starting state to the first a state, from the last a state to the first b state, and from the last b state to the first c state. The last c state will be an accept state. That accept state will then transition to a non-accept state if it reads any more characters, and then that state will loop on itself for the remainder of the string.
- The last path will start with 1001 states to read through 1001 a's. The 1001th state will be the first accept state, called A_1. This state will loop on itself when reading a's. This state will transition to a non-accept state if it reads a symbol that is not a or b. This non-accept state will then loop on itself for the remainder of the string. Now back to A_1. This state will transition to the second accept state, A_2, if it reads a b. A_2 will loop on itself when reading b's. It will transition to a non-accept state if it reads a symbol that is not b or c. This non-accept state will then loop on itself reading the remainder of the string. Now back to A_2. A_2 will transition to the third accept state, A_3, if it reads a c. A_3 will loop on itself when reading c's. It will transition to a non-accept state if it reads a symbol that is not c. This non-accept state will then loop on itself reading the remainder of the string.

We can see that the NFA described above recognizes the language L_3. Thus, by NFA-FA equivalence, and the fact if a language is recognized by a FA then it is regular, we can conclude that L_3 is regular.

Problem 2, CS21 Midterm, Matt Lim

(a) We will show that the language

$$\text{CFL-IN-REG} = \{(G, E) : G \text{ is a CFG, } E \text{ is a regular expression, and } L(G) \subseteq L(E)\}$$

is decidable. Before getting into the meat of the proof, we will first establish some truths. First of all, we know that we can convert our regular expression E into an equivalent NFA E_{NFA} by using the procedure for this conversion given in Theorem 1.54 in the book. Secondly, we know that we can convert our CFG G into an equivalent NPDA G_{NPDA} by using the procedure for this conversion given in Theorem 2.20 in the book. Lastly, both these conversions can be done using a TM. Now, onto the proof.

To prove this, we first construct a new NPDA N_{NPDA} from G_{NPDA} and E_{NFA}, where N_{NPDA} accepts only those strings that are accepted by G_{NPDA} and not accepted by E_{NFA}. Thus, if everything in $L(G)$ is also in $L(E)$, N_{NPDA} will accept nothing. The language of N, where N is the CFG that is equivalent to N_{NPDA}, is

$$L(N) = L(G) \cap \overline{L(E)}$$

We know that $L(N)$ is context free because it is the intersection of a CFL and a regular language (regular languages are closed under complement). The proof for this is in the book, but we don't need to prove it since the problem gives it to us as a fact. Since we know this, we know that it is in fact possible to construct our NPDA N_{NPDA}, given the CFG-NPDA equivalence. We can construct N_{NPDA} from G_{NPDA} and E_{NFA} with the construction for proving the class of CFLs are closed under intersection with regular languages. This construction is an algorithm that can be carried out by Turing machines. So, we can construct a TM F that will decide CFL-IN-REG as follows:

F = "On input $\langle G, E \rangle$, where G is a CFG and E is a regular expression:

1. Construct NPDA N_{NPDA} as described.
2. Run TM R from Theorem 4.8 of the book on input $\langle N \rangle$. This TM takes a description of a context free grammar and determines if its language is the empty set.
3. If R accepts, accept. If R rejects, reject."

(b) We will show that the language

$$\text{REG-IN-CFL} = \{(E, G) : G \text{ is a CFG, } E \text{ is a regular expression, and } L(E) \subseteq L(G)\}$$

is undecidable.

First, we will show that the language

$$\text{CFL-EQ-REG} = \{(G, E) : G \text{ is a CFG, } E \text{ is a regular expression, and } L(E) = L(G)\}$$

is undecidable. We can see this is true by considering the possible ways in which one could decide it. One way would be to use G to go through all derivations of the grammar, and try running each derived string through the equivalent NPDA of E (and vice versa). However, this idea doesn't work, as infinitely many derivations might have to be tried. Another way would be to construct a machine C whose language $L(C)$ is the symmetric difference of $L(G)$ and $L(E)$; that is, $L(C) = (L(G) \cap \overline{L(E)}) \cup (\overline{L(G)} \cap L(E))$. Then, one could check if the language of C is empty, in which case $L(G)$ and $L(E)$ would be the same. However, this machine cannot be constructed, because CFLs are not closed under complementation. These are the only two ways one might attempt to decide CFL-EQ-REG. And, since neither work, we can conclude CFL-EQ-REG is undecidable.

Now, onto the proof. Assume, to the contrary, that REG-IN-CFL is decidable by TM R. Also, let the TM that decides CFL-IN-REG (proved to be decidable in part (a)) be called T. Then we can construct a TM S to decide the language CFL-EQ-REG.

S would operate as follows.

S = "On input (G, E), where G is a CFL and E is a regular expression:

1. Run TM R on input (E, G) and TM T on input (G, E).
2. If R and T accept, accept.
3. If R or T reject, reject."

Clearly, if R decides REG-IN-CFL, then S decides CFL-EQ-REG. But we proved above that CFL-EQ-REG is undecidable. So we have a contradiction, and can conclude that REG-IN-CFL is undecidable.

Problem 3, CS21 Midterm, Matt Lim

To prove that L_1 and L_2 are recursively inseparable, we will first prove that L_1 and L_2 are undecidable. To do this, it suffices to show that the language $HALT = \{\langle M, x\rangle : \text{TM } M \text{ halts on input } x\}$ is undecidable, since both L_1 and L_2 hinge on determining whether M halts. To prove that $HALT$ is undecidable, we will do a proof by contradiction (can be found in the book):

Assume that TM R decides $HALT$. We can construct a TM S to decide A_{TM}, with S working as follows.
$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, reject.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, accept; if M has rejected, reject."

Then we have that if R decides $HALT$, then S decides A_{TM}. This is a contradiction, and because A_{TM} is undecidable, $HALT$ must also be undecidable.

Now we have that since $HALT$ is undecidable, L_1 and L_2 are also undecidable. This is because to decide L_1 and L_2, it is necessary to decide if M halts. Now consider the complement of L_1, denoted by $\overline{L_1}$. We know that decidability is unaffected by taking the complement. To see that this is true for decidable languages, consider the following. Say we take the complement of a decidable language M. Then we have that the strings in $\overline{M}$ that map to accept map to reject in M, and that the strings in $\overline{M}$ that map to reject map to accept in M. So we still have that there are no infinite loops, and that everything maps to either reject or accept. So $\overline{M}$ is decidable. Then we have that $\overline{L_1}$ is undecidable, since if it were decidable, its complement L_1 would also be decidable, which we know not to be true.

Now we will get into the meat of the proof. First consider the language
$L_3 = \{\langle M\rangle : M \text{ doesn't halt on input } \langle M\rangle\}$. Note that L_3 is the complement of $HALT$ (just in a different form), and is thus undecidable. It is clear that $\overline{L_1} = L_2 \cup L_3$. Now assume, to the contrary, that L_1 and L_2 are recursively separable. Then we have that there exists a decidable language D for which $L_1 \cap D = \varnothing$ and $L_2 \subseteq D$. Because of the first requirement, we have that $D \subseteq \overline{L_1}$. This gives us the statement $L_2 \subseteq D \subseteq \overline{L_1}$. Since $\overline{L_1} = L_2 \cup L_3$, we have two basic possibilities for D. One is that $D = L_2$, and the other is that $D = L_2 \cup L_4$, where $L_4 \subseteq L_3$. Notice that L_4 will always be undecidable, since it will always have the requirement that it needs to decide that M doesn't halt on input $\langle M\rangle$, which is undecidable. In the first case, D is clearly undecidable. In the second case, D is also undecidable, since it is the union of two undecidable languages. So in either case, D is undecidable. This is a contradiction. So our initial assumption that L_1 and L_2 are recursively separable must be false, and we conclude that L_1 and L_2 are recursively inseparable.

Problem 4, CS21 Midterm, Matt Lim

(a) To prove that every language generated by a right-linear CFG is regular, we will use induction. More specifically, we will induct on the number of productions.

First we will consider the base case, where there is only one production. Then the starting variable S must yield a string x, where x is any string of terminals. This is because all other steps produce non-terminals. We can let a language $L = x$. Then we have that L is regular, because it is clearly possible to generate a FA that accepts the string x, and thus recognizes L. So our base case is satisfied.

Next comes the inductive assumption. For this, we will use strong induction, and assume that S yields a regular language in n steps, where $1 \leq n \leq k$ (we will ignore the non-terminals when considering languages S generates). Now we must prove that S yields a regular language in $k+1$ steps. To do this, suppose we have generated a language in k steps. Then, by our inductive assumption, that language is regular. Consider the possible next steps. An arbitrary non-terminal (at the end of the generated string/language) that we will call A could become xB or x, where x is again a string of any terminals and B is arbitrary non-terminal. Note that for our purposes, these transitions are the same, because when considering the language generated by S, we only care about terminals (if there are non-terminals in the string, then they can just be reduced to a string of terminals, and the following proof still holds). So we must only consider one of the two transitions. Let us consider WLOG $A \to x$. Then we have that the new language is the concatenation of two languages. We know that our newly generated language x is regular, because of our inductive assumption. We also know that the other language is regular by our inductive assumption, because it was generated in n steps, where $1 \leq n \leq k$. Then we have that our new language is the concatenation of two regular languages. And since regular languages are closed under concatenation, then the language generated by our $k+1$st step is regular as well. Another way to prove this is to look at the derivation $S \to xB \to^* xy$. By induction y is regular and x is regular, so we have that xy, the concatenation of two regular languages, is also regular.

Thus, we can conclude that every language generated by a right-linear CFG is regular.

(b) To prove that every regular language is generated by some right-linear CFG, we will again use induction. This time, we will induct on the number of strings in an arbitrary language L.

First we will consider the base case, generating the first string in our language. In this case, the starting variable S can yield a string x, where x is a string of any terminals. Since x can be a string of any terminals, then we can generate any string (assuming that the alphabet of terminal symbols suffices). So we can generate all strings, and thus we can generate the first string that is in our language. So our base case is satisfied.

Next comes the inductive assumption. Assume we can generate the first k strings of L. Now we must prove that we can generate the $k+1$st string in our language. To show this, first consider how we would generate the first k strings. To generate the first string in the language, we could have $S \to x_{1a}$, where x_{1a} is our first string. To generate the second string in the language, we could have $S \to x_{1b}B \to x_{1b}x_{2a}$. Note that I am assuming that the problem means that the x's in the two forms of productions can be different. With this assumption, we can let $x_{1b} = \epsilon$, which means the overall result of the production becomes $S \to x_{2a}$. We can let this be our second string. To generate the third string in our language, we can do the exact same thing. Instead of going from $x_{1b}B \to x_{1b}x_{2a}$, we could do $x_{1b}B \to x_{1b}x_{2b}C \to x_{1b}x_{2b}x_{3a}$, and let $x_{1b}, x_{2b} = \epsilon$ and let x_{3a} be our third string. Basically, the pattern is that every non-terminal has a production that takes it to xN, where $x = \epsilon$ and N is a non-terminal, and another production that takes it to x, where x is a string in the language. Now back to our inductive assumption. If we are able to generate the first k strings of L, then the last steps of generating the kth string look like the following: $x_{1b} \dots x_{(k-1)b}K \to x_{1b} \dots x_{(k-1)b}x_{ka}$ (let x_{ka} be our kth string). So to generate our $k+1$st string, we need only to make the following modifications to the above steps: $x_{1b} \dots x_{(k-1)b}K \to x_{1b} \dots x_{(k-1)b}x_{kb}K_2 \to x_{1b} \dots x_{kb}x_{(k+1)a}$ (let $x_{(k+1)a}$ be our $k+1$st string). As before, all the x_{nb} strings are epsilon. Thus, we can generate the $k+1$st string of L, which means we can generate all the strings of L.

Then, by induction, we can conclude that we can generate all the strings of an arbitrary regular language with some CFG. Thus, we can conclude that every regular language is generated by some right-linear CFG.

(c) We will define the CFG P as follows.

$$S \to \epsilon \mid 0B \mid 1C \mid 0 \mid 1$$

$$B \to S0$$

$$C \to S1$$

Now we will prove that P generates exactly L, where

$$L = \{w : w \text{ is a palindrome}\}$$

To do this, we must prove two things.

First, we must prove that every string generated by P is a palindrome. To do this we will use induction.

First we will consider the base case, where there is only one production. Then the starting variable S must yield a terminal. Then we have that there are three possible productions: $S \to \epsilon$, $S \to 0$, or $S \to 1$. We can clearly see that all these strings are palindromes. So our base case is satisfied.

Next comes the inductive assumption. Assume P generates a palindrome in k steps, where k is the number of productions done on S. We do not use the number of total productions because that number is not relevant. This is because only S can produce a string of only terminal(s), and also because B and C map back to S. Now we must prove that a palindrome is generated in $k+1$ steps. To do this, consider the derivation $S \to 0B \to 0S0 \to^k 0y0$. Note that this is $k+1$ steps (using our definition of k) since $S \to 0B \to 0S0$ is only one step by our definition. By our inductive assumption y is in L, and thus $0y0$ is in L. The same argument applies to the other case, as follows. Consider the derivation $S \to 1C \to 1S1 \to^k 1y1$. Note that this is $k+1$ steps (using our definition of k) since $S \to 1C \to 1S1$ is only one step by our definition. By our inductive assumption y is in L, and thus $1y1$ is in L. In either case, we get a string that is a palindrome, and thus in L. Thus we have that, by induction, every string generated by P is in L.

Second, we must prove that every string in L is generated by P. To prove this, we will use strong induction on the length of strings.

First we will consider the base cases. By the $S \to \epsilon$ production, S can generate an empty string. By the $S \to 0$ production, S can generate the string 0. And by the $S \to 1$ production, S can generate the string 1. Thus the empty string can be generated (length of zero, the minimum length) and strings of length one (the minimum length greater than zero) in the language can be generated.

Next comes the inductive assumption. Assume that all palindromes of length n, where $0 \leq n \leq k$, can be generated by P. Now we must prove that all palindromes of length $k+1$ can be generated. To do this, we must consider two types of palindromes (both of length $k+1$). The first we will consider are the palindromes that start and end with 0. In other words, we will consider palindromes of the form $0y0$, where y is a palindrome of length $k-1$. By our inductive assumption, y can be generated by our CFG. And by the transitions $S \to 0B \to 0S0$, $0y0$ can be generated (the S in between the two zeroes can generate y). Now we will consider palindromes that start and end with 1. In other words, we will consider palindromes of the form $1y1$, where y is a palindrome of length $k-1$. By our inductive assumption, y can be generated by our CFG. And by the transitions $S \to 1C \to 1S1$, $1y1$ can be generated (the S in between the two ones can generate y). Thus we have that, by induction, our CFG P can generate palindromes of any length, and thus that P can generate every palindrome and every string in L.

Then we have proved that every string in L is generated by our CFG P and that every string generated by our CFG P is in L. Thus, we conclude that our grammar P generates exactly L, the language of palindromes over the alphabet $\Sigma = \{0, 1\}$.

Problem 5, CS21 Midterm, Matt Lim

Suppose we have a language expressible as:

$$L = A/B = \{x : \text{there exists some y} \in B \text{ for which } xy \in A\}$$

where A and B are both RE. Then L is RE, because given an input x, we can simply enumerate all y's in B in lexicographic order, and for each one decide whether $xy \in A$. If the answer is ever "yes" then we accept x; otherwise, we continue on until all the y's are accounted for, and then reject. Clearly the language accepted is exactly L. In other words, we can construct a TM that, given an input x, enumerates all y's in B in lexicographic order and checks if $xy \in A$ for each y. If the answer is "yes", our TM would accept, else it would continue on until all the y's were accounted for, and then reject. Clearly this TM recognizes L, and since a language is RE if recognized by some TM, then L is RE.