ECE 468
Problem Set 1: Regular expressions and finite automata (**Solutions**)

1. For strings containing the letters 'a', 'b', 'c', and 'd' give a regular expression that captures all strings that use their letters in *reverse alphabetical order*, but use at most three of the four possible letters (note that the strings themselves can be longer than 3 letters long, since letters can repeat).

   Solution The regular expression that captures the above language is:

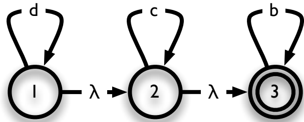
   $$(d^*c^*b^*)|(d^*b^*a^*)|(d^*c^*a^*)|(c^*b^*a^*)$$

   Deconstructing this a bit, note that each of the four "clauses" only uses three of the four letters. Each letter can be repeated zero or more times (repeating zero times is how we can get strings that use fewer than three different letters), and the ordering ensures that the letters appear in reverse alphabetical order.

2. Give a *non-deterministic* finite automaton that captures the regular expression from above. Show the automaton in graphical form.

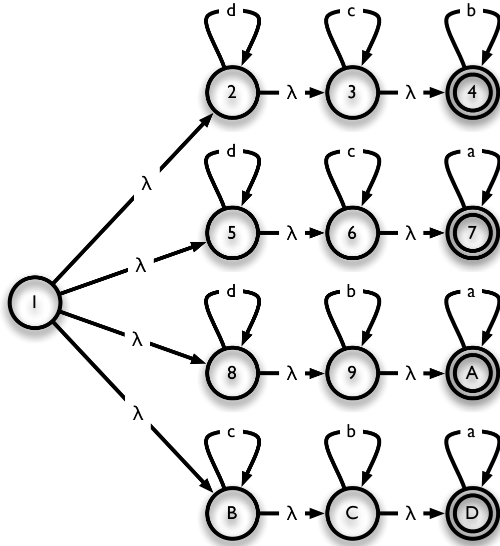
   Solution We will build this NFA up piece by piece. First, note that the NFA for a^* is:

   So the NFA for $d^*c^*b^*$ is:

   Note that we are being extra careful here. When hooking two NFAs up back to back, the final states of the first NFA are hooked up to the start state of the second NFA, and the first NFA's final states are made non-final. In this case, it is actually safe to keep all of the states final, but it is good practice to not do that, to avoid messing something up.

   Now we can put everything together to build the overall NFA:



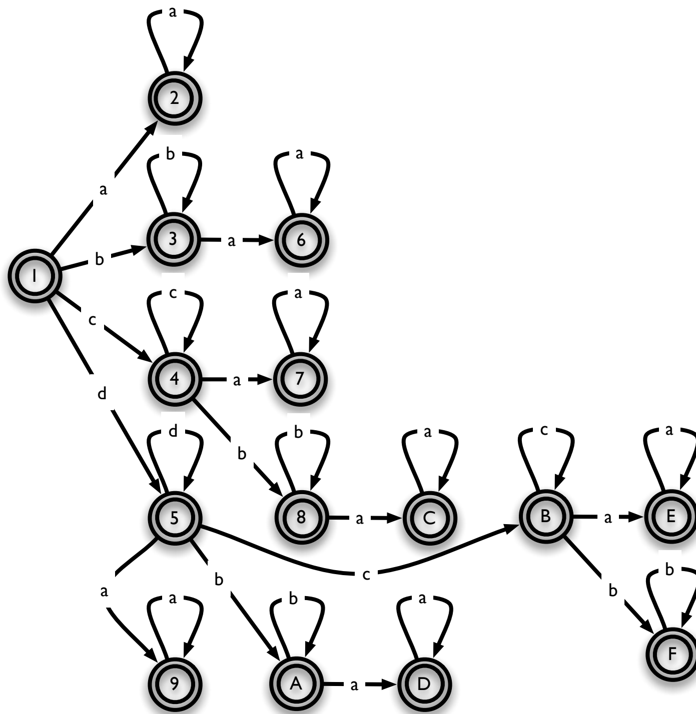
Note that we're using hex numbers to label the states, just so everything has one "digit."

3. Using the construction described in class, give a *deterministic* version of the automaton. You only need to show the transition table.

   Solution Remember that the way to perform DFA construction is to simulate every possible "configuration" of states that can occur after seeing a particular letter (and taking all possible λ transitions). So when we start out, our "pointers" immediately take all possible lambda transitions, giving us an initial state of {1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D}. (That's really long to write out, so we'll just call that state "all.") We can now build the table from here:

State	a	b	c	d	final?	New state
all	7AD	49ACD	3467BCD	23456789A	Yes	1
7AD	7AD	err	err	err	Yes	2
49ACD	AD	49ACD	err	err	Yes	3
3467BCD	7D	4CD	3467BCD	err	Yes	4
23456789A	7A	49A	3467	23456789A	Yes	5
AD	AD	err	err	err	Yes	6
7D	7D	err	err	err	Yes	7
4CD	D	4CD	err	err	Yes	8
7A	7A	err	err	err	Yes	9
49A	A	49A	err	err	Yes	A
3467	7	4	3467	err	Yes	B
D	D	err	err	err	Yes	C
A	A	err	err	err	Yes	D
7	7	err	err	err	Yes	E
4	err	4	err	err	Yes	F

Note that we added a column that shows the relabeled name for each state, just to make later parts of the process easier. Though you did not need to show this, here is the graphical representation of the new machine:



4. Give a *reduced* version of the finite automaton, using the algorithm we used in class. You only need to show the state transition diagram.

   Solution: This machine can be made considerably simpler. We start by hypothesizing that all final states are the same, and that all non-final states are the same. In this case, we only have final states, so our hypothesized reduced machine has just one state: 123456789ABCDEF. We now try to prove that the states behave differently. One set of derivations proceeds as follows. Note that if you consider states and letters in different orders, your intermediate hypotheses may look different. But you will wind up in the same place:

   (a) On a 'd', only states 1 and 5 do not go to error, so 1 and 5 must be different from the rest: 15, 2346789ABCDEF. States 1 and 5 *look* the same here: on a 'd', they both stay in 15, on anything else they go to the merged state 2346789ABCDEF. We'll come back to this.

   (b) Let us try to split 2346789ABCDEF. On a 'c', states 4 and B are the only states that do not go to error. Our new machine has states: 15, 4B, 236789ACDEF.

   (c) Let us try to split 236789ACDEF. On a 'b', states 2, 6, 7, 9, C, E and D go to error, but the rest do not. Our new machine has states: 15, 4B, 38AF, 2679CED.

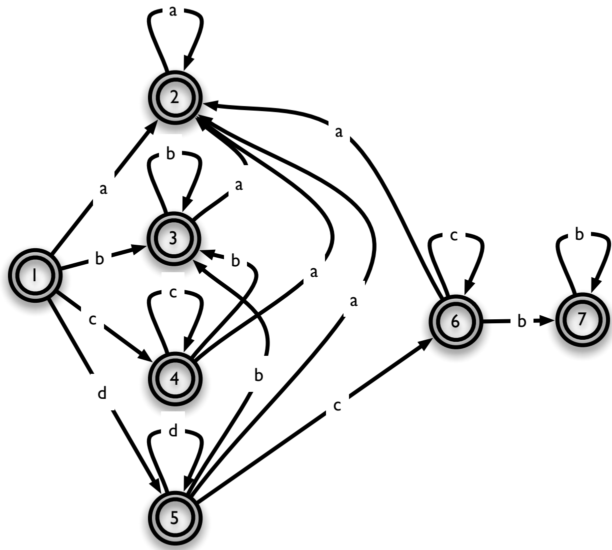
   (d) Let us try to split 38AF. On an 'a', F goes to error, the others do not. Our new machine: 15, 4B, 38A, F, 2679CED.

   (e) Now we can split apart 4 and B: on a 'b', 4 goes to merged state 38A, while B goes to F. Our new machines: 15, 4, B, 38A, F, 2679CED.

   (f) Now that 4 and B have been split apart, we see that on a 'c', state 1 goes to 4, and state 5 goes to B, so they actually behave differently. Our new machine: 1, 5, 4, B, 38A, F, 2679CED.

   (g) We cannot make any more splits, so we are done.

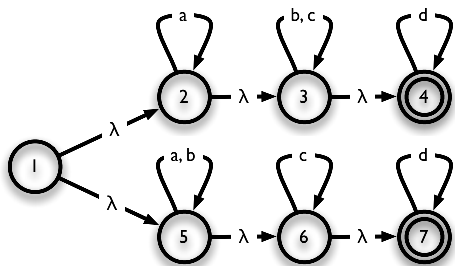
   Our final machine looks like:



5. Build a *reduced, deterministic* automaton for the following regular expression:

$$(a^*(b|c)^*d^*)|((a|b)^*c^*d^*)$$

Solution The NFA for this machine follows a similar pattern to the previous one:



If we construct the DFA transition table for this machine, we get:

State	a	b	c	d	final?	New state
1234567	234567	34567	3467	47	Yes	1
234567	234567	34567	3467	47	Yes	2
34567	567	34567	3467	47	Yes	3
3467	err	34	3467	47	Yes	4
47	err	err	err	47	Yes	5
567	567	567	67	7	Yes	6
34	err	34	34	4	Yes	7
67	err	err	67	7	Yes	8
7	err	err	err	7	Yes	9
4	err	err	err	4	Yes	A

And when we reduce the machine, we get:

State	a	b	c	d	final?
2	2	3	4	5	Yes
3	6	3	4	5	Yes
4	err	4	4	5	Yes
5	err	err	err	5	Yes
6	6	6	8	5	Yes
8	err	err	8	5	Yes

Which looks like: