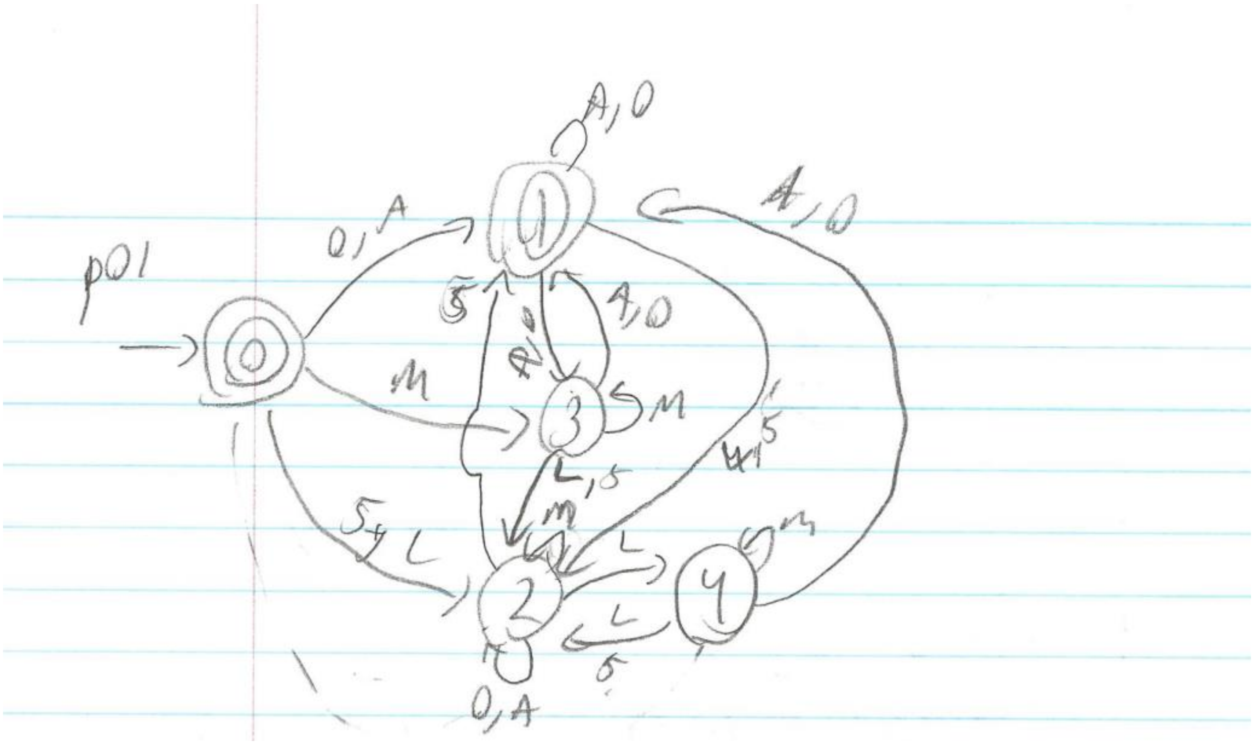
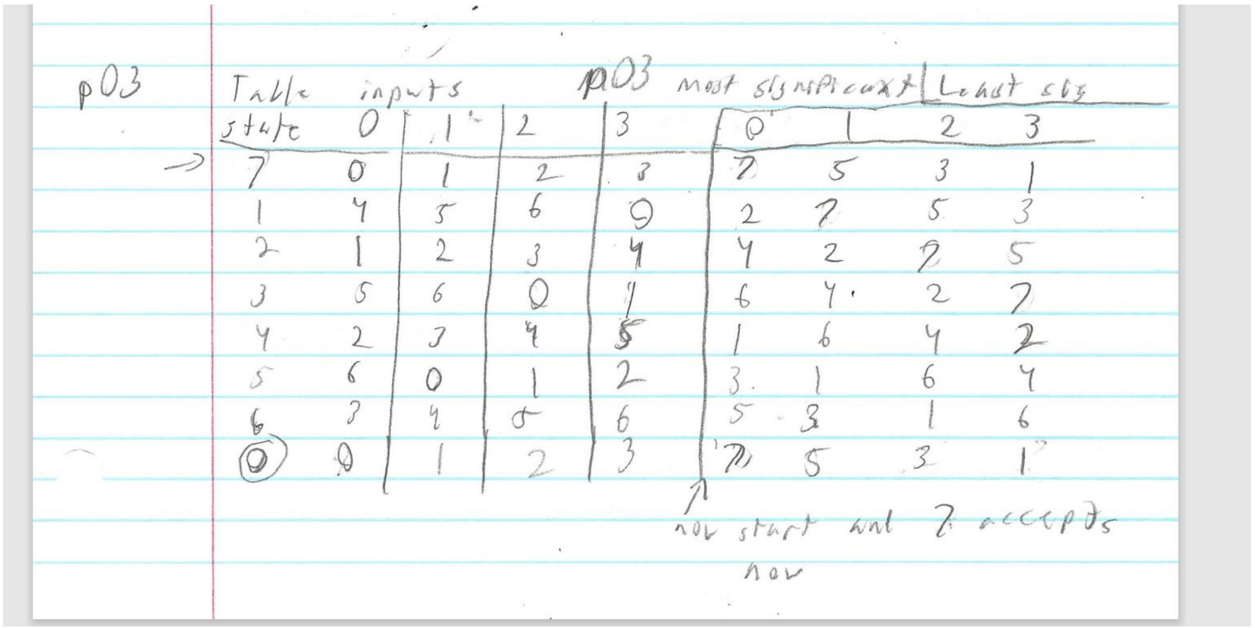
P01:



This is a DFA for p01 where L = 1,3,7,9,B,D and M = 2,4,6,8,C,E all these letters behave the same and it takes up a bunch more pace to write each one out. After making this I went and copied the transitions into a machine file as they appear here.

P02: This machine file is the exact same as the machine above except state 0 is not an accepting state so I copied the p01 file and removed 0 from the accept states.

P03:



p03 Table inputs | p03 most significant | Least sig

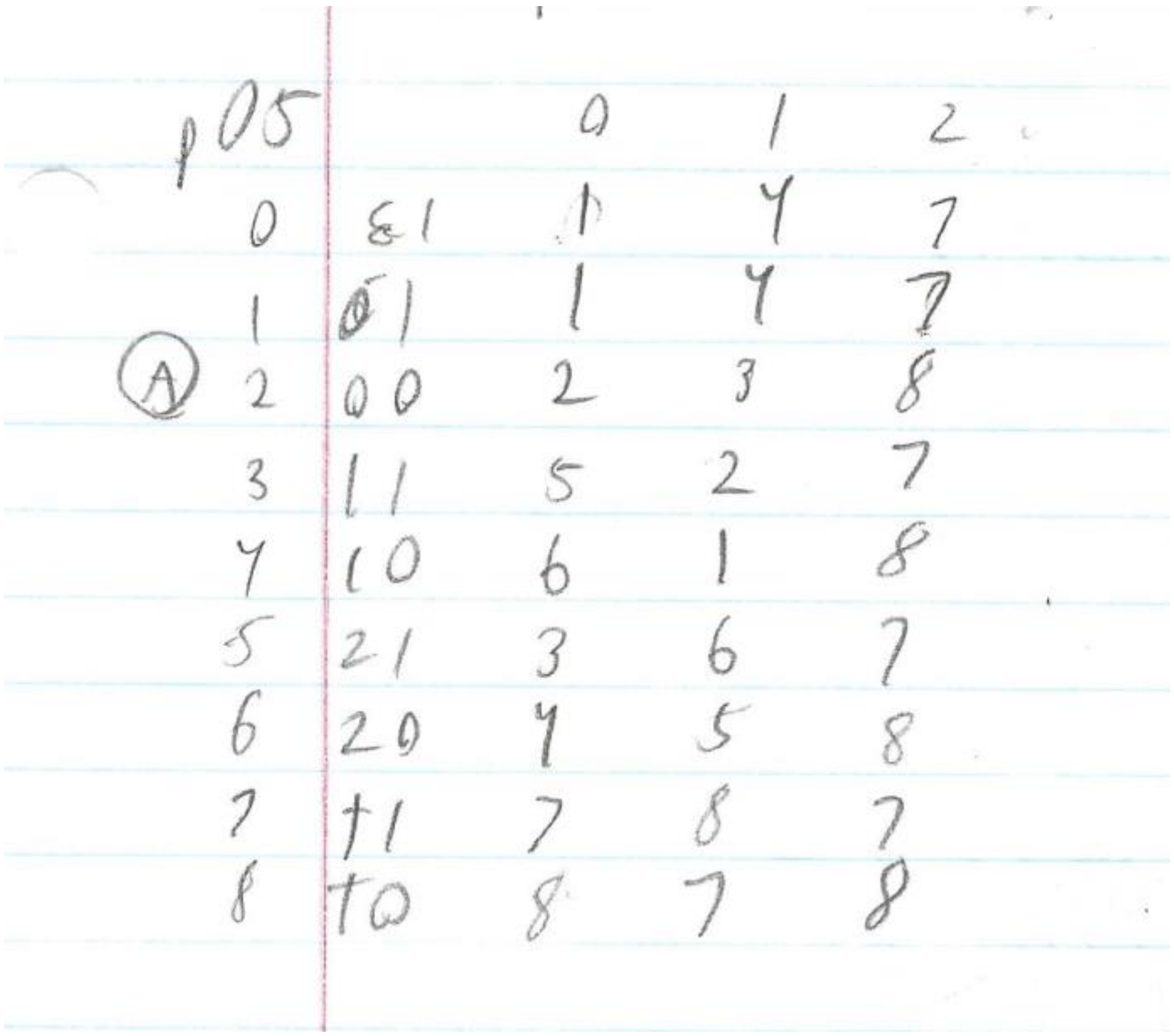
state	0	1	2	3	0	1	2	3
→ 7	0	1	2	3	7	5	3	1
1	4	5	6	0	2	7	5	3
2	1	2	3	4	4	2	7	5
3	5	6	0	1	6	4	2	7
4	2	3	4	5	1	6	4	2
5	6	0	1	2	3	1	6	4
6	3	4	5	6	5	3	1	6
$\boxed{0}$	0	1	2	3	7	5	3	1

now start and 7 accepts now

I made a transition table in order change the machine from reading most significant to least significant and I believe all the logic is correct, but as we only ever changed the way inputs are read on like 2 machines this one gave me some trouble.

P04: I did not have enough time to finish machine p04 so as such I have provided a junk file that doesn't work. If given the time I could make this machine the same as I did for the 3 above, but other classes, other homework and other problems in this project mean that is unrealistic.

P05:

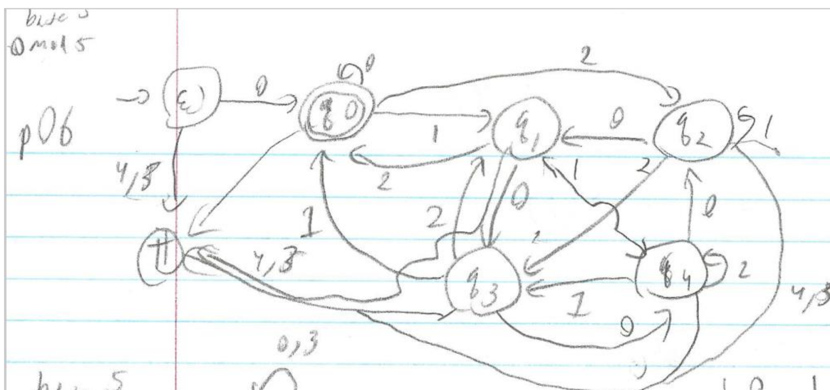


	p05		0	1	2
	0	ε1	1	4	7
	1	01	1	4	7
Ⓐ	2	00	2	3	8
	3	11	5	2	7
	4	10	6	1	8
	5	21	3	6	7
	6	20	4	5	8
	7	†1	7	8	7
	8	†0	8	7	8

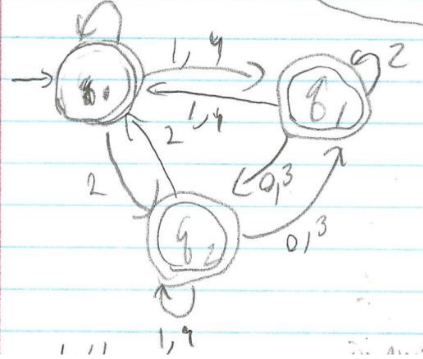
As we did a very similar problem on hw 2 I went and took my answer for that problem and simply combined the two machines but this time I was looking for one that was accepted by only machine 1 (aka binary divisible by 3) and not by machine 2.

base 3
0 mod 5

p06



base 5
not 0 mod 3



	0	1	2	3	4	0	1	2	3	4
						R				
0	0	1	2	0	1	0	1	2	0	1
1	2	0	1	2	0	2	0	1	2	0
2	1	2	0	1	2	1	2	0	1	2

R = forward ✓

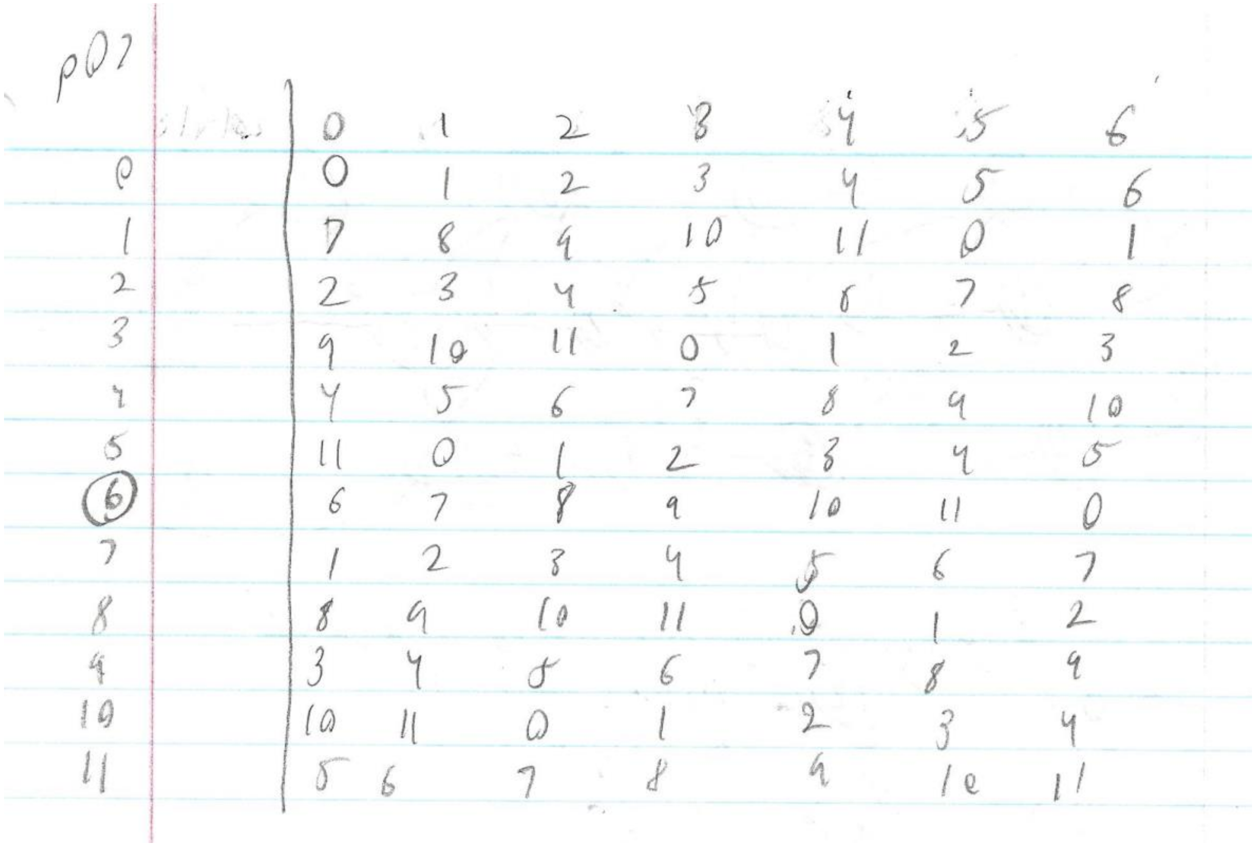
P06:

table 1,4

state	states b3 , b5	inputs 0	1	2	3	4
0	ε 0	00	11	22 ✓	T	T
1	00	00	11	22 ✓	T	T
2	11	32	40	01 ✓	T	T
3	22	12	20	31 ✓	T	T
255	T	T	T	T ✓	T	T
4	32	41	02	10 ✓	T	T
5	01	02	10	21 ✓		
6	40	20	31	42 ✓		
7	12	31	42	00 ✓		
8	31	42	00	11 ✓		
9	10	30	41	02 ✓		
10	41	22	30	41 ✓		
11	02	01	12	20 ✓		
12	21	12	20	31 ✓		
13	20	10	21	32		
14	42	21	32	40		
15	30	40	01	12		

For p06 I made both of the DFA's and combined them in a transition table where I had two total accept states. I then renamed the states and made the machine file to accept it.

P07:

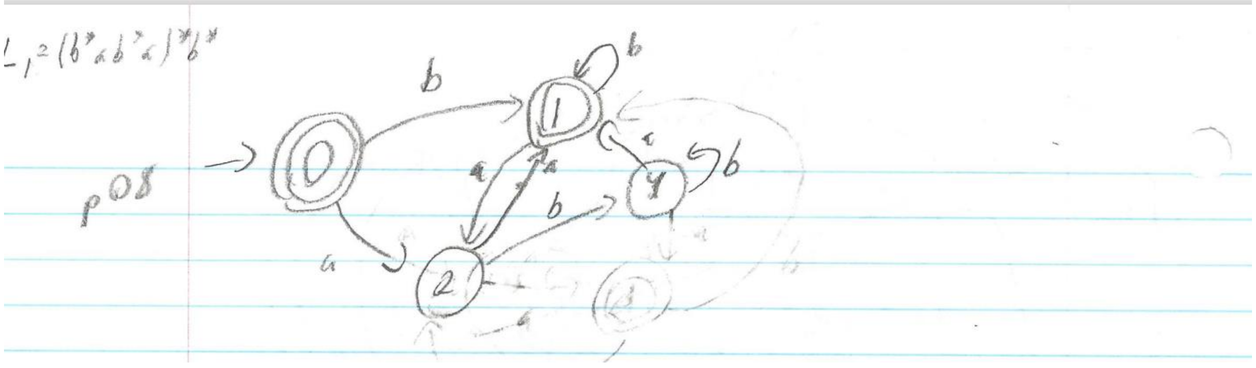


p07

states	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	0	1
2	2	3	4	5	6	7	8
3	9	10	11	0	1	2	3
4	4	5	6	7	8	9	10
5	11	0	1	2	3	4	5
6	6	7	8	9	10	11	0
7	1	2	3	4	5	6	7
8	8	9	10	11	0	1	2
9	3	4	5	6	7	8	9
10	10	11	0	1	2	3	4
11	5	6	7	8	9	10	11

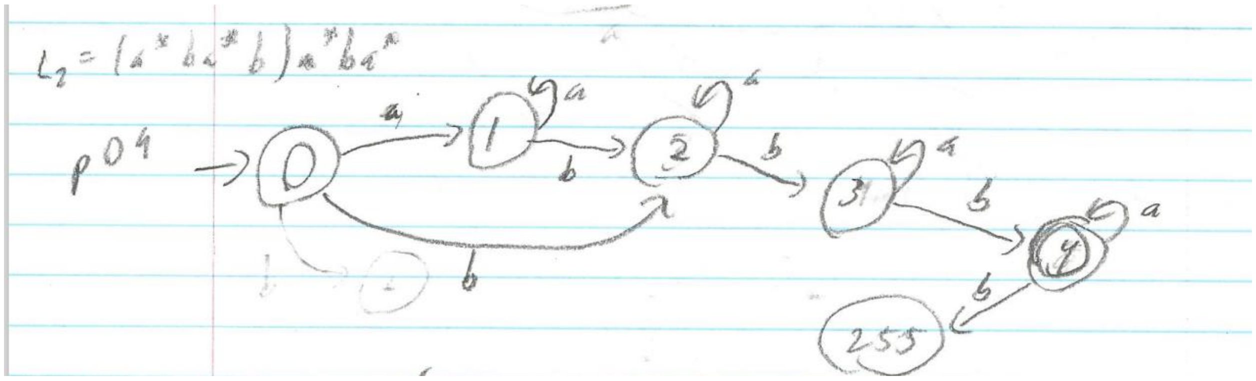
The strategy for p07 was to simply see that if it is divisible by 3 and 2 but not 12 then make it so I can read base 7 mod 12 and then since 2 and 3 can both divide 6 but 12 can't that means that anything 6 mod 12 would be an acceptable string. So, I accept any input that ends in state 6.

P08:



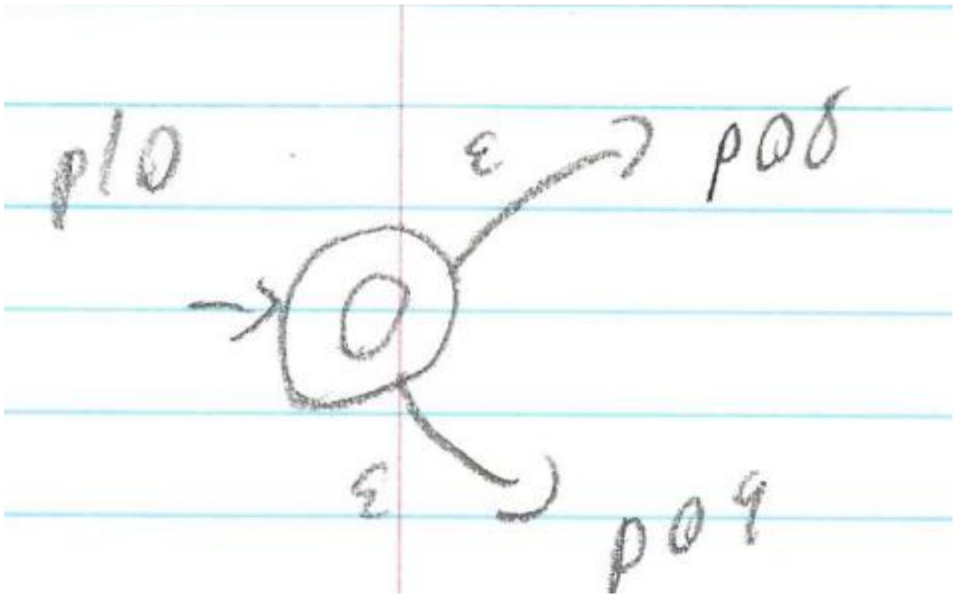
A simple DFA to accept L1. The first thing I did was see what the minimal string it could accept was and that was empty or a single b. Then I noticed that if we have an a we must always have a second one and there can be an infinite amount of b's before and after each a, but there can also be none hence why aa is accepted.

P09:



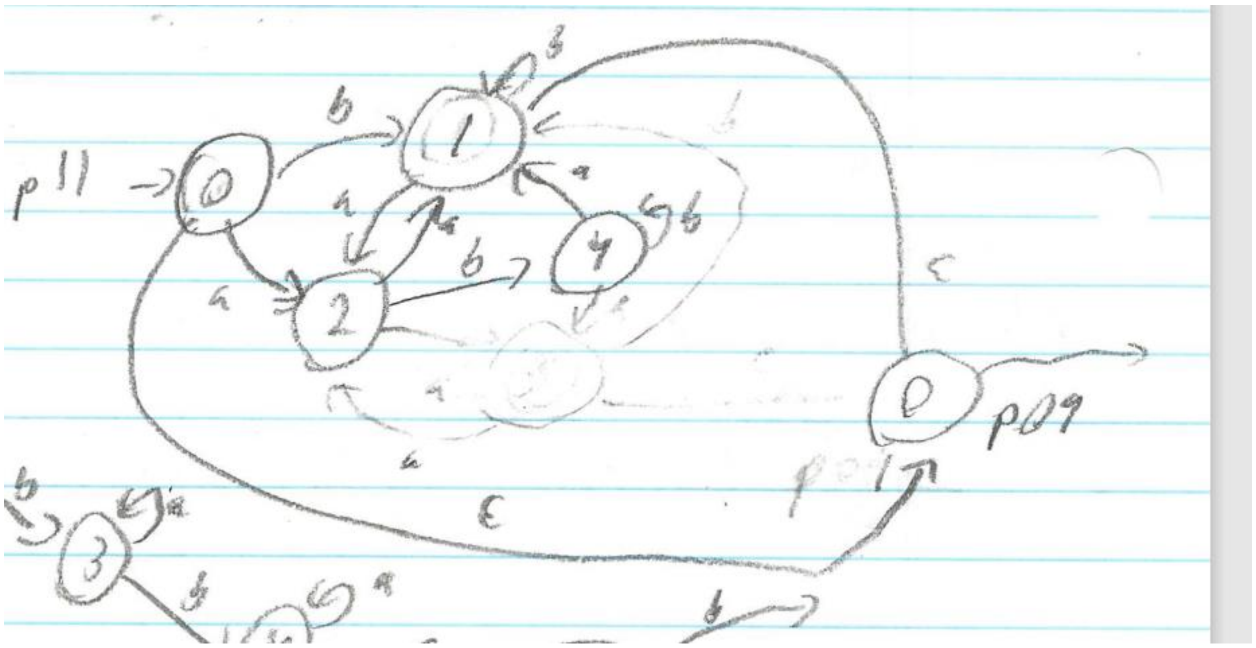
Another simple DFA for L2 the goal was to make sure that I was catching all the a’s correctly and the minimal string was 3 b’s so it a 4th be appears I know to move the string to non-accepting trap state 255.

P10: L1 union L2



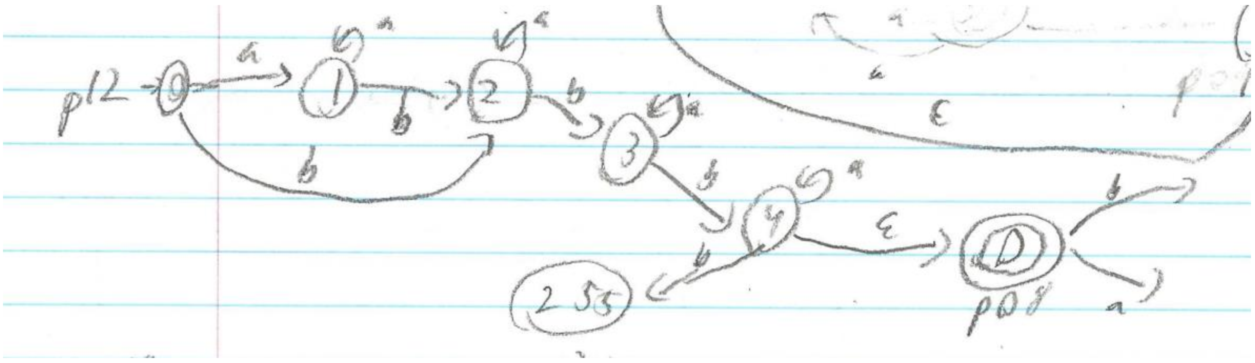
It can be either in L1 or L2 so I simply added a new start state to the machine with an epsilon transition to the start states for L1 and L2 and just copied the machine from p08 and p09.

P11: L1 concat L2



A rough sketch but it shows that in order to make L1 concat with L2 I want to make a machine that whenever L1 can be accepted I make an epsilon transition to the start of L2 (p09) and then I can see if the rest of the string is accepted by p09, if not then I keep trying until either the string is done or it is accepted.

P12: L2 concat L1

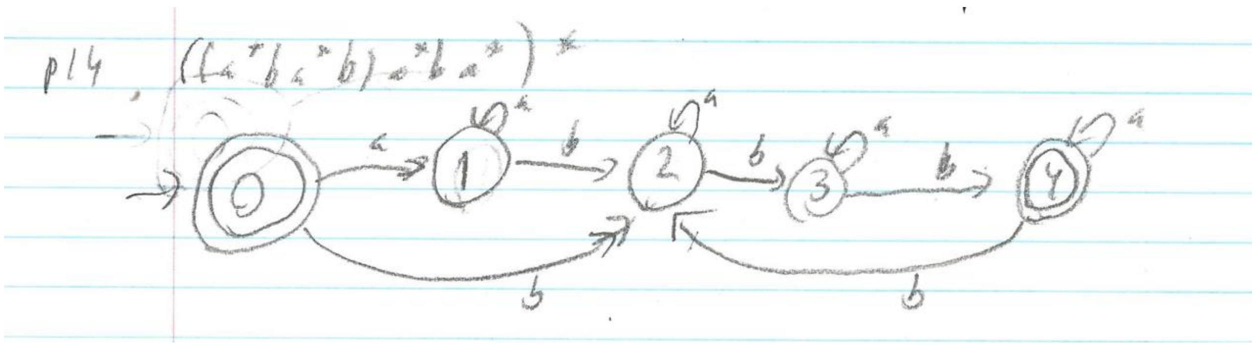


Similar to how I approached p11, here I am instead starting in L2 and whenever I reach the accept state for L2 I use an epsilon transition to try the remainder of the string in L1 and accept the string if it ends in either of the L1 accept states.

P13: L1*

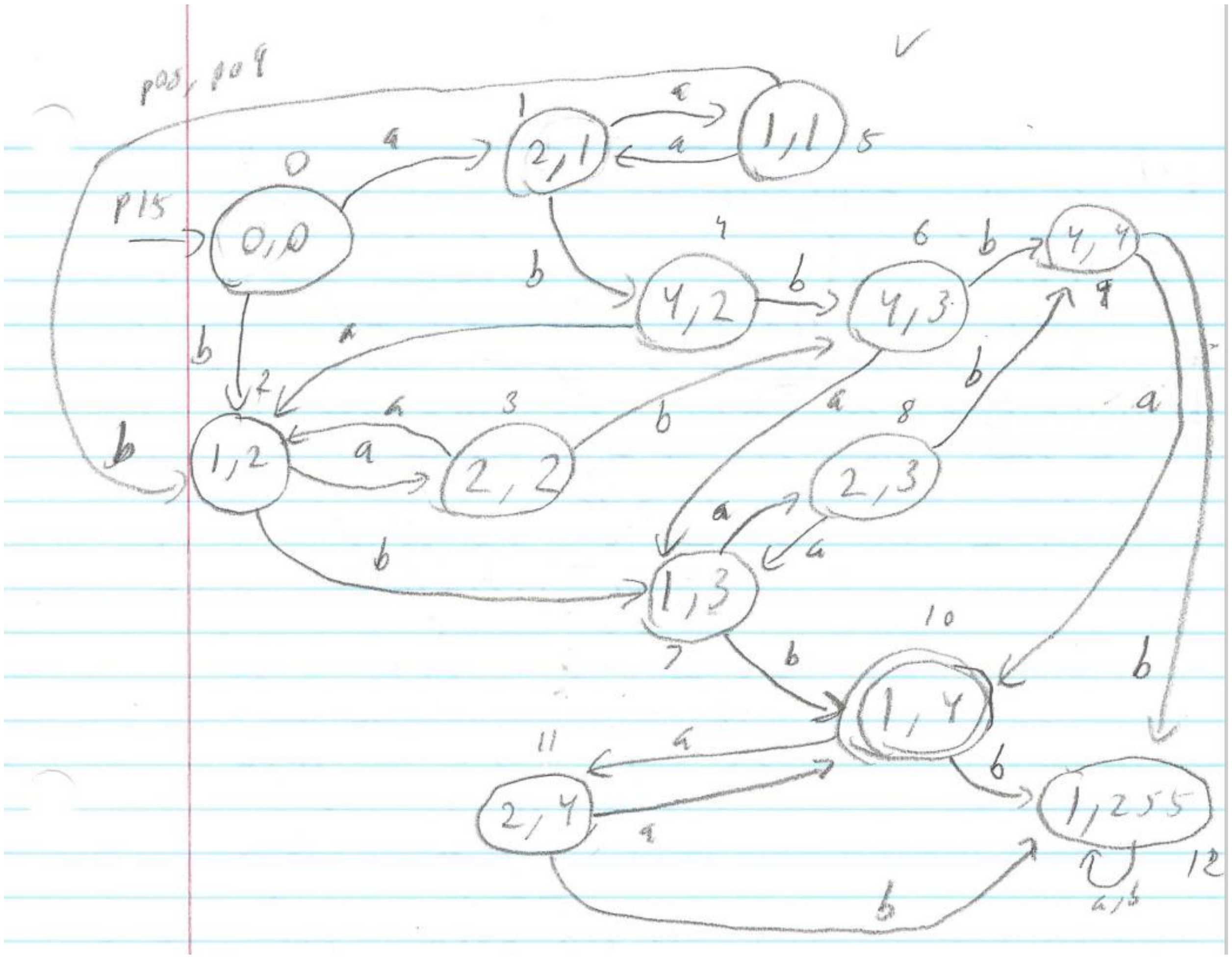
Since L1 can already have it all repeat itself and accept epsilon this machine is the exact same as L1 so I simply reused the machine from p08. I did attempt to confirm this by making a machine that was for L1*, but it turned out identical, so I simply copied the machine.

P14: L2*



Since before if we had more than 3 b's in L2 it was non-accepting and it could not accept the empty string some changes had to be made, with the main ones being a loop to preform the strings over and over again, and needing to make 0 an accept state. This machine doesn't recognize when a new string starts but if it sees a 4th b it knows that either a new string has started or that it won't accept it.

P15: L1 and L2



A DFA for p15 is show here. I combined the states in the order L1,L2 and then made this to cover all the possible transitions up until L2 entered 255, because in 255 it can't be in both L1 and L2 so there was no more point in tracking the states. The only accepting state was 10 or 1,4 as if we have any input, I can no longer go back to state 0 in L1 which is the only other accept state.

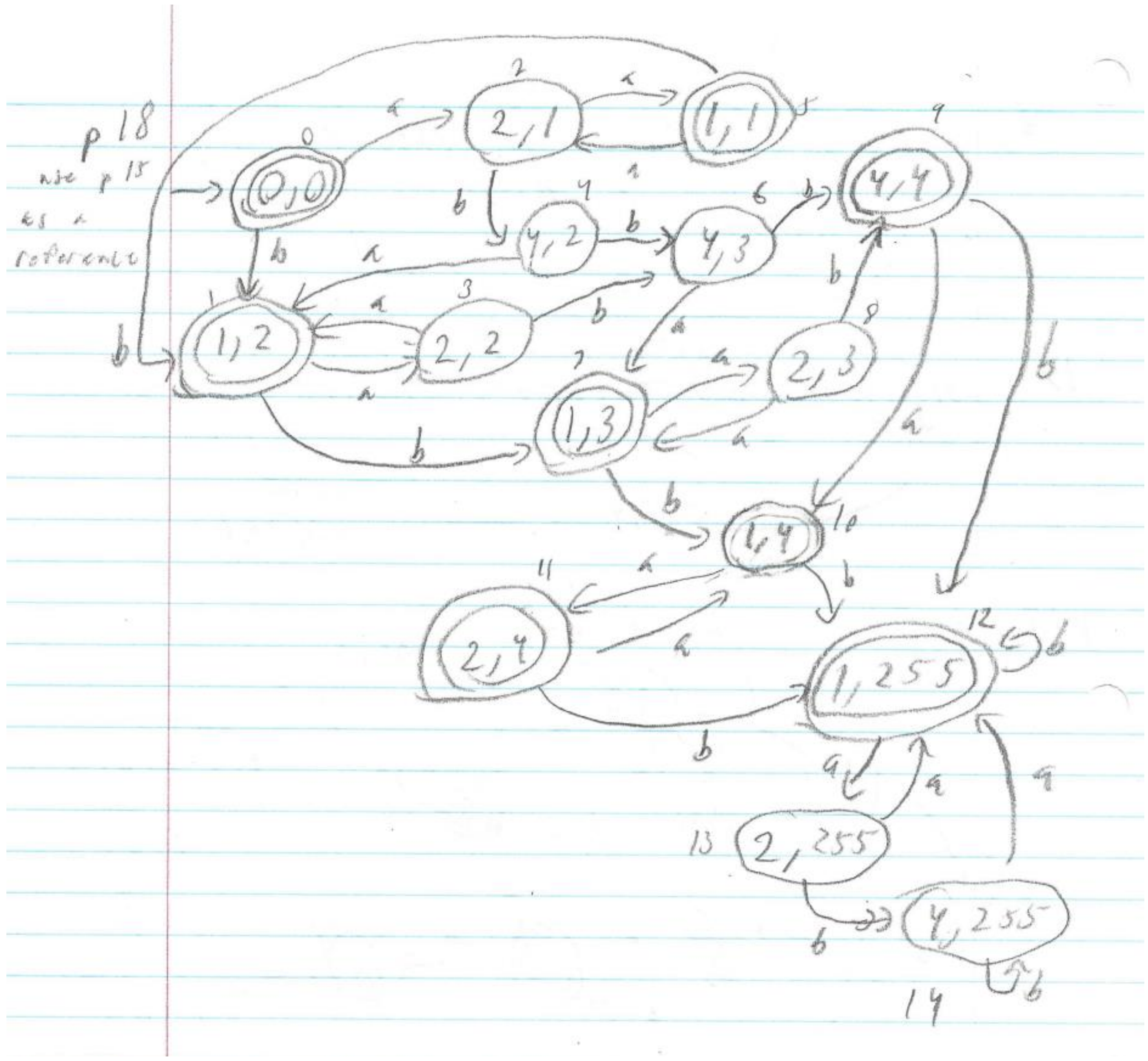
P16: Not L1

For p16 I simply kept the same file I used in p08 for finding what was in L1 and simply changed all the non-accept states to accept states and changed the accept states to non-accept states, while also making it so that 255 was accepted so that the alphabet could be accepted.

P17: Not L2

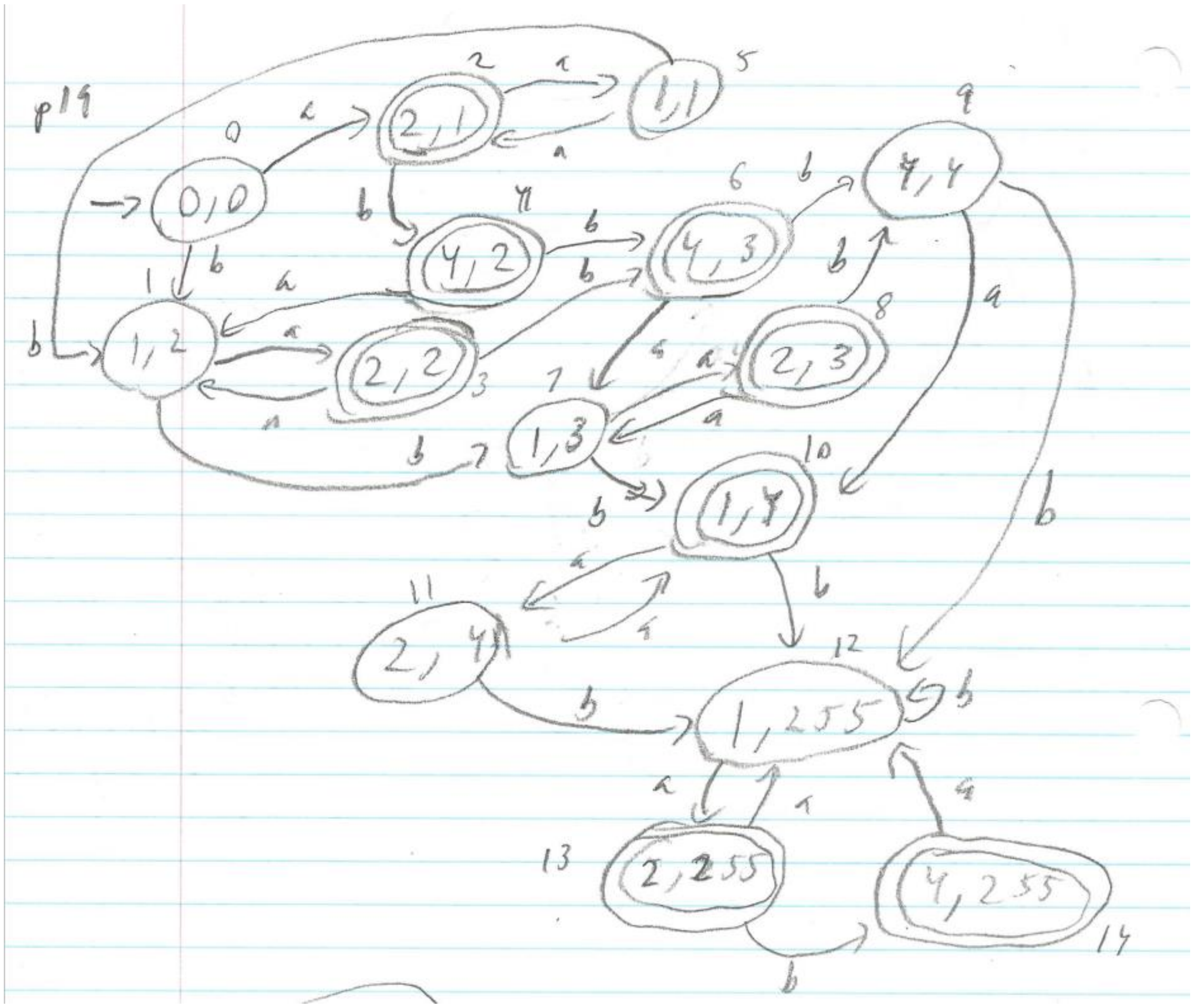
Again a similar approach as the p17 above where I swapped the accepting status of every state and made 255 an accept state in order to catch anything that was outside of the alphabet as this is what we agreed upon in class. Making a new state to accept everything but not be 255 seemed like a large amount of effort when we agreed that 255 was to be a trap state meant to be used for the purpose of catching strings that don't belong.

P18: L1 XOR L2



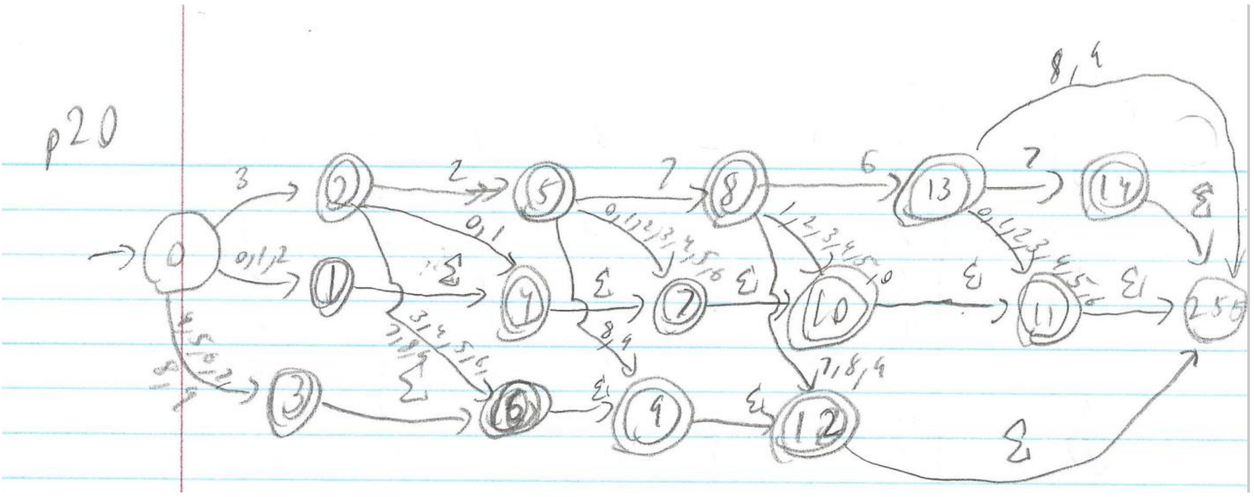
XOR means we want to exclude what we found in the intersection of both L1 and L2 so I used p15 as a reference and reselected the accept states, and I had to expand out 1,255 because we could end in a spot where it is neither in L1 or in L2 in that section, but besides that the DFA is the same for the most part and I simply changed the states around in order to make sure that

P19: L1 XNOR L2



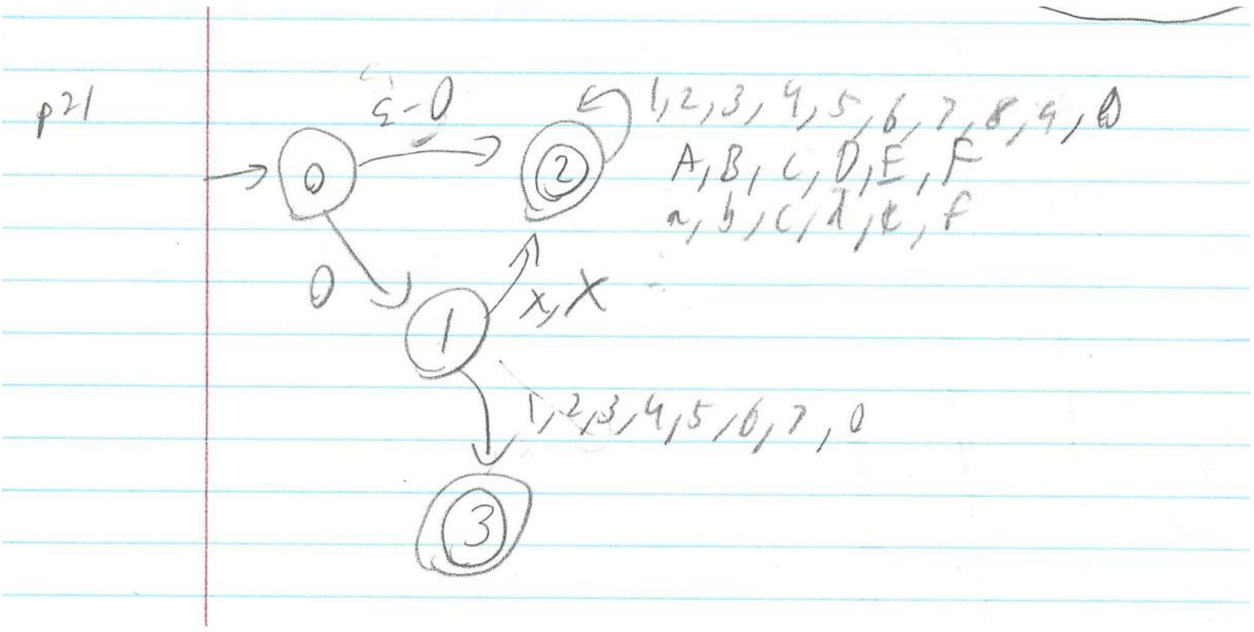
From my understanding of XNOR is the opposite of XOR so I simply swapped the states that p18 was accepting and made them the non-accepting states of p19 and vice versa so we again have a lot of accept states, but it is to make sure that we can accept all. The only thing I am unsure about with XNOR is if it accepts 1,4 because that is the result of AND but through all my understanding it would.

P20:



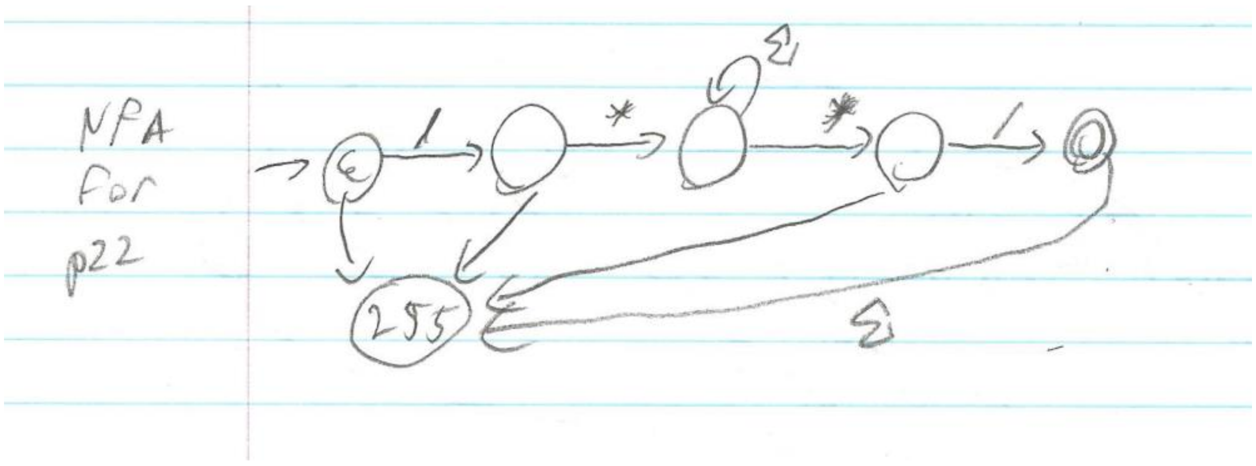
Here is the diagram I made for p20; the objective was to have it accept the max integer so the top path was all the ints in order that could be accepted. If it ever had a number lower than that I moved it to the middle track to keep eyes on numbers that are smaller than the max, but I made an error in this graph and fixed it on my file. State 1 should loop on itself with 0's because 00000001 = 1 but while 1 would be accepted 00000001 wouldn't be by this machine as displayed above, but besides that the bottom path was to track anything larger than 32767, so it would accept numbers like 333, but it wouldn't accept 32777, which is only a little larger than the max.

P21:



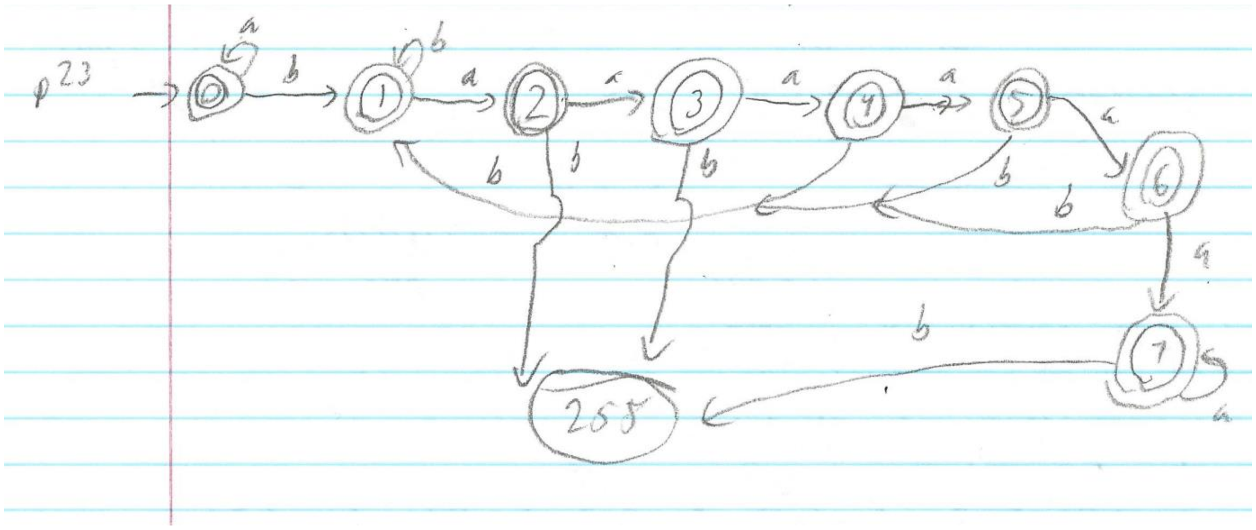
So this diagram has some errors in it but the final file has those errors fixed. So if we start with a 0 I have to check for an x or X to see if the number is in hex or not, if it is in hex I then move it to an accepting state like 2 shown here, where it loops on itself for any hex digit. Where if it is followed by an octal digit I move to state 3 which loops on all the octal digits. If it starts with 1-9 however than I assume we are bealing with a deciam digit and move to a new state 4 to accept anything that stays between 0-9. So there is no direct transition to state 2 from 0 as state 2 need 0x. This shows the basic idea of checking for if it is ocatal or hex before continuing.

P22:



For p22 we need to find all the comments so the idea is pretty simple you check to see if it starts with a / followed by a * and then burn whatever printable characters are inside the string, until you see * then look for the last /, and if anything else appears after that the string is no good. So the basic idea is put most of the printing characters in the alphabet for the middle state and just burn until a *

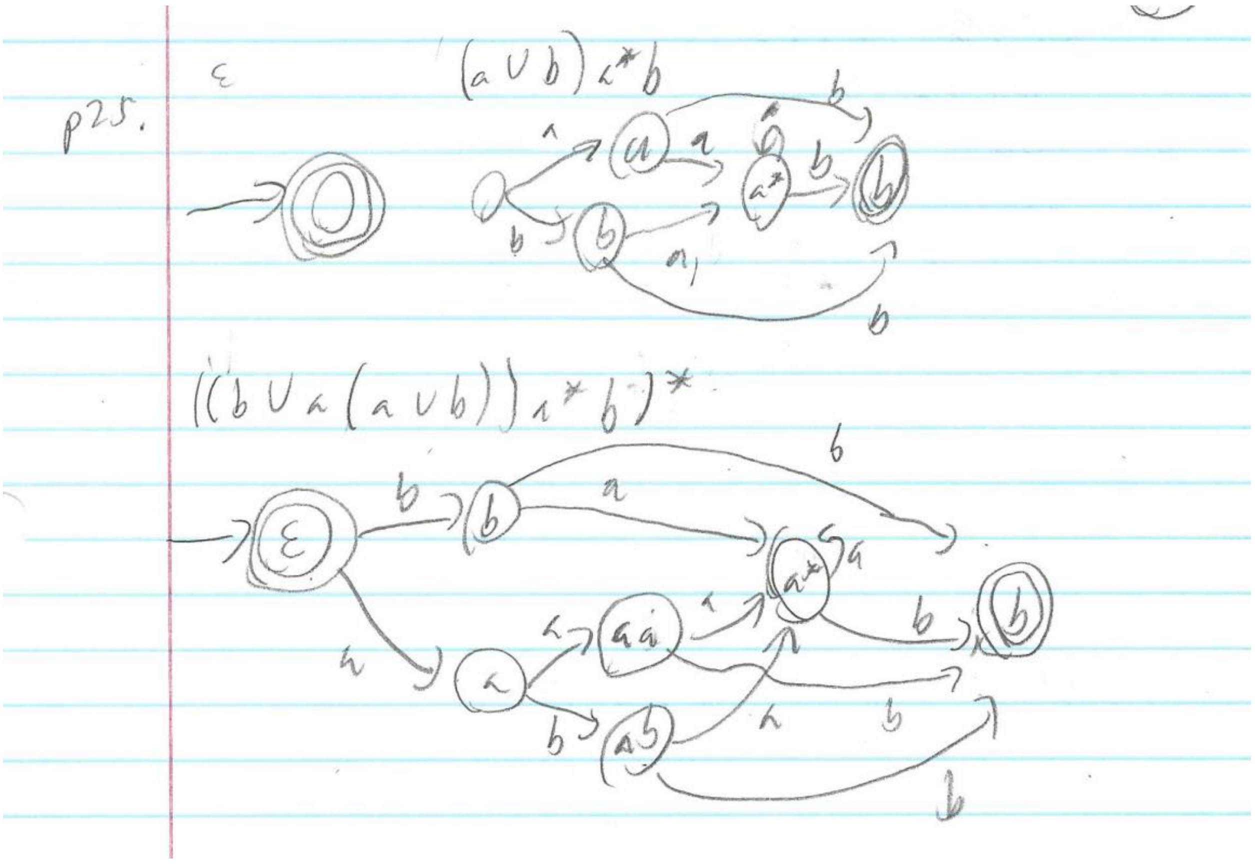
P23:

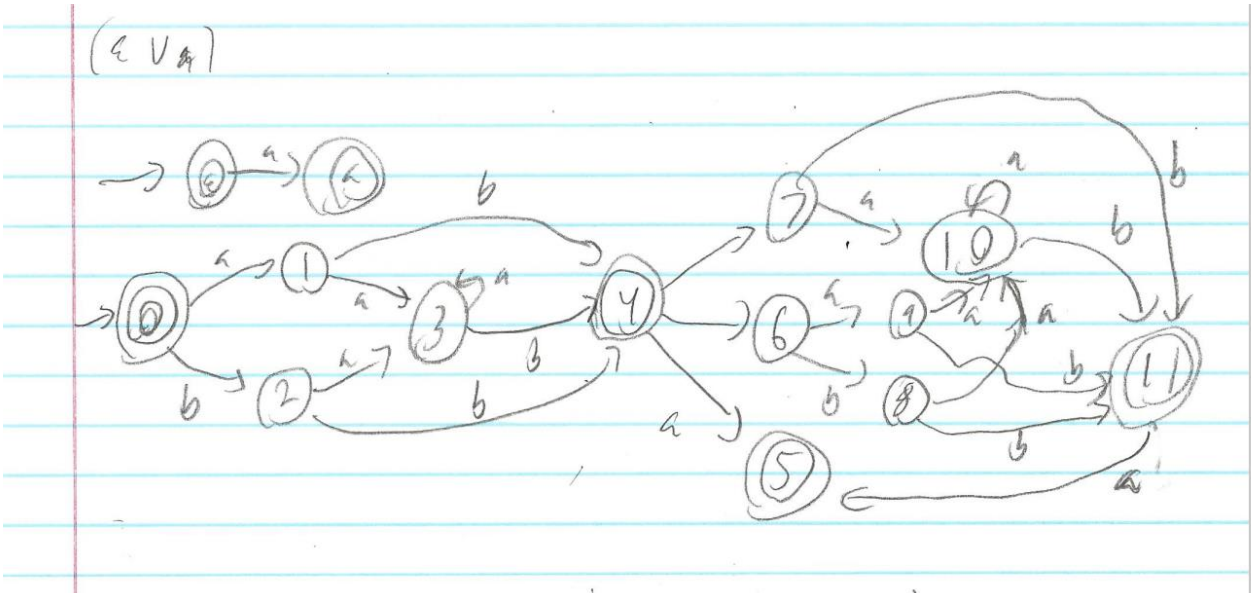


The basic idea for this machine was to only start tracking a's if b's were seen and a's followed because empty is acceptable, so is only a's and so is only b's but once we had b's then a's then more b's we had to start keeping track. State 7 is a state that has more a's than allowed but it is accepting because if no b's are after it than it breaks no rules about a's seperating b's. Same with 2 and 3 as they have too few a's but as long as no b's they are fine.

P24: For p24 I did not make a diagram, but the concept is simple I can read as many interger or hex digits as I want as it was not spicified that hex numbers have the prefix of 0x and octal is not accepted. Then I read these digits in a non-accept state until I read a period after which I move to an accept state. In this accept state I keep reading deciaml and hex digits, but any time I read an e or an E I move to a new non accept state that lloks to see if that symbol is followed by a + or a – to handle those cases and if it does it goes back to the accept state and keeps burning.

P25:





The diagrams show my thought process which was to make all the separate parts fisrt and then put them together in the final machine. The First check was to see if epsilon or (a ∪ b)a*b as that was the simplest starting point for the expression. Then I checked (b ∪ a(a ∪ b))a*b)* to see how that machine could be made and then I finally made epsilon or a, and then I put the machines together either by using epsilon transistions for union or by using concatination methods to combine them. The multiple accept sates handle all the different ways the string could choose to go in order to be formed.