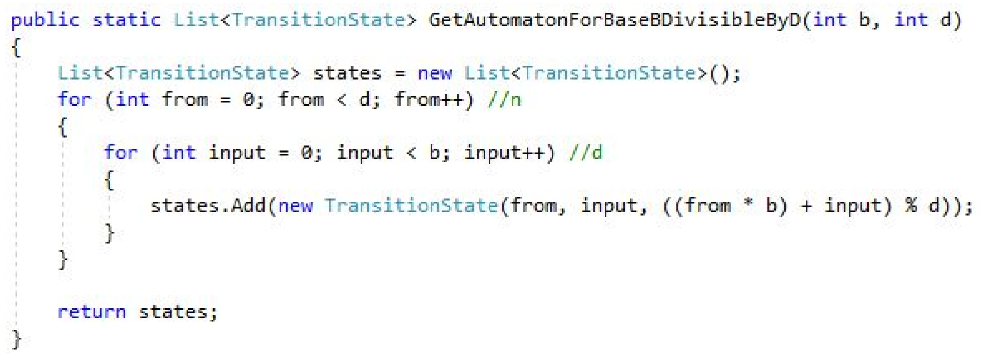
Ryan Darras, CS 5070 - PJ02

*Used a combination of https://www.cefns.nau.edu/~edo/Classes/CS315_WWW/Tools/fsm.html and paint.net to create images

<u>P01 = M(15,10,M,T)</u>

<u>P01 Answer</u>

To find the modulus of any base and any divisor, you can simply plug it into an algorithm that looks like this:



```
public static List<TransitionState> GetAutomatonForBaseBDivisibleByD(int b, int d)
{
    List<TransitionState> states = new List<TransitionState>();
    for (int from = 0; from < d; from++) //n
    {
        for (int input = 0; input < b; input++) //d
        {
            states.Add(new TransitionState(from, input, ((from * b) + input) % d));
        }
    }

    return states;
}
```

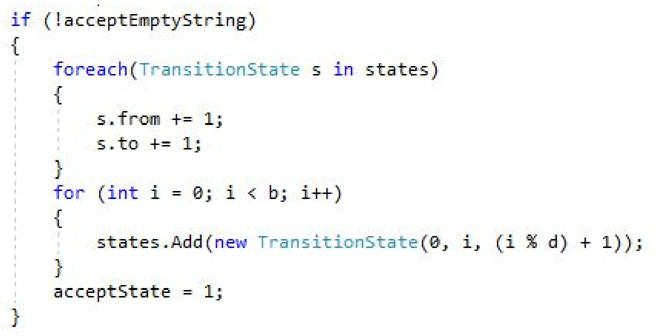
Where b is the base, and d is the divisor (b = 15, d = 10 in this case).

You loop through “d” times with variable from, because if we are trying to find out if our number is divisible by 10, then we are going to need 10 states. One for each possible modulus. We then have to iterate through “b” using variable input to account for each of the possible inputs that we can handle when we are stationed at each node. To find out which node we go to when we have the current node and the input, we plug it into equation “((from * b) + input) % d”.

<u>P02 = M(15,10,M,F)</u>

<u>P02 Answer</u>

This answer is almost identical to the previous question (P01), however the only difference is that the previous question will accept the empty string. To solve this problem, all we have to do is create a new start node that is not an accept state. From it, we create transitions to every other node based on the first input character. The following code demonstrates this practice:



```
if (!acceptEmptyString)
{
    foreach(TransitionState s in states)
    {
        s.from += 1;
        s.to += 1;
    }
    for (int i = 0; i < b; i++)
    {
        states.Add(new TransitionState(0, i, (i % d) + 1));
    }
    acceptState = 1;
}
```

All we are doing is incrementing every nodes number up by 1, and then adding in the 0 node (start node) with transitions to every node based on its input.

P03 = M(4,7,L,F)

P03 Answer

I had some trouble with this one figuring out how to instead read the string least significant digit first. I was also confused on how a number that is in base 4 can even be divisible by 7 when 7 isn't in base 4. The base 4 version of 7 in base 10 would be 13. Does this mean that we need to turn our base 4 value into base 10 and then find if it is divisible by 7? Anyways, I just ran the above algorithm on this one as well to get the result. I'm 99% positive it isn't going to be correct when read least significant digit first, but it should be correct when read most significant digit first.

P04 = M(37,143,M,T)

P04 Answer

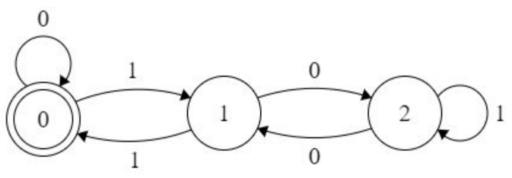
This answer is identical to the answer for question P01.

P05 = {w | w is base-2 divisible by 3 AND w^r is base-3 not divisible by 2}

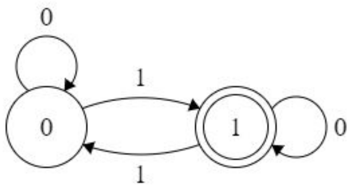
P05 Answer

To solve this, I first found each part individually as a dfa. I also assumed that even though w^r is base 3, its language is still 0,1.

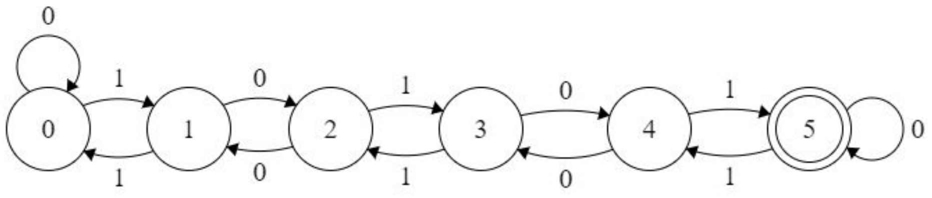
W is base-2 divisible by 3:



W^r is base-3 not divisible by 2:



Then I took the intersection of them to get:



I then manually encoded the transitions into the file.

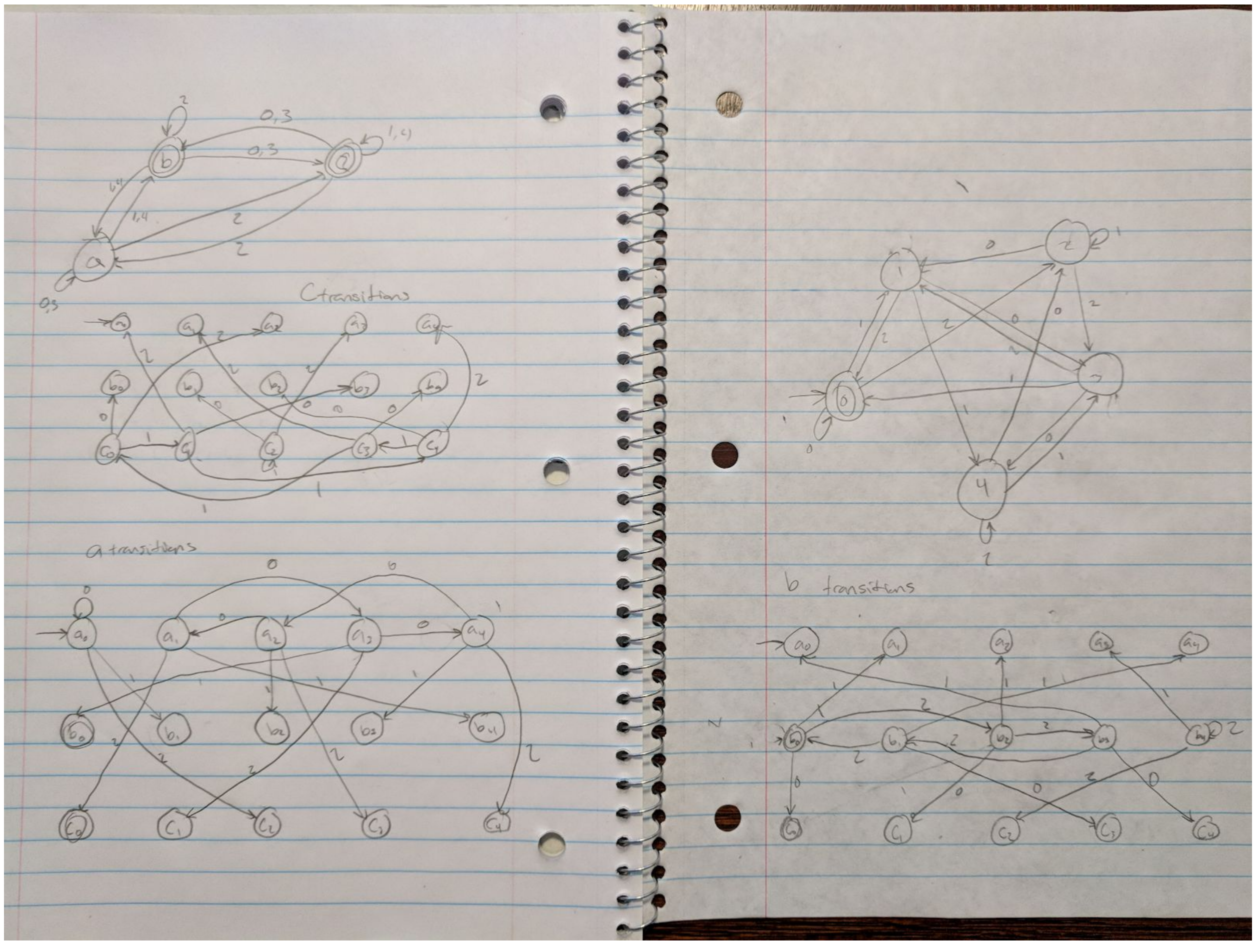
P06 = {w | w is base-3 divisible by 5 AND w^r is base-5 not divisible by 3}

P06 Answer

Same thing as previous answer, but much more work. I used my program to generate P01, P02, P03, and P04 to generate the two parts of this question.

Below is all of my work. Note that I did transitions separately so that it didn't get cluttered up, but the 3 similar looking drawings should be the same FA (which is what is encoded in the answer).

The top left FA is (base-5 not divisible by 2)^r and the top right is base-3 divisible by 5.



P07 = {w | w is base-7 divisible by 2 and 3 but not divisible by 12}

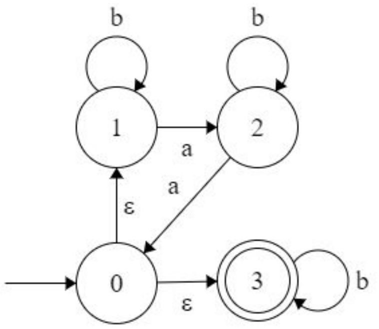
P07 Answer

To sum up how I would solve this problem, I would intersect the three languages L_1 = base-7 divisible by 2, L_2 = base-7 divisible by 3, L_3 = base-7 not divisible by 12. So $L_1 \cap L_2 \cap L_3$. This would result in a potentially massive DFA but it would yield the expected results. If I had extra time, I would write an algorithm to do this for me, but I have 2 research papers to write. =(

P08 = L_1 = (b*ab*a)*b*

P08 Answer

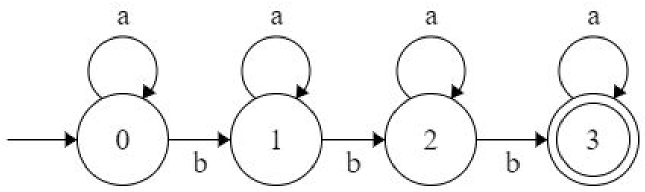
Since everything is *'d, we accept the empty string. To handle the parenthesis, we are going to epsilon into the parenthesis loop, returning back to the start state when finished. To handle the rest, we epsilon transition into the accept state, which loops upon itself when the input is just b



P09 = L_2 = (a*ba*b)a*ba*

P09 Answer

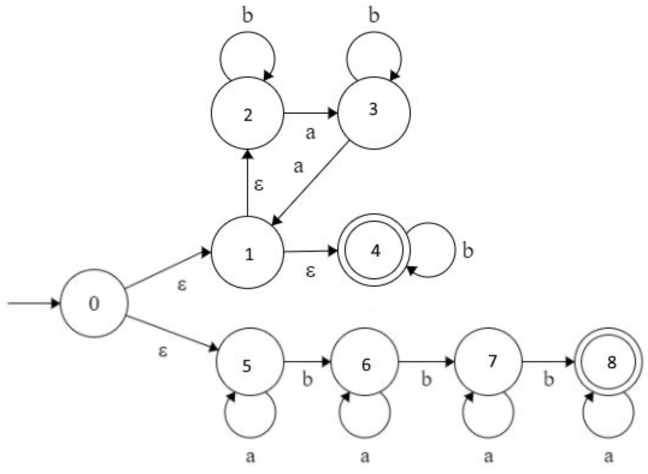
The parenthesis in this equation have no significance to the problem, so we can see that this language will accept any string with only 3 b's and any number of a's.



P10 = $L_1 \cup L_2$

P10 Answer

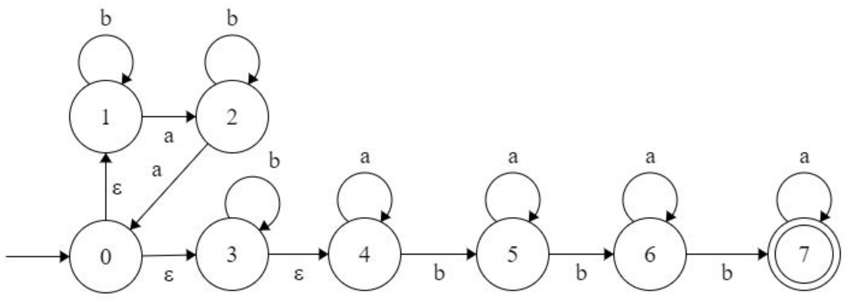
Simple union of the above two answers.



P11 = $L_1 \circ L_2$

P11 Answer

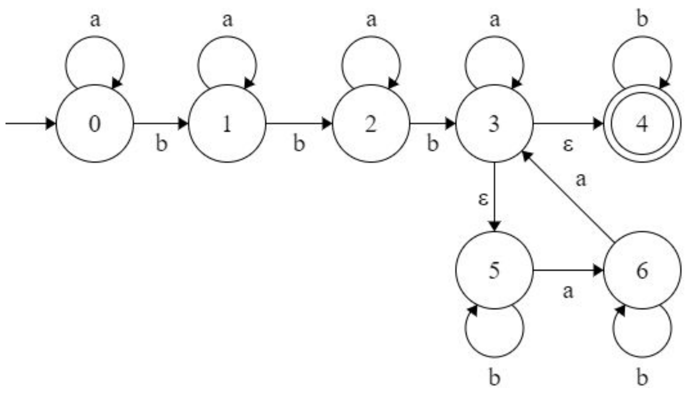
Simple concat of the P08 and P09.



P12 = $L_2 \circ L_1$

P12 Answer

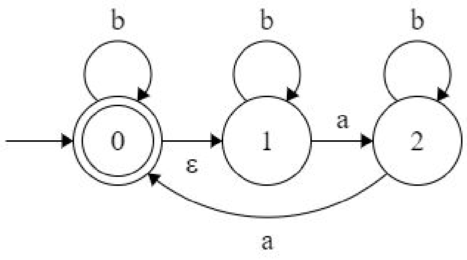
Same thing as 11, but backwards



P13 = L_1*

P13 Answer

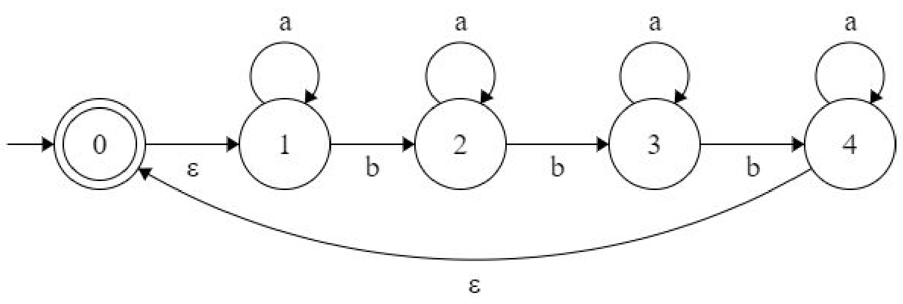
We can take the answer from P08 and connect it back to its starting node. This just causes it to loop, which means it L_1*. Node 0 in this case, handles both the first b* in the parenthesis and the b* at the end. Because they are both b*, and in the case of L_1*, they can be joined to just be a single looping b.



P14 = L_2*

P14 Answer

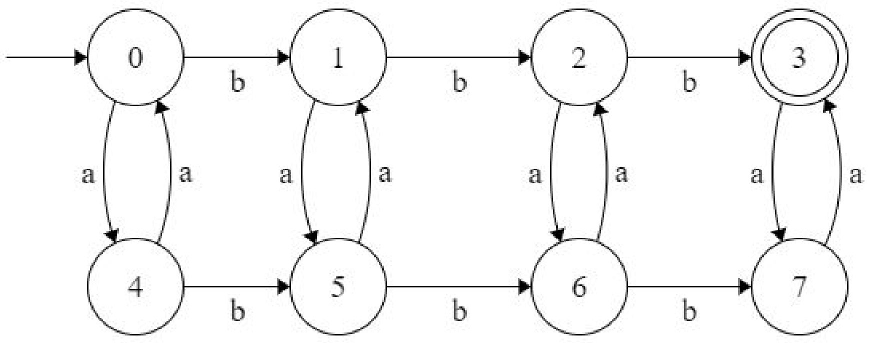
We need to add a transition before we do anything so that we can accept the empty string, because whenever you * a language, the empty string is in it. Then we just do L_2 but make sure to loop back around with an epsilon transition so we can do it again.



P15 = $L_1 \cap L_2$

P15 Answer

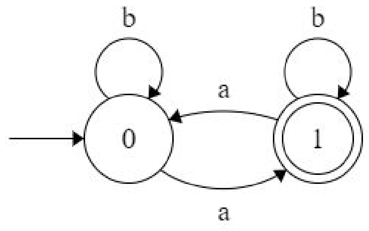
We just need to find the intersection of L_1 and L_2. To do this we just do product construction. Note, I used a minified version of L_1 to calculate this.



P16 = NOT L_1

P16 Answer

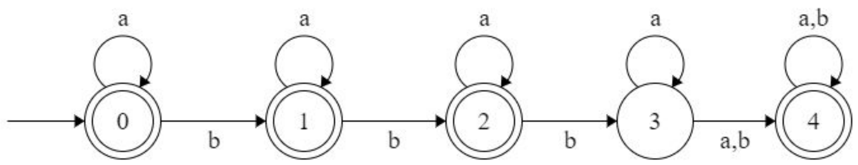
To get this, we just need to alternate states between accept/non-accept. I will be using the minified version I discovered while doing problem 15.



P17 = NOT L_2

P17 Answer

To get this, we just need to alternate states between accept/non-accept and add a state at the end that continues to accept anything after we have passed the original accept state.



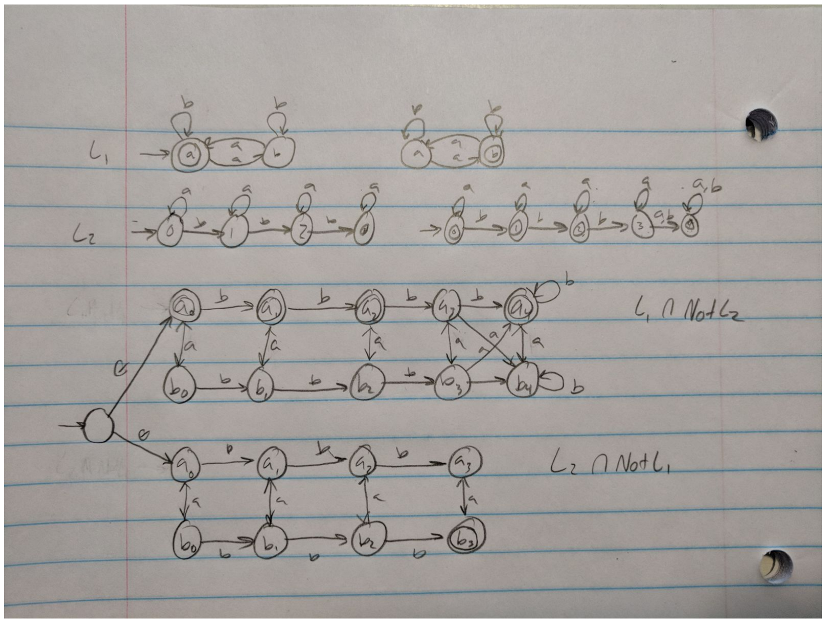
P18 = L_1 XOR L_2

P18 Answer

Simplifying our question into things we have already done before gives us this:

(L_1 ∩ NOT L_2) ∪ (L_2 ∩ NOT L_1), or "The intersection of L_1 and NOT L_2 or the intersection of L_2 and NOT L_1"

Note: The bottom NFA on this screenshot is the solution to this problem.



P19 = L_1 XNOR L_2

P19 Answer

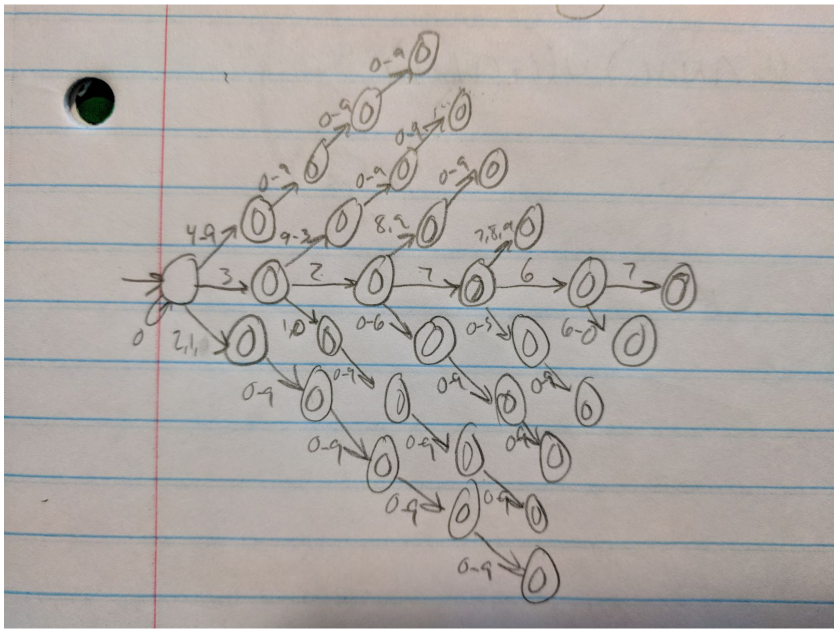
Similar to the previous question, but this time we are going to be taking:

(L_1 ∩ L_2) ∪ (NOT L_1 ∩ NOT L_2). I'm not doing this on paper because it will look almost identical to the above screenshot. Only thing I will be doing is changing around some accept states. The machine file for this question is a direct copy of P18, minus the accept states.

P20 = { n | n is a 15-bit decimal integer constant }

P20 Answer

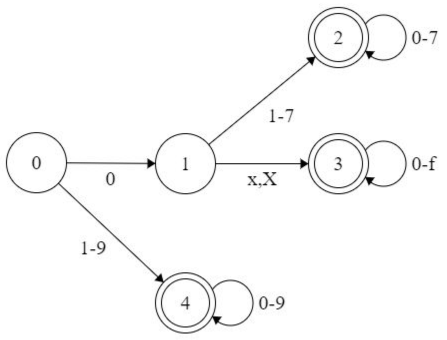
Long story short, I feel like I brute forced this, but it should work nonetheless. If first input is greater than 3, move up. If first input is less than 3, move down. If first input is 3, move forward. Up means that because you were above the first digits highest value to be legal, you can only be a total of 4 numbers long (because 32767 is 5 numbers long). Down means that you are below the leading digits value, so you can be 5 numbers long, doesn't matter what numbers past that either. Forward means you are the same as the maximum, meaning it can continue to go either way.



P21 = { s | s is a C-style integer }

P21 Answer

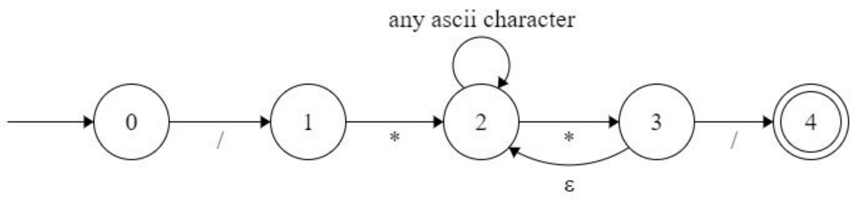
We just need to check the differences at the start of the string for whether it is decimal, octal, or hex. I am also assuming that we are not constraining ourselves to the ~4.1b limit on an unsigned integer in C. Decimal can start with 1, 2, 3, 4, 5, 6, 7, 8 or 9. Both octal and hex start with 0, but hex follows it up with an X or x. Then for octal, we can only accept 0-7, and hex we can only accept 0-f.



P22 = { s | s is a C-style block comment }

P22 Answer

Fairly simple, we just check to see if it leads with /* and ends with */ I've removed "`" from the ascii character list because it is what we use for these programs to simulate epsilon. Note that we also have an epsilon transition from 3 to 2. This is so that we don't automatically close out any time we see a *. We must see the */ next to each other. Also note that some characters weren't recognized by my notepad so I removed them.

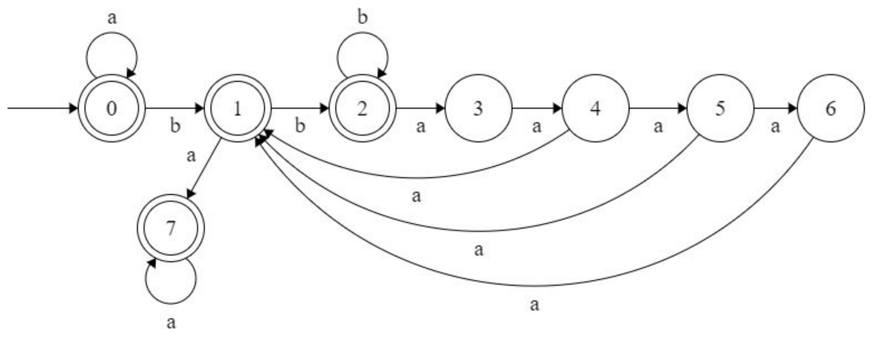


.

P23 = { w | w ∈ {a,b}* and every group of consecutive b's is separated by at least three, but no more than 5 a's}

P23 Answer

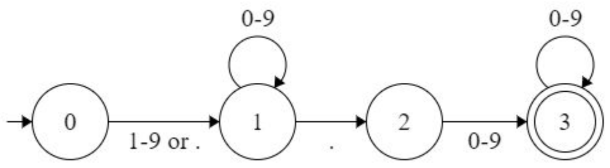
Fairly simple, albeit its ugly and I bet there are better ways to do it, but you just create three separate paths between the B's. One path for 3 a's, one for 4, and one for 5. If it meets our requirement, then the machine will proceed. We branch off and accept states that loop a's because the language can possibly accept a string with only a's in it, and when the string comes to the last b, it can still have as many a's remaining as it wants because it only wants 3,4,5 a's between b's. I wasn't entirely sure if I needed to verify somewhere that it was indeed a group of consecutive b's or if a single b was good enough.



P24 = { x | x is a C-style floating point constant }

P24 Answer

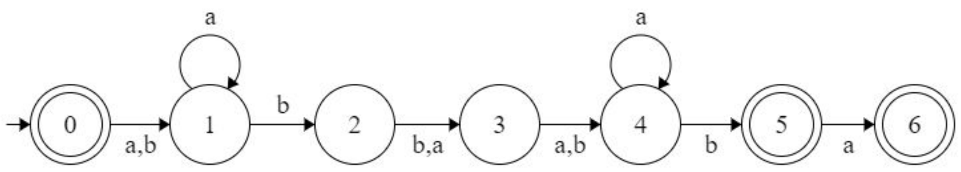
This is assuming that you do not need a leading 0 if the value is < 1, "IE 0.5". This is also ignoring any truncation after a certain amount of precision digits. If we were to take into account the procision digits, we would replicate the transition 2 - 3 a number of times at which point we would break off at a certain point and round to the nearest 10th.



P25 = ε∪((a∪b)a*b((b∪a(a∪b))a*b(ε∪a))

P25 Answer

Because of the leading epsilon, we can say that we must accept the start state. I am also assuming that in the middle of the expression “b∪a(a∪b)” can be more clearly rewritten as “(b∪a)(a∪b)” because of the order of operations. This may or may not throw off my entire machine, but it is unclear to me if it should be this way or not.


0
a,b
1
a
b
2
b,a
3
a,b
4
a
b
5
a
6