Regular Expressions
An Alternate Approach by Bob Mathews

In this month's challenge, we were asked to check a number of constraints with regular expressions.

For example, “the string does not contain 78.”

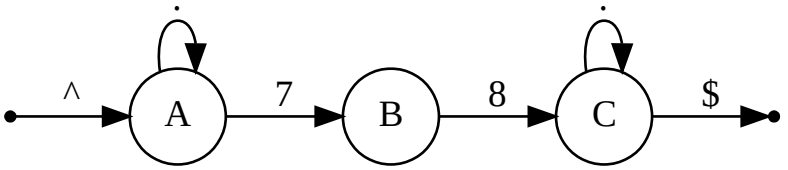
Of course, that can be done with `!/78/`.

But it turns out that we can push the `!` operator inside the regular expression, without using any Perl regex extensions. Then we can play some other tricks.

This is going to require a completely different way of thinking about regexes, so hold on to your hats.

First, I'm going to write that pattern as `/^.*78.*$/`, for Reasons.

Finite State Machine



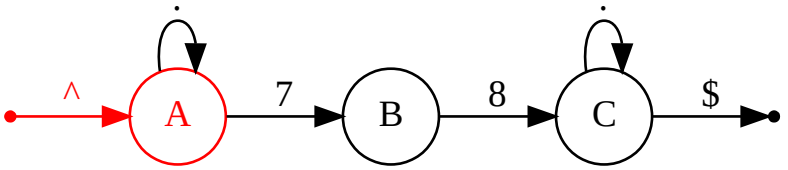
This is called a Finite State Machine (FSM).

It has a set of states, connected by arrows (known as “directed edges” in graph theory jargon).

The edges are labeled with regular expressions.

The edge labels are important to the operation of the machine, but the state labels are only there for descriptive purposes.

Finite State Machine

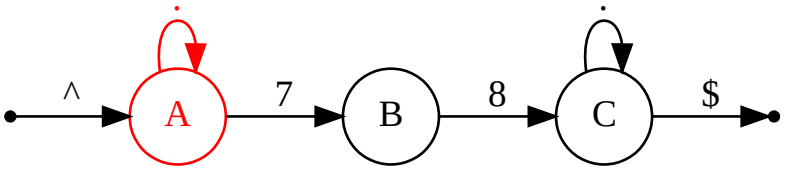


x78x

Let's try it out with a short input string.

Start in state A...

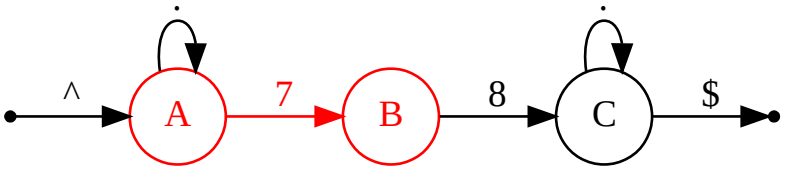
Finite State Machine



x78x

When we see x, go around the loop and back to A.

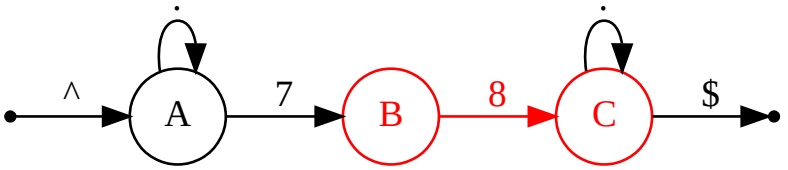
Finite State Machine



x78x

7 advances the machine to state B.

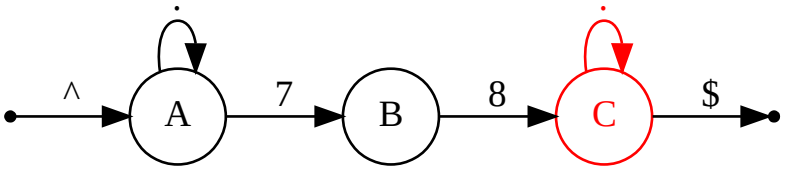
Finite State Machine



x78x

8 advances it to C.

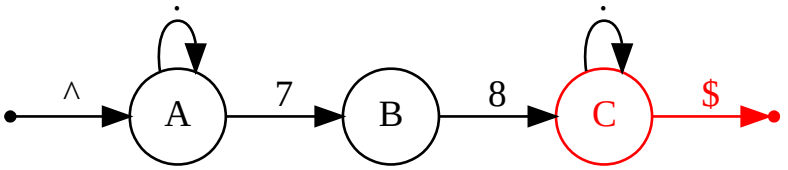
Finite State Machine



x78x

x goes around the loop back to C.

Finite State Machine

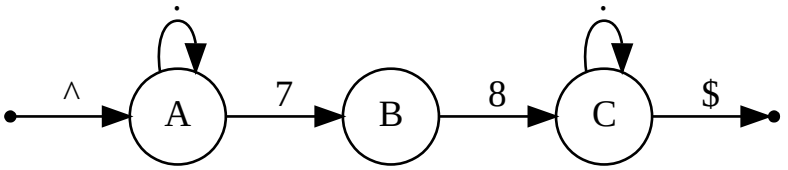


`x78x`

At the end of the string, move to the final state.

We say that the machine “accepts” the string **`x78x`**.

Finite State Machine



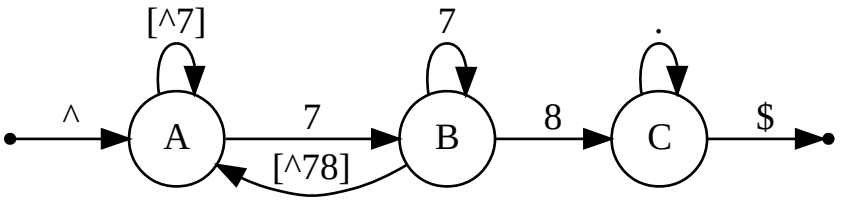
Complication: When we see a 7, do we go to state B or stay in A? The labels allow both moves.

With input **`x7x78x`**, stay in A on the first 7. Move to B on the second 7.

If there is more than one allowable choice at some point, the machine is called “nondeterministic.”

Perl's regex engine handles this by backtracking. It tries to match the entire string with the A-loop, then backs up to the second 7 and advances to B.

Deterministic Finite Automaton



We can modify the machine so that there is never any need to guess the right move.

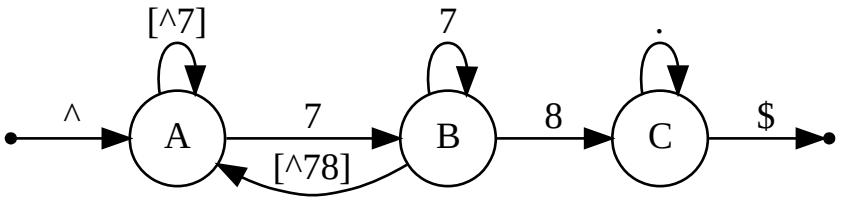
There is an algorithm for doing this automatically, but in this case it's not too hard to come up with the answer by hand.

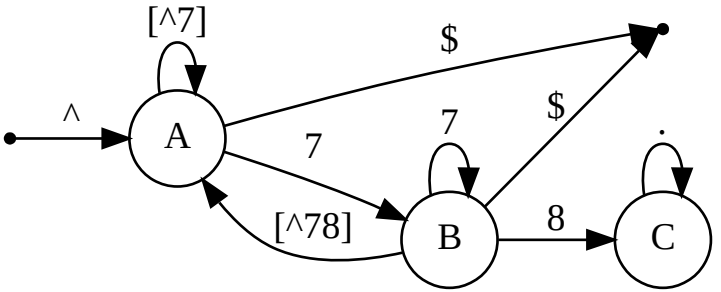
Always advance from A to B on a 7.

If we're in state B and we get another 7, just stay there.

If B gets something that's not 7 or 8, go back to A.

Invert DFA



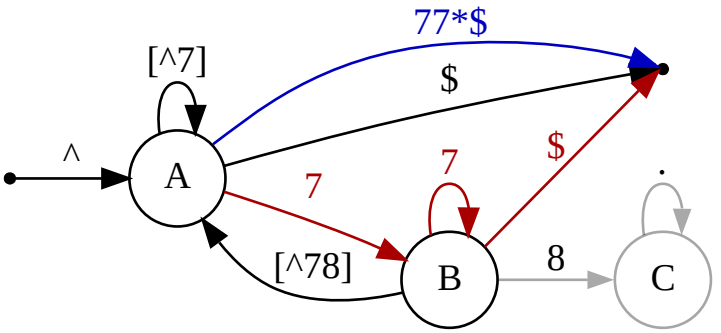


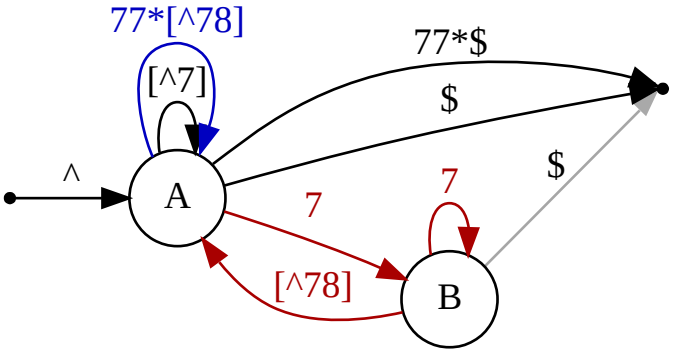
Remove $-edge.

Add $-edges to nodes that didn't have them before.

Resulting DFA accepts exactly what the old one didn't accept.

Convert DFA to Regex





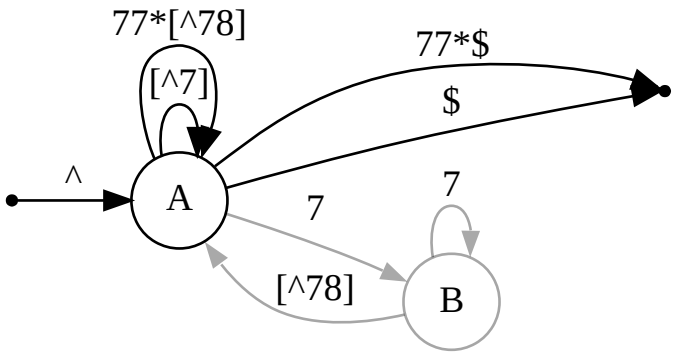
Node C is dead (can't reach End).

Replace A-B-End path (red) with A-End edge (blue). New label combines labels from path.

Replace A-B-A path (red) with A-A edge (blue).

Now we can delete node B.

Convert DFA to Regex



Now there are two edges from A to the end.

Replace them with a single edge, joining labels with |.

Same for the A-A loops.

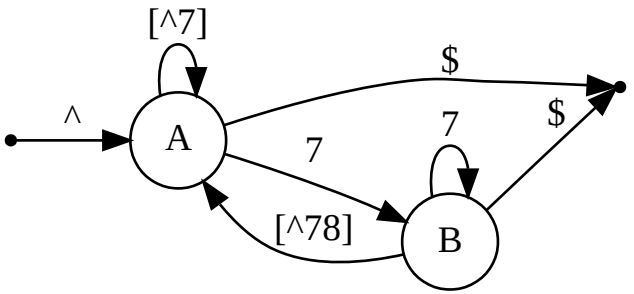
Finally, read off the regular expression.

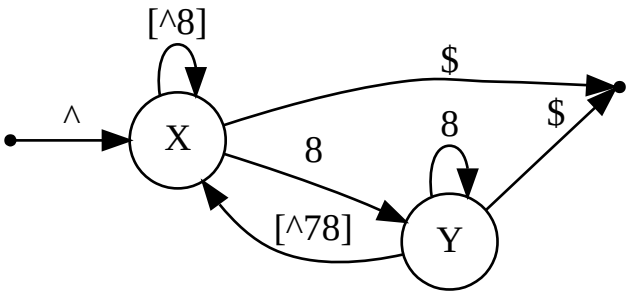
```
^([^7]|77*[^78])*($|77*$)
```

That can be hand-optimized to:

```
^([^7]|7+[^78])*7*$
```

Combine DFAs



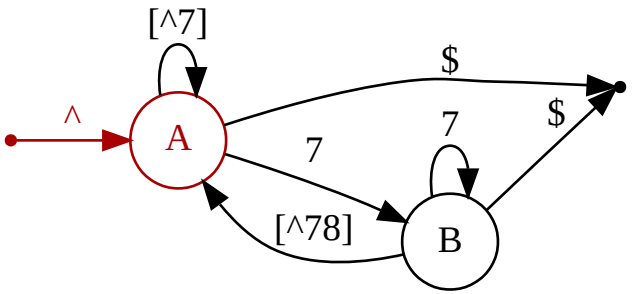


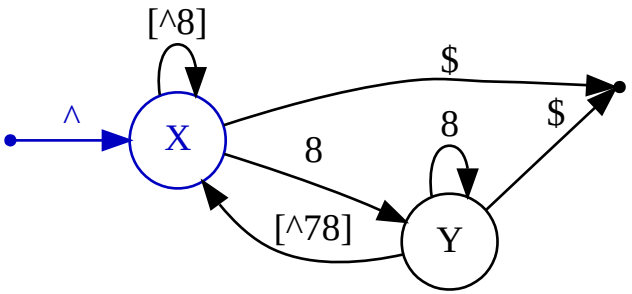
DFA AB accepts strings that do not contain 78.

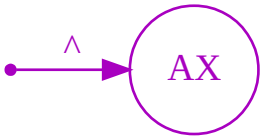
DFA XY accepts strings that do not contain 87.

We're going to combine these two into a single machine that accepts strings that do not contain 78 or 87.

Combine DFAs

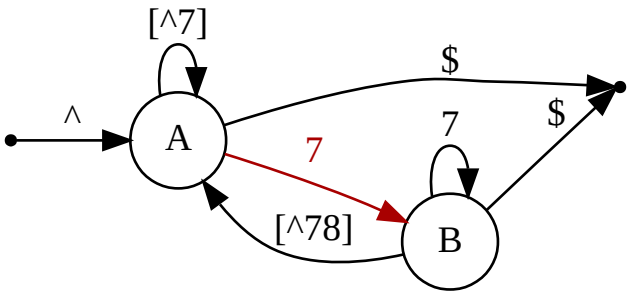


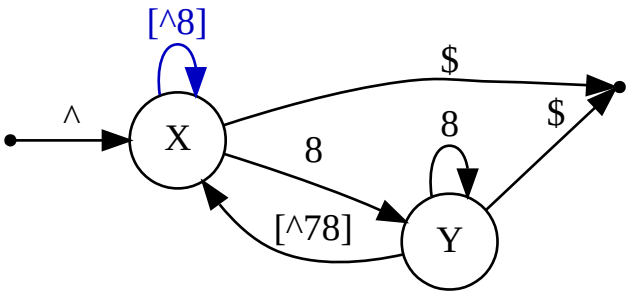


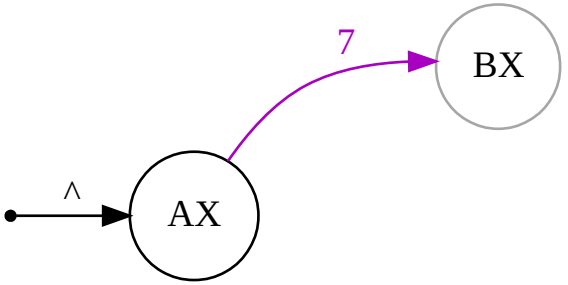


Start with a node that combines node A from the first DFA with node X from the second one.

Combine DFAs

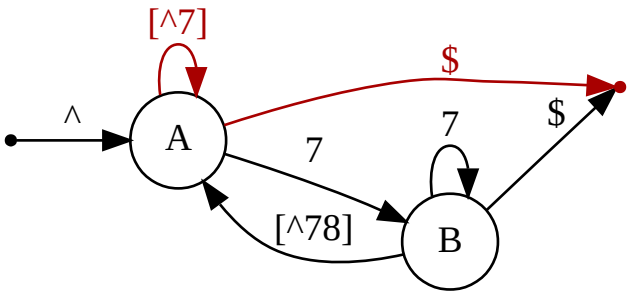


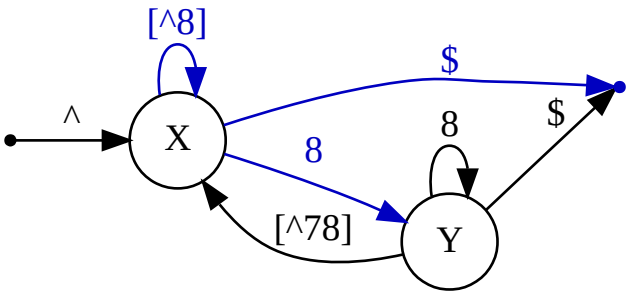


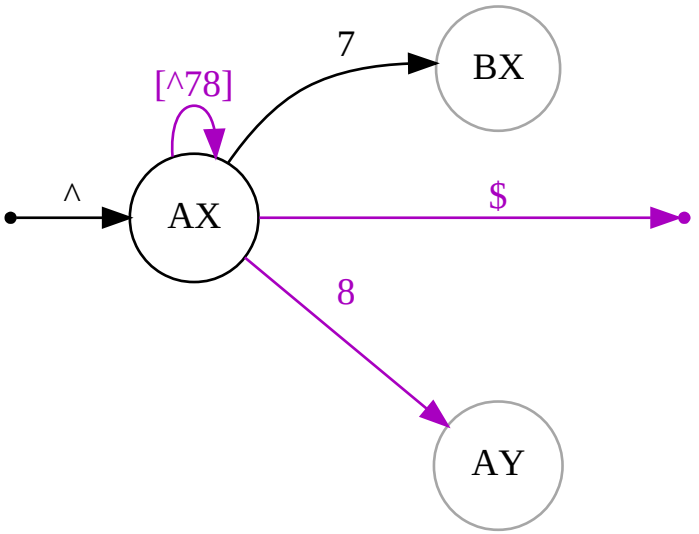


On a 7, A becomes B and X stays X.

Combine DFAs

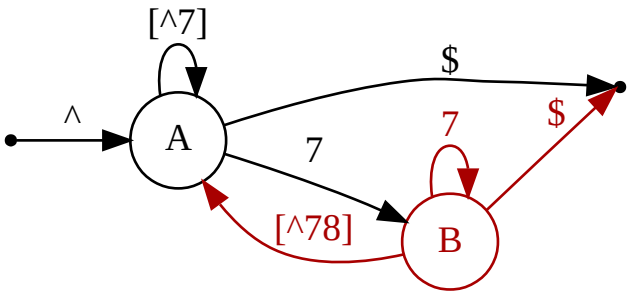


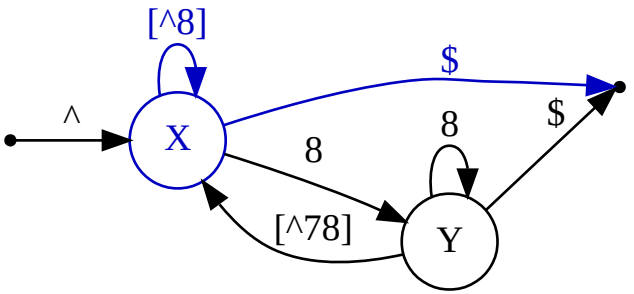


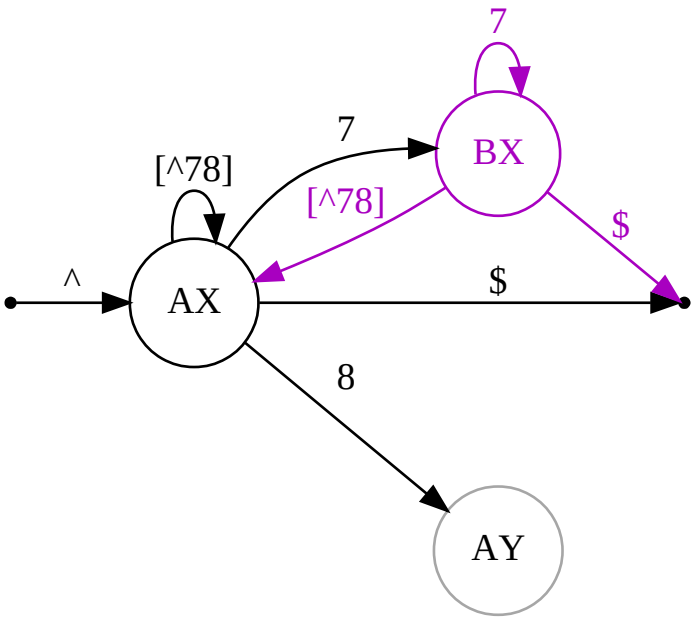


Add other edges from node AX.

Combine DFAs

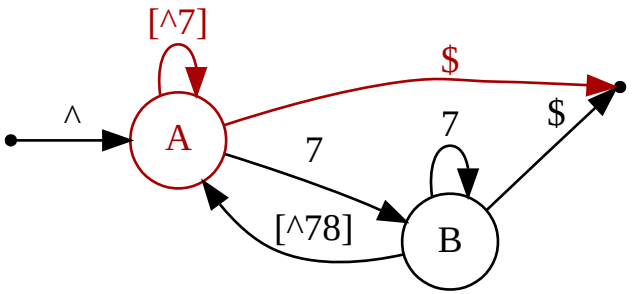


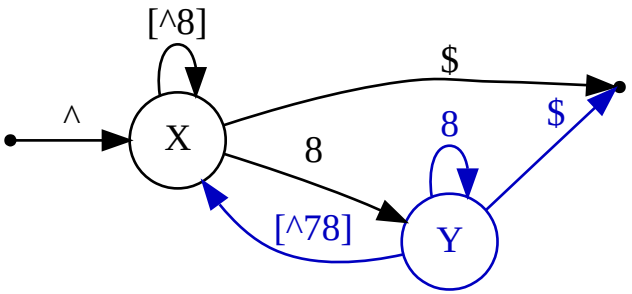


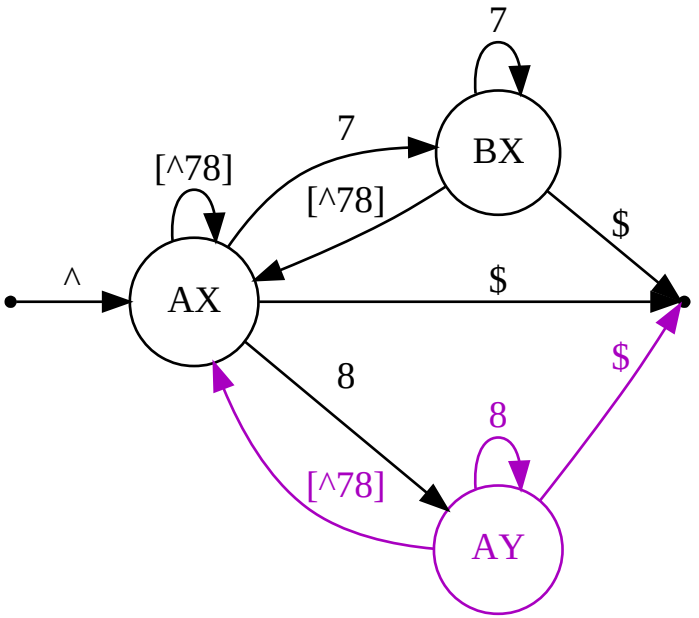


Build node BX and its edges.

Combine DFAs







Add AY.

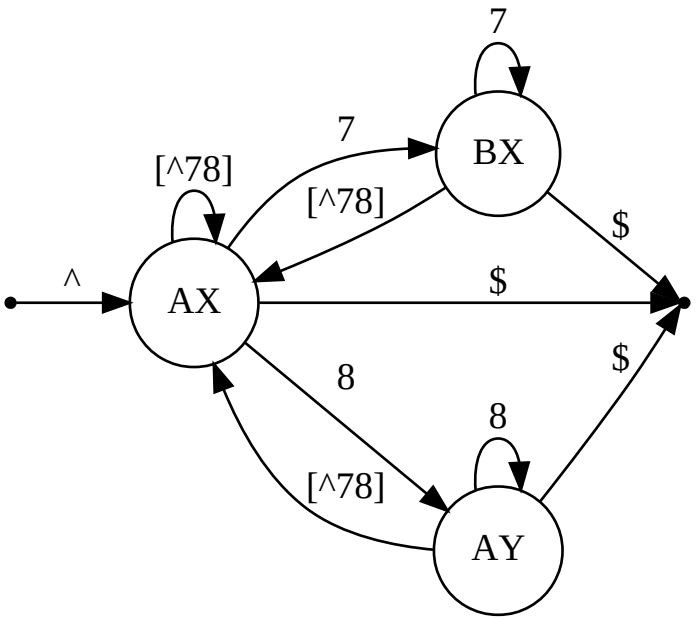
There is no BY.

Combine DFAs

This DFA accepts strings that do not contain 78 or 87.

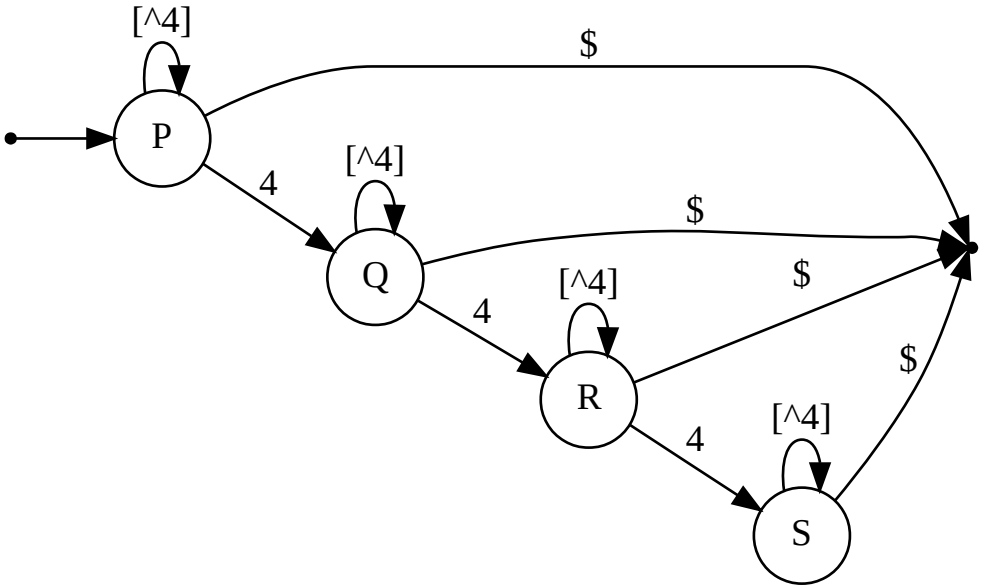
We can get the regex using the same method as before.

```
^((7*|8*)[^78])*(7*|8*)$
```



Keep Going...

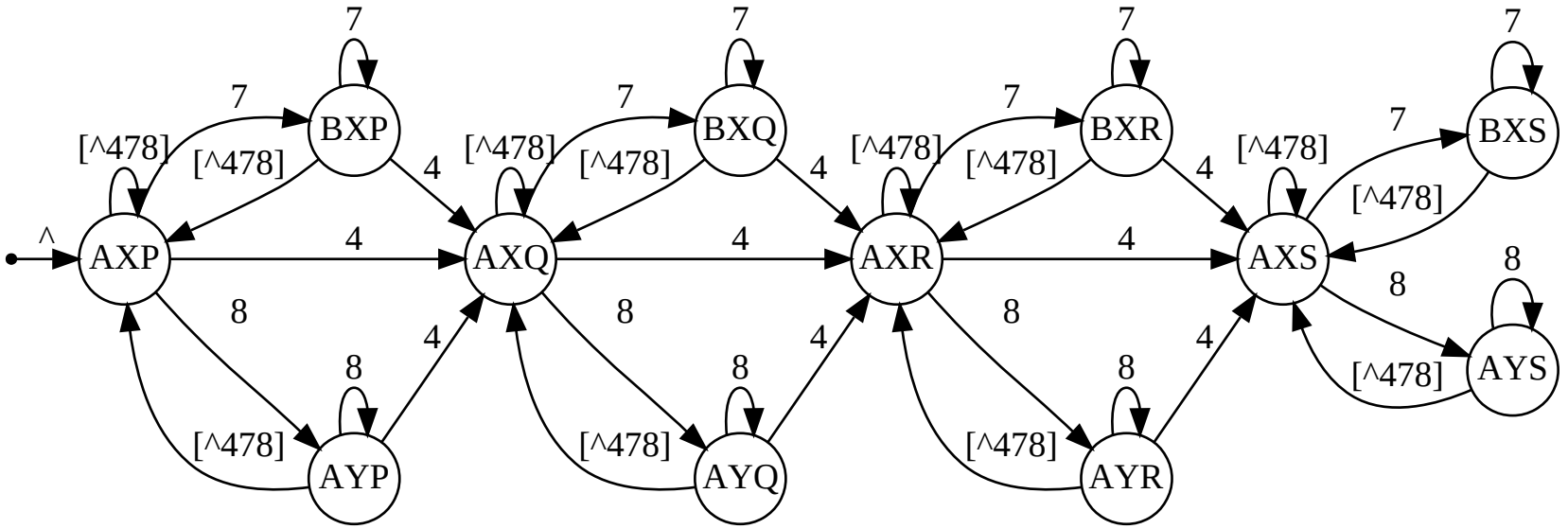
Here is a machine that implements the rule,
“4 cannot occur more than 3 times.”



Regex: `^[^4]*(|4[^4]*(|4[^4]*(|4[^4]*)))$`
Or: `^[^4]*(4[^4]*){0,3}$` (Perl-specific extension)

When we combine this with the previous machine, we get...

A Big Mess



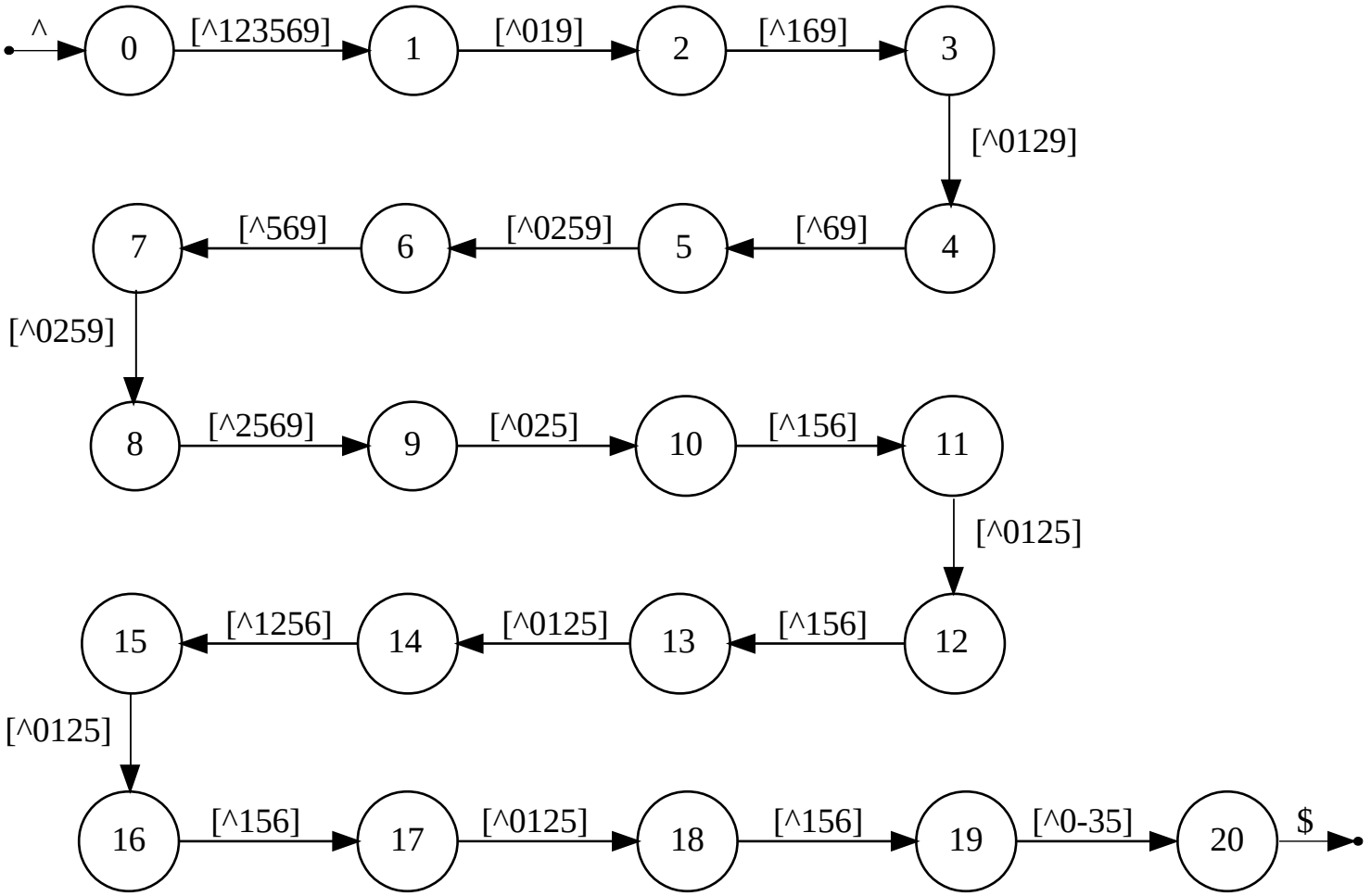
There's a $-edge for every one of those states,
but I've left them off, to keep the mess in check.

Regex: `^((7*|8*)[^478])*(7*|8*)(|4((7*|8*)[^478])*(7*|8*)(|4((7*|8*)[^478])*(7*|8*))(|4((7*|8*)[^478])*(7*|8*)))$`

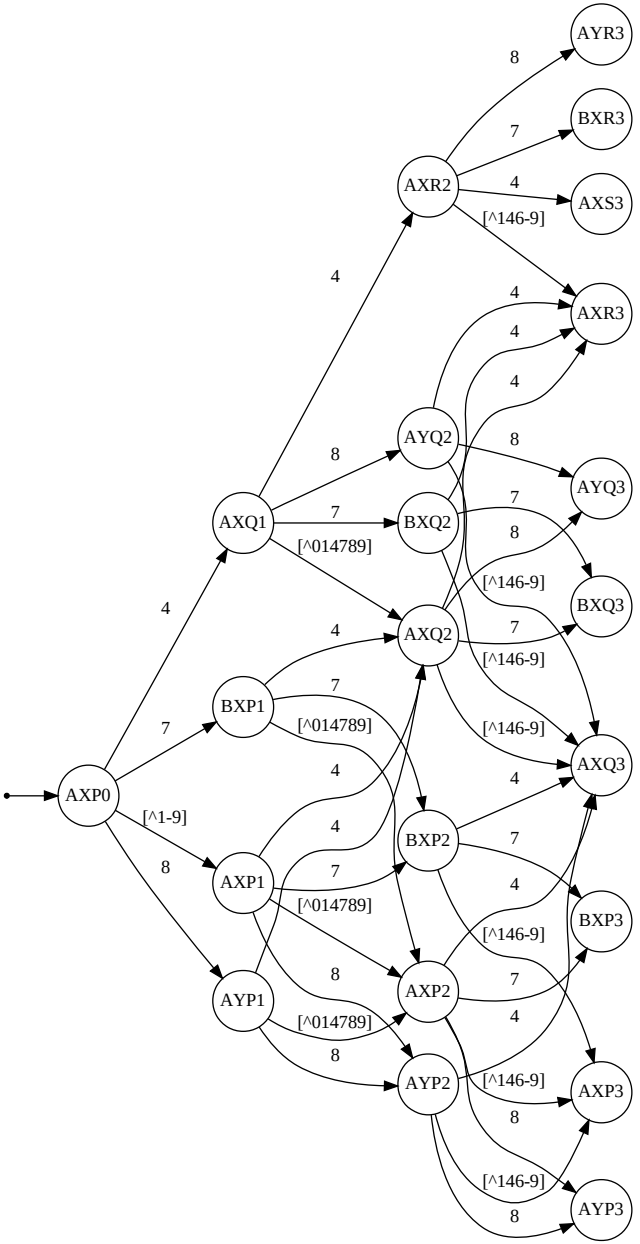
Perl: `$r='((7*|8*)[^478])*(7*|8*)'; /^$r(4$r){0,3}/;`

Not Finished Yet...

This combines all the “can only appear in these positions” rules.



A Big Ugly Mess



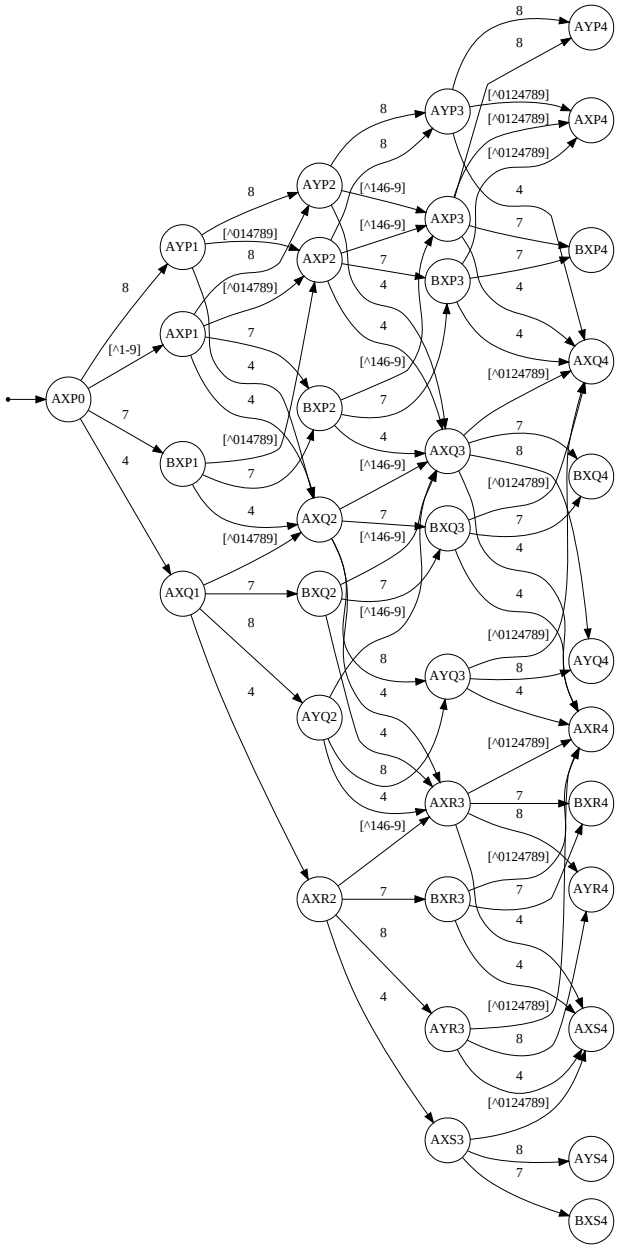
When we combine those two machines, things start to get really bad.

This is only the first few levels of the resulting monstrosity.

A regex for this part is:

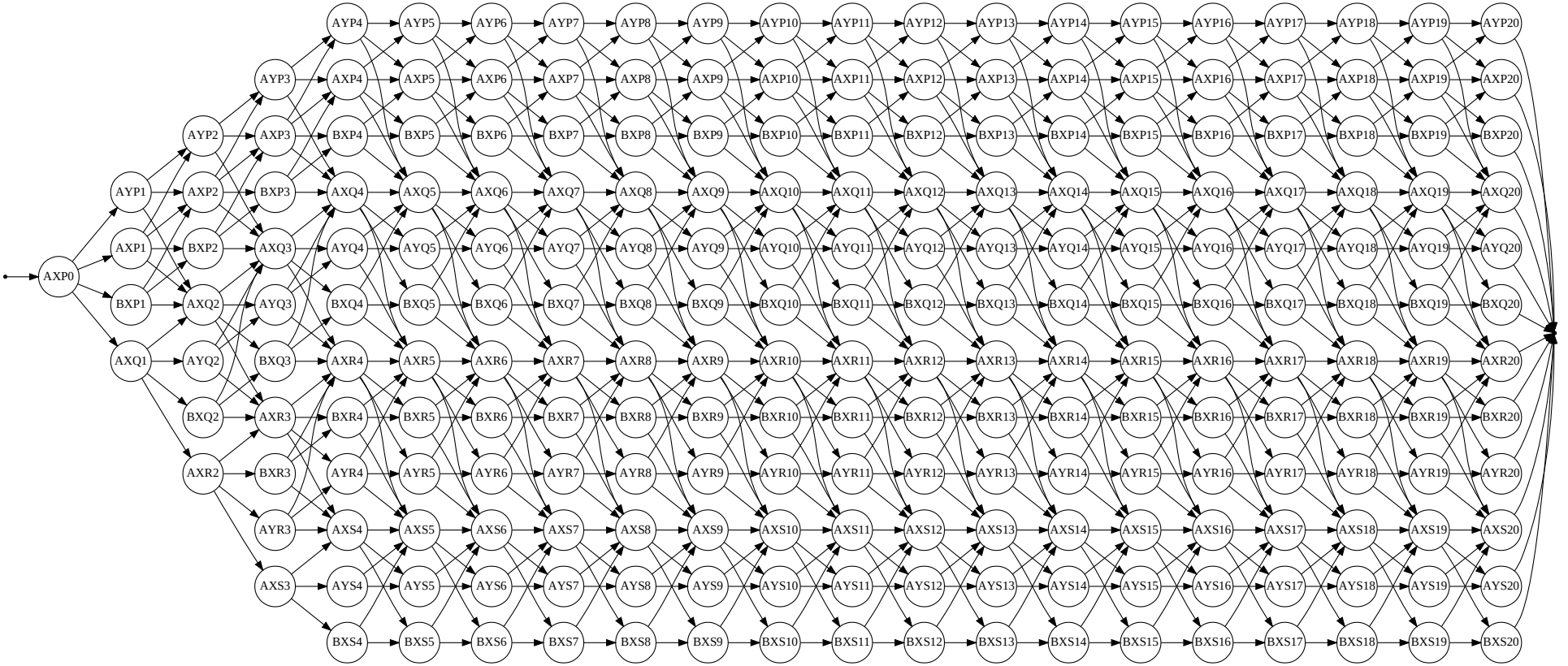
```
^([^1235689]7[^1689]
|[^1235679]8[^1679]
|[^123569][^01789][^169])
```

A Bigger Ugly Mess




```
^([^1235689]77[^01289]
|[^1235689]7[^16-9][^0129]
|[^123569][^01789]7[^01289]
|[^1235679]88[^01279]
|[^1235679]8[^16-9][^0129]
|[^123569][^01789]8[^01279]
|[^1-69][^01789][^16-9][^0129]
|[^123569][^014789][^16-9][^0129]
|[^123569][^01789][^146-9][^0129]
|[^123569][^01789][^16-9][^01249])
```

A Huge Ugly Mess



There are 226 states in this machine.
The edge labels have been omitted for sanity.
The regex grows exponentially, so forget it.

And That's Not All...

Because adding the lower-case and upper-case rules is going to multiply the size of the machine by 6.

That's enough for me.

What have we learned here?

There's a reason we don't do everything with regexes.

It's because regexes are awful.

These techniques are mostly useful if you want to horrify onlookers when someone asks, “can you make a regex to do this?”