Non-Deterministic Finite State Machines

Contents

[Examples 4](#_Toc132659141)

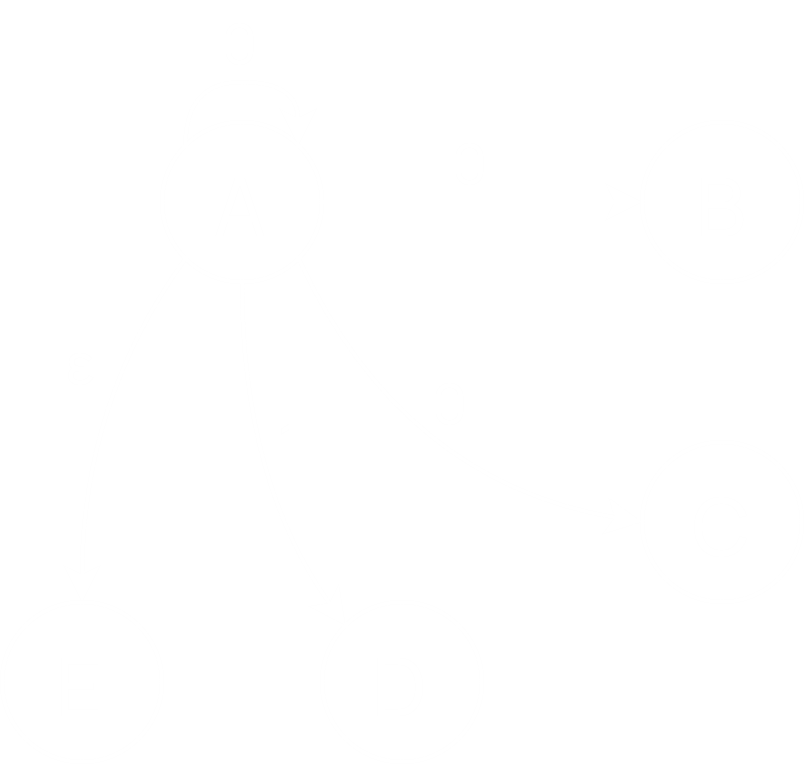
[Epsilon Values 8](#_Toc132659142)

[Concatenating Languages 10](#_Toc132659143)

[Kleene Closure 11](#_Toc132659144)

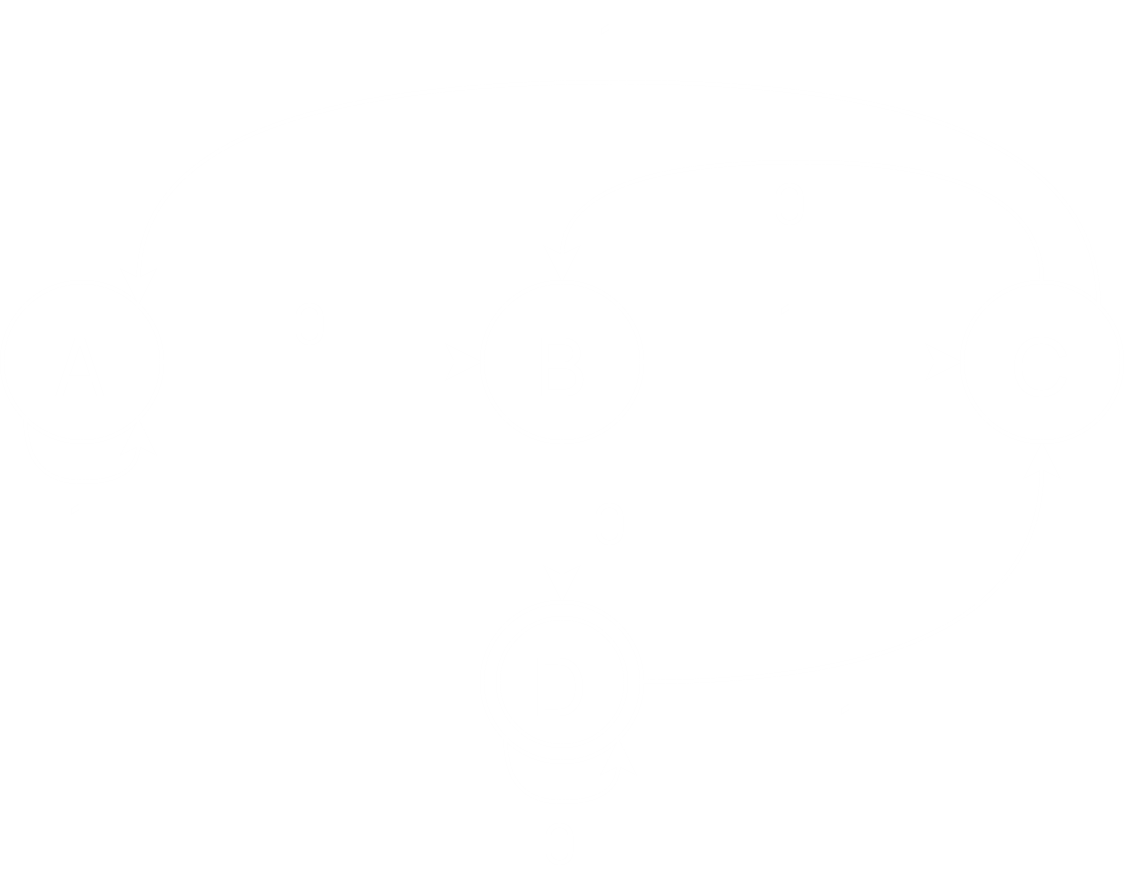
So far, we have been studying **Deterministic Finite State Machines**. The main attribute of these machines is that when we are at a given state and we are provided with an input symbol, we can only move to one other specified state.

The main attributes of **Non-Deterministic Finite State Machines** on the other hand, is that they allow us to move to one of multiple possible states from the current one when provided with some input symbol. In fact, we can even move to a different state without consuming any symbols (denoted by ).

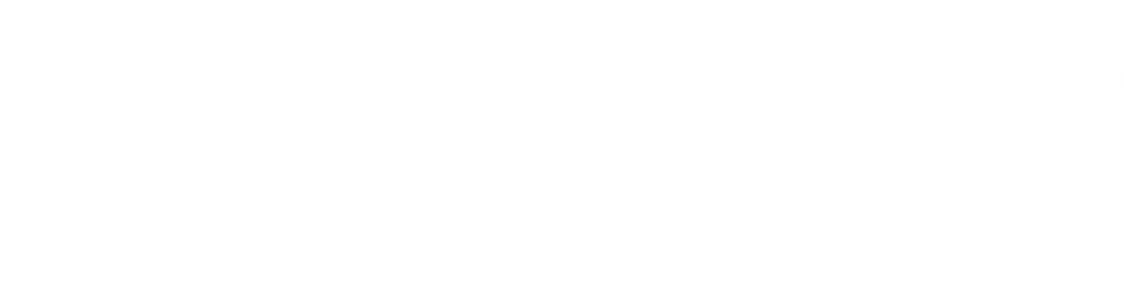


If we have the set of states , the possible states we can move to from the current one are nowhere or to or to or to and both. Thus, the possible states are , i.e., the power set of .

NFAs are frequently far easier to create than DFAs, since they allow us to explore multiple paths in parallel. For example, consider the DFA for a language which accepts all strings where the second to last symbol is .



Now consider the NFA.



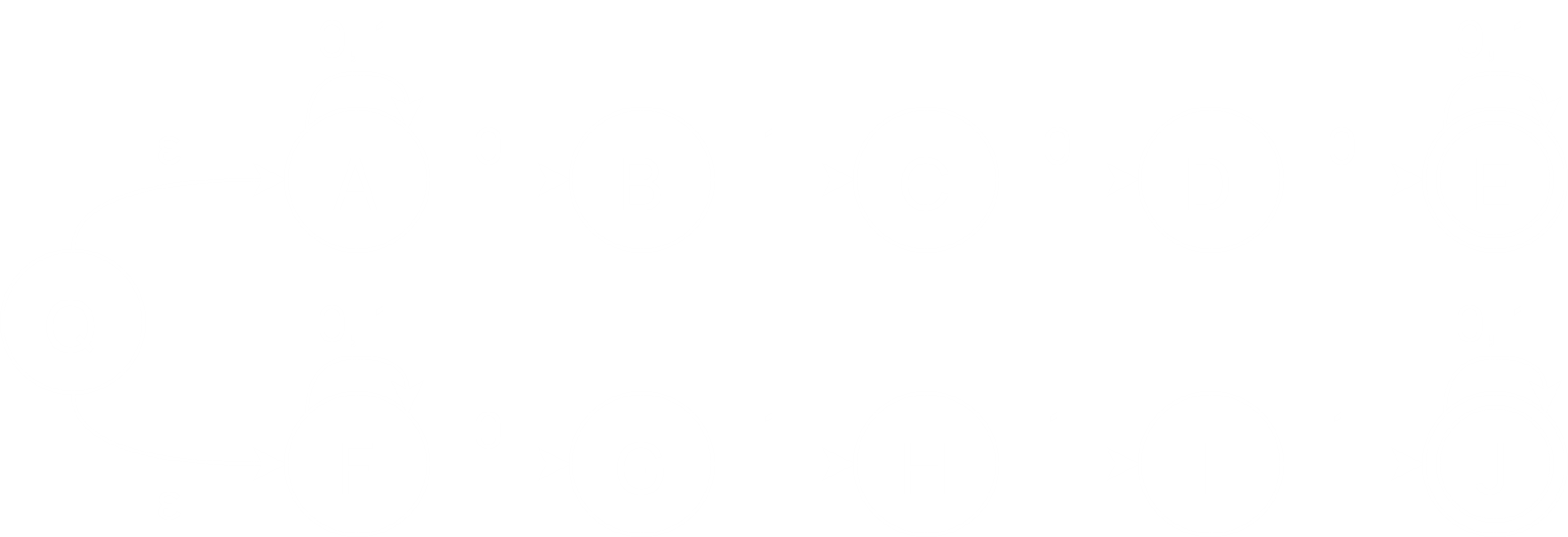
For every possible NFA, there is always a corresponding DFA. However, more likely than not, the DFA will be more complicated.

For NFAs:

* all possible states

## Examples

The machines we created for in particular becomes way easier using NFAs. Consider that we have one machine that accepts strings containing and another machine that accepts strings containing . All we need to do is construct the two machines separately and join them a starting state connected via .

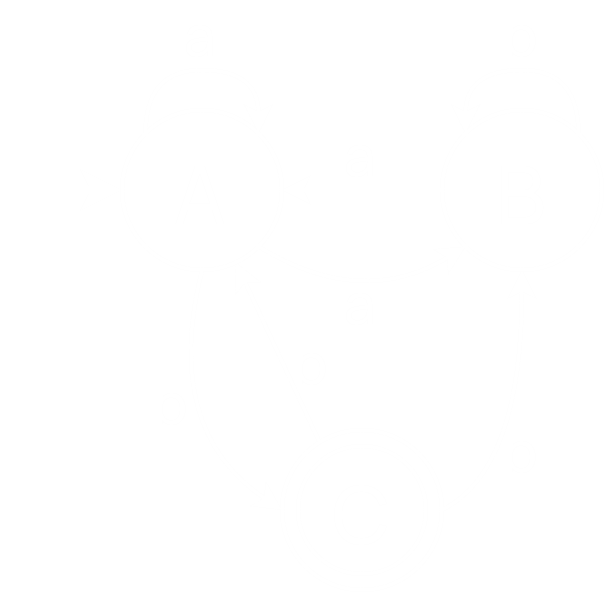


Concatenation also becomes easier. Suppose we want . This means that the input string can be split at some point such that the first half is accepted by and the second half is accepted by . This is difficult to achieve using a DFA, but with an NFA we can just split the string at all possible points and pass them to and simultaneously.

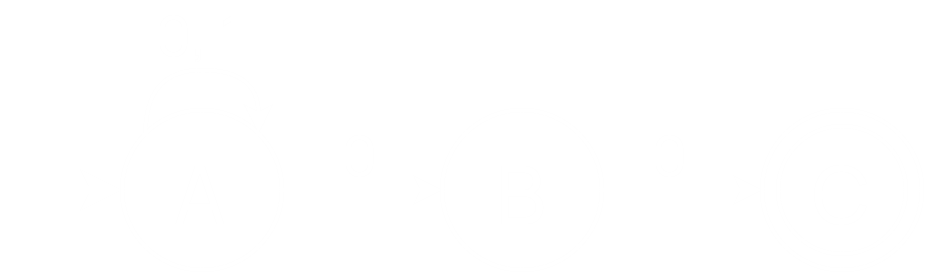
Find the NFA given by

where :

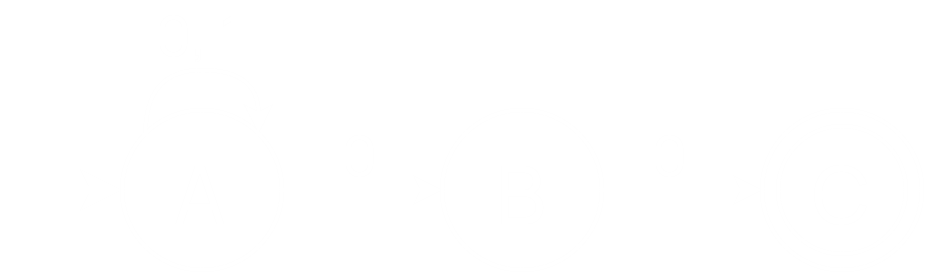
|  |  |  |
| --- | --- | --- |
|  | a | b |
| A | A, B | C |
| B | A | B |
| C | - | A, B |



Find the NFA that accepts all strings over ending with .

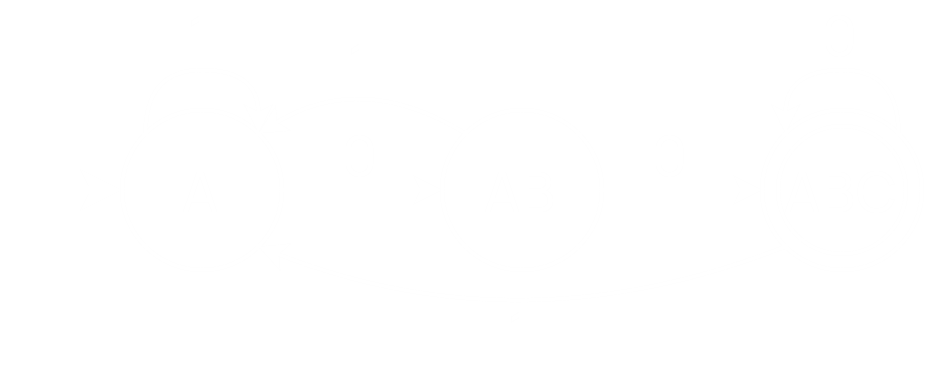


We previously mentioned a theorem, that every NFA has an equivalent DFA. This inherently means that an NFA also requires **regular languages**. Thus, NFAs are not necessarily better than DFAs from a processing perspective. However, if we need to check if a language works with a DFA, it is frequently easier to build an NFA first and check the language there before moving on to build the often more difficult DFA.



For the above NFA, the possible states are . For the equivalent DFA, the states would be . Not all of these states will be present in the actual diagram. The final states in the DFA would be any of the states with in them.

For a given transition, we find the next state based on how many directions we can go to from the current state. For example, if we are currently at and the next symbol we get is a , we can go to either or . As a compensation, we go to .



This was basically **proof by construction**. We can also go about creating this DFA formally.

For the NFA, .

For the corresponding DFA, .

where

, meaning if has states, will have states

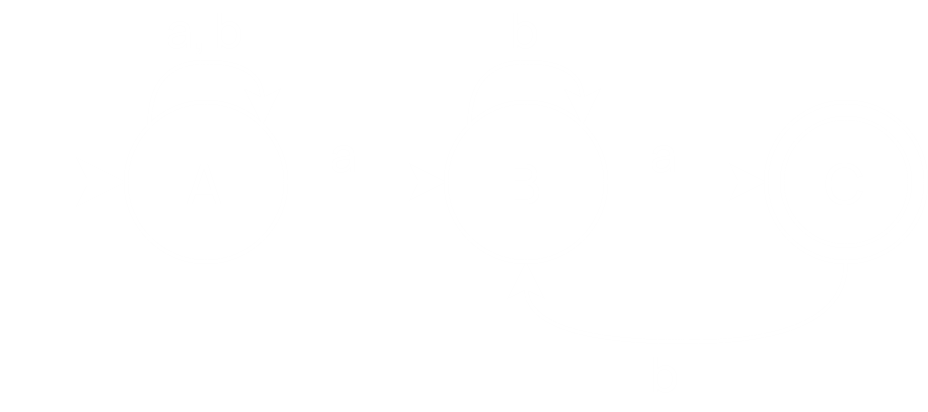
Using these definition, for the same NFA, where ,

:

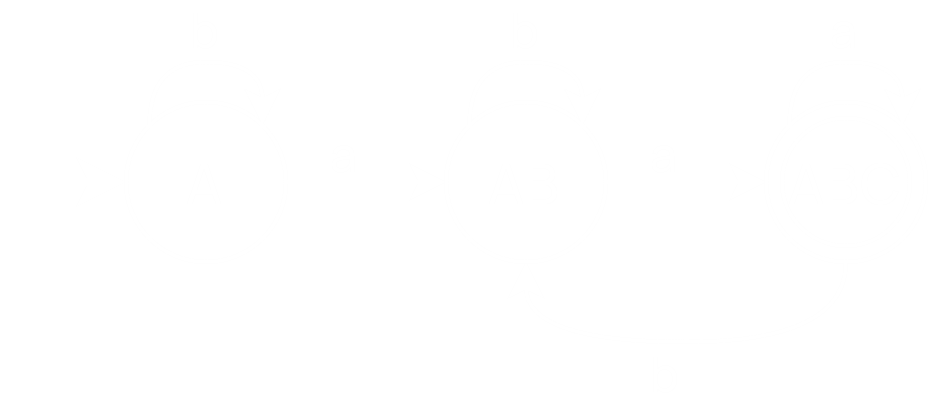
|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| A | AB | A |
| AB | ABC | A |
| ABC | ABC | A |

For the following NFA, create the corresponding DFA.

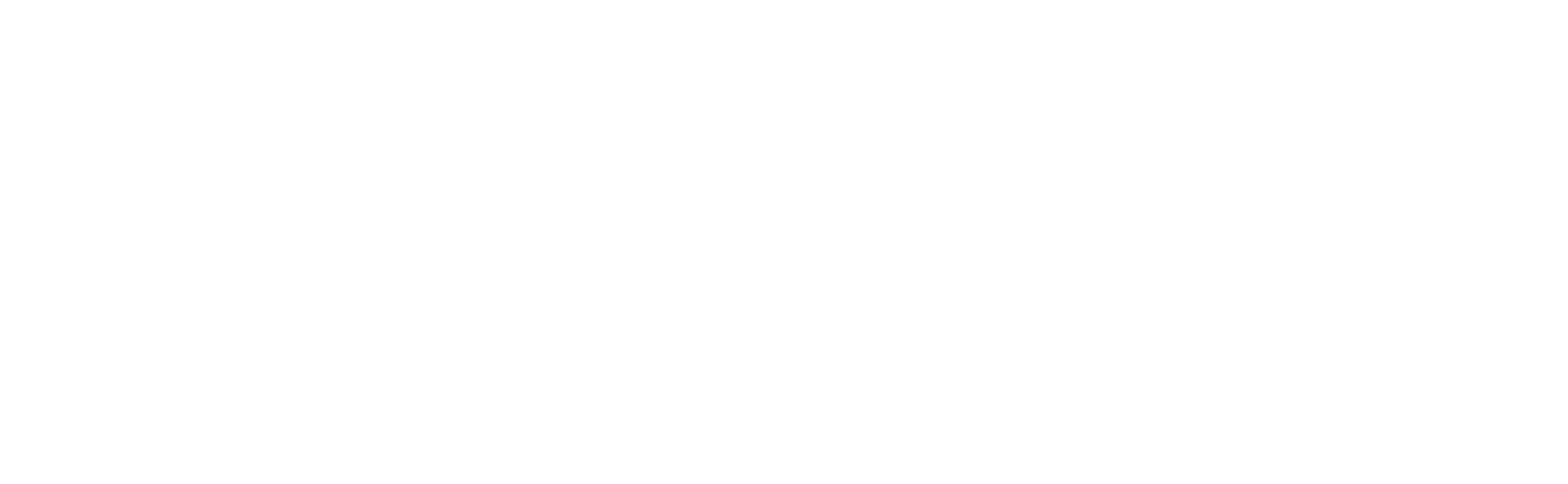
NFA:



DFA:



## Epsilon Values

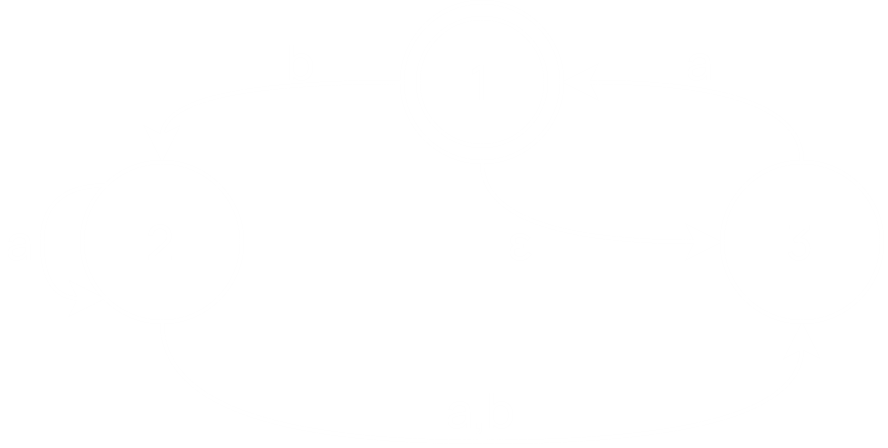


Suppose we are current at the state . The **epsilon closure** of gives us the possible states we can go to from . Formally,

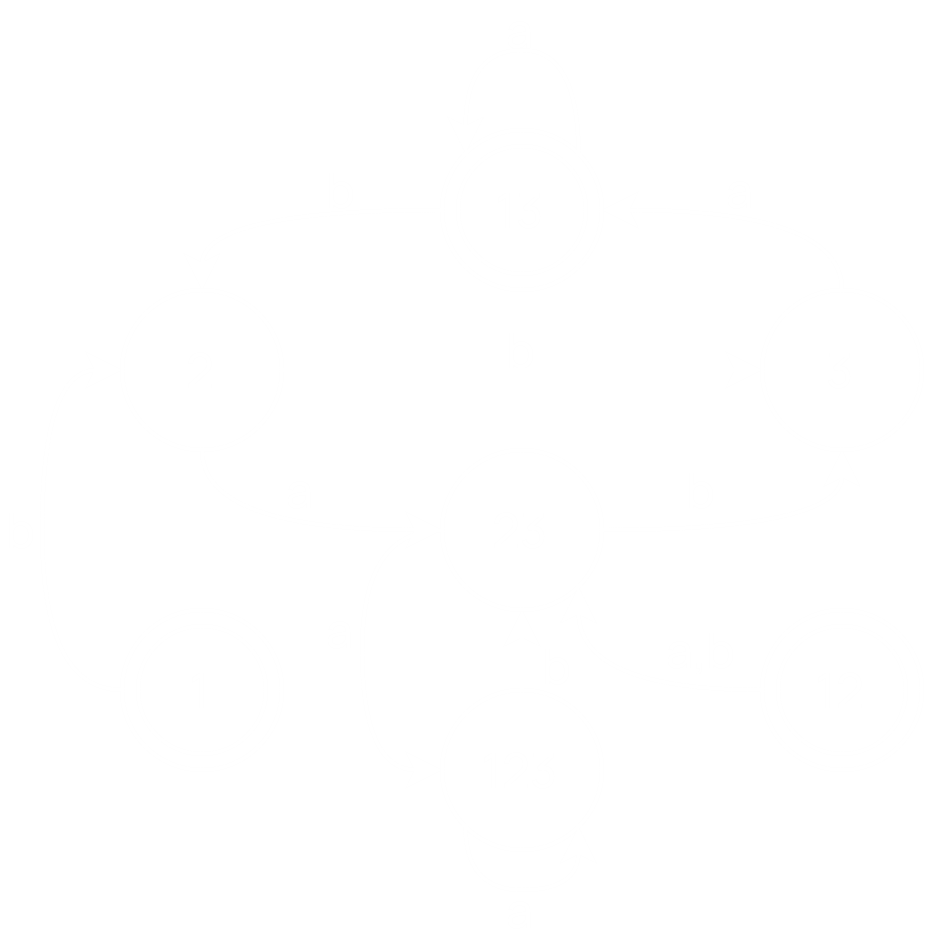
For the diagram above, if , the state reachable via epsilon closure is .

Based on this information, we need to make two changes to how we define the NFA.

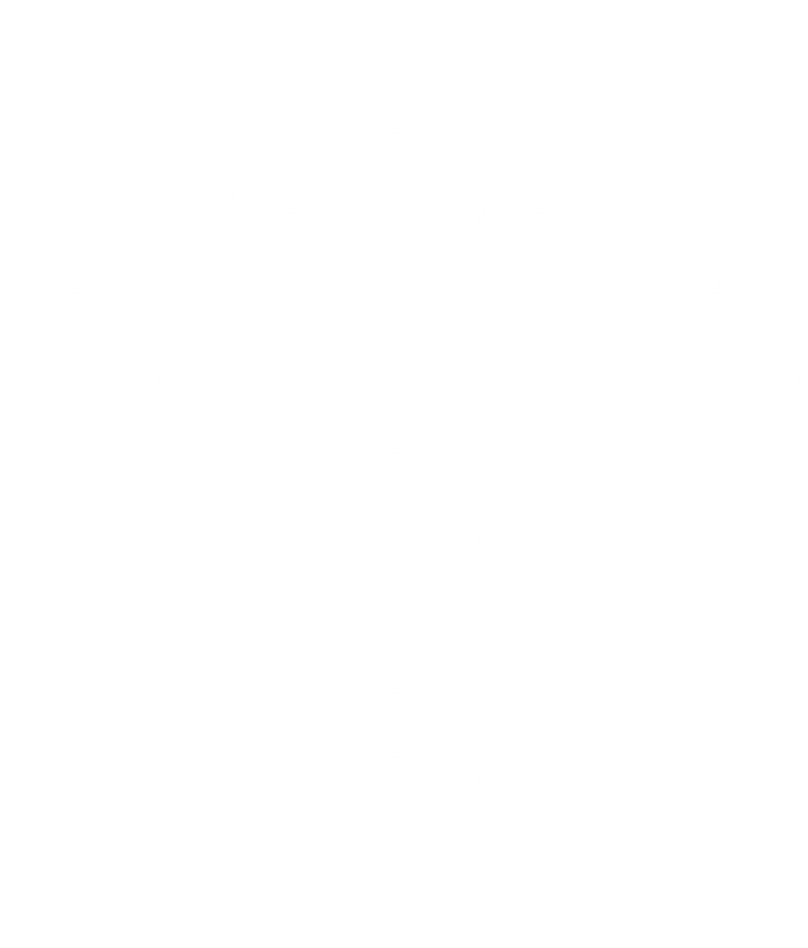
Note that the start state is now all the states that are reachable from via epsilon edges.



For the above NFA,



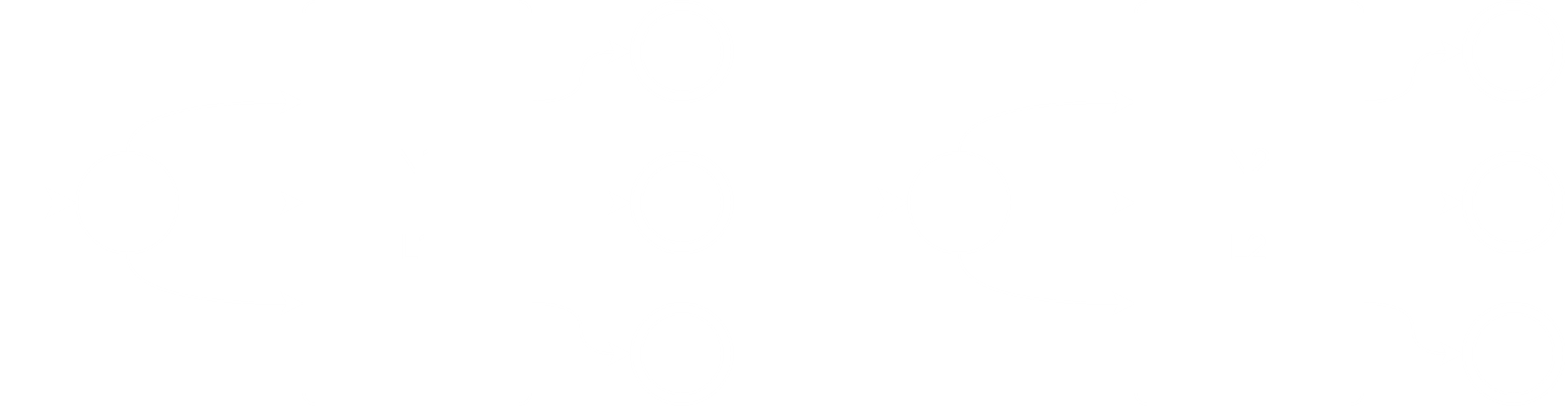
In the above DFA, notice that the states 1 and 12 cannot be reached. Thus, we can discard them.

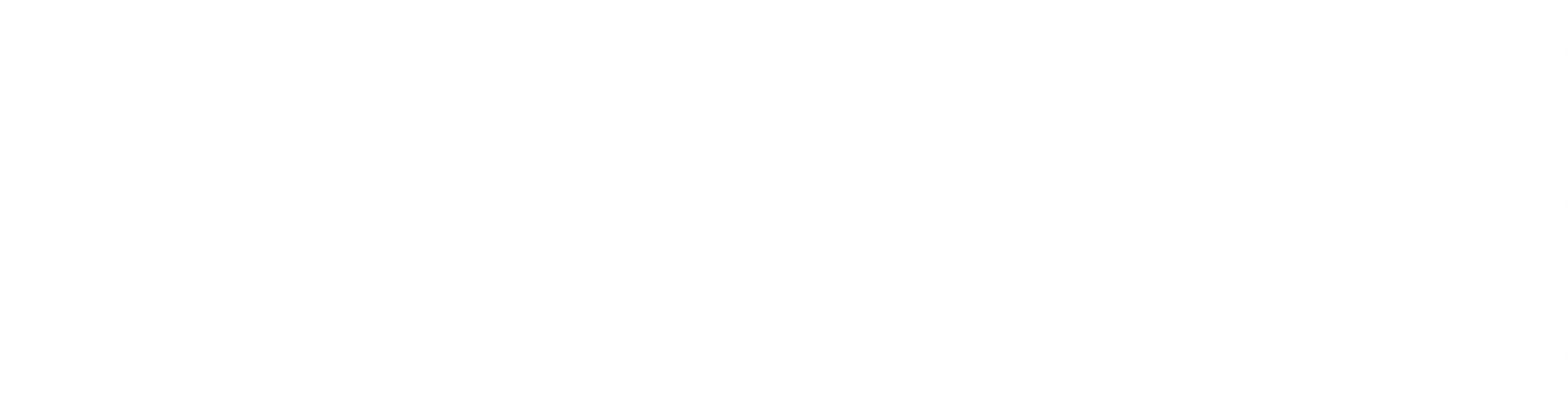


## Concatenating Languages

We previously saw that . For some string , means that and . The issue with this when working with DFAs was we have no way of knowing where ends and begins. This is solved by NFAs.

With an NFA, we can simply connect the final states of some machine which accepts to the start state of some machine which accepts . The connection is made using .



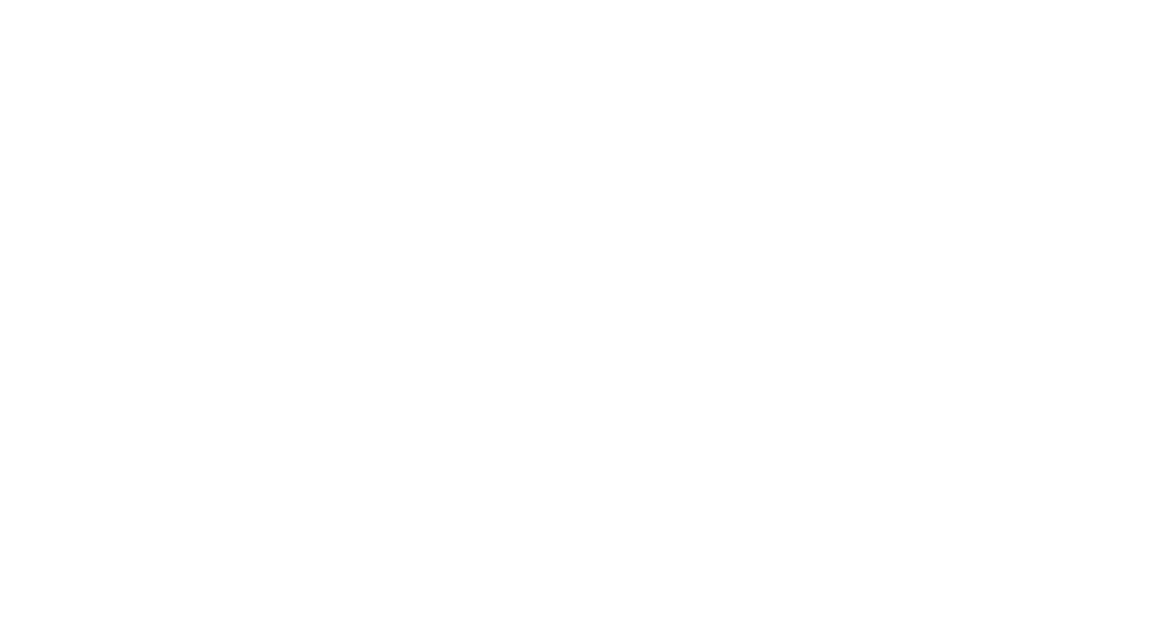


Suppose and . Thus,

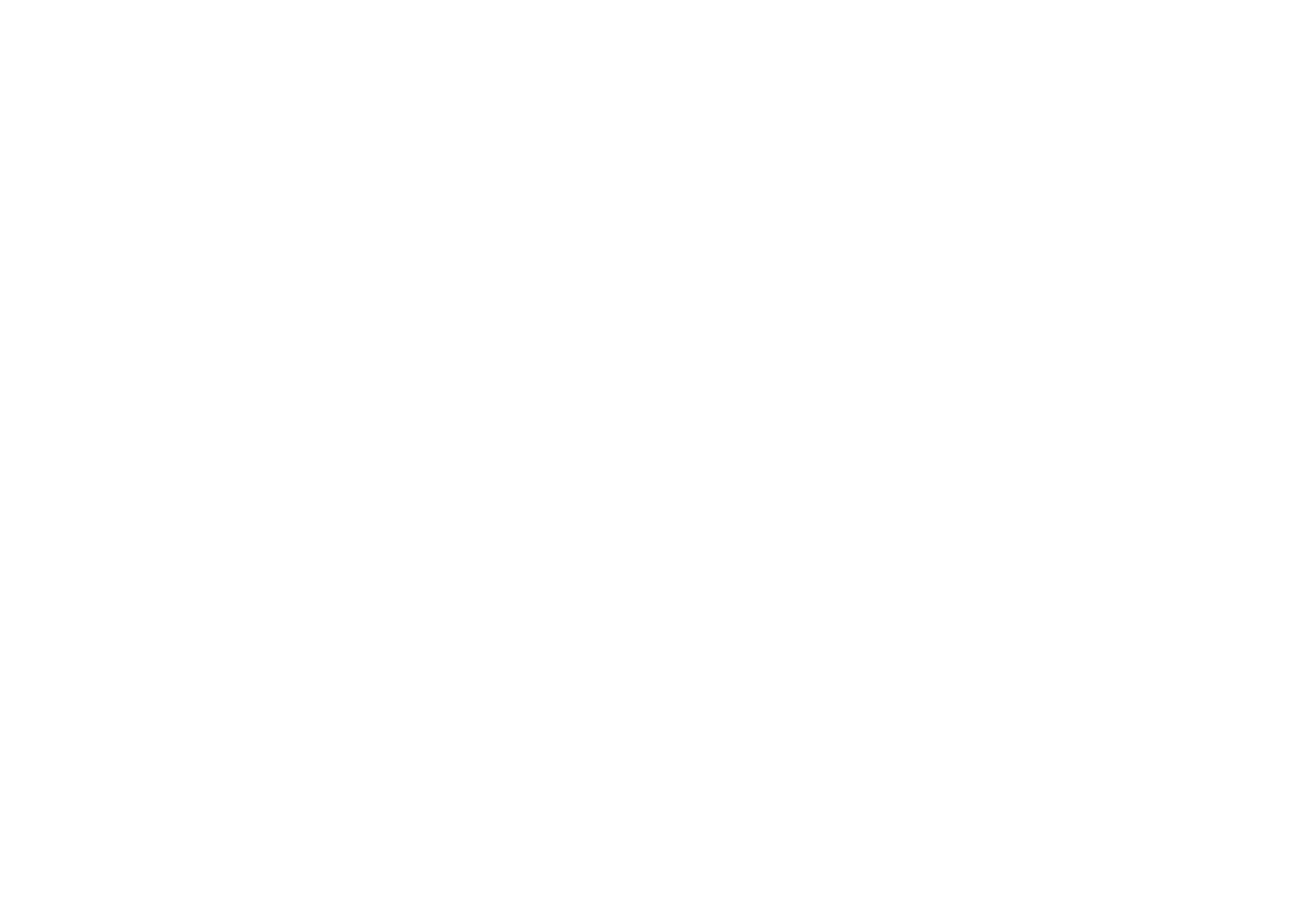
## Kleene Closure

Suppose . Thus, . For such a language, we can use an NFA to build a machine.

Machine for :



Machine for :



Here, we added a new state that is also a final state as the entry point. This captures the empty input. To capture repetitions, we simply joined the final states to the old start state.