WEEK 7: Finite Automata

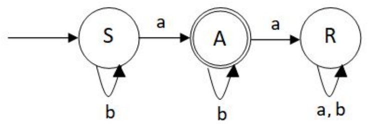
-- General --

The major connection:

→ Finite Automata represent Regular Expressions

→ They are a visual representation of whether an input string is a match / is accepted by a given language

For example, suppose our alphabet is {a, b} and we want a finite automata that accepts strings with exactly one ‘a’



The states are labeled S for start, A for accept, and R for reject just for clarity (they can be named whatever you want).

An equivalent regular expression for this automata would be: **b*ab***

How do we come up with such an automata?

→ We start with one state representing the empty string. Then we ask ourselves, “if I take an ‘a’ transition, how does that progress me toward my goal or possibly set me back?” similarly for all characters in our alphabet for every new state we create

→ So when we first take an ‘a’ transition we have reached our goal of having exactly one ‘a’

→ If I take another ‘a’ transition then I can no longer have a string with exactly one ‘a’ so I will then reject no matter what the rest of my string reads

→ Transitions on ‘b’ don’t progress us toward our goal so each state self loops on a ‘b’

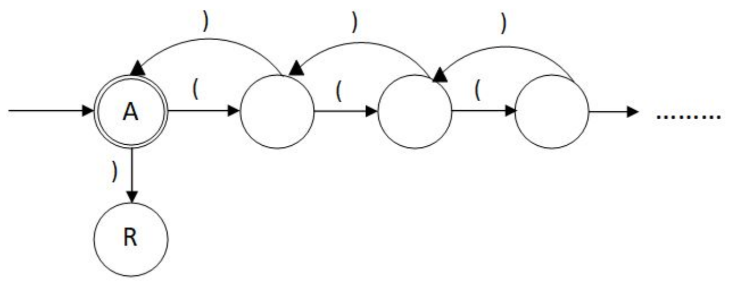
-- So what’s the point? --

(Most of this section is beyond the scope of the course but still very cool to know that it exists!)

Why would we need finite automata if we already have regular expressions? I will illustrate the answer with an example:

Could you write a regular expression for valid parentheses such as “()”, “()()”, “(())”?

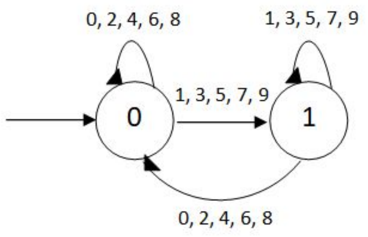
Take a minute to think about it …. If you're not coming up with anything think about this: if there exists a regular expression for the language, then there must also exist a finite automata for the language and vice versa. So let's try to draw one.



Notice how it goes on forever …. *Infinitely* …. A *finite* automata cannot have *infinite* states. Thus, this is somewhat of an informal proof for why the language for valid parentheses is <u>not regular</u> and a regular expression for the language does not exist! Although not regular, we will see how to express such languages later on with CFGs.

Another reason finite automatas are useful is because some regular expressions can be harder to formulate than the finite automata associated with the same language. So if the two are equivalent, we could algorithmically create a regular expression from a finite automata!

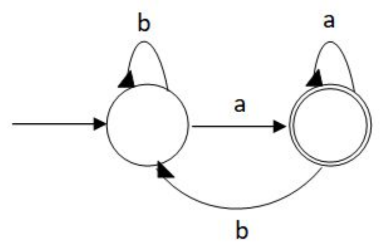
Finite automata are surprisingly powerful and give us deeper insights about the languages we are examining as well as help us write regular expressions. They are also useful in classifying input strings. For example, we could write a finite automata to show us whether an input over the alphabet {0-9} is odd or even:



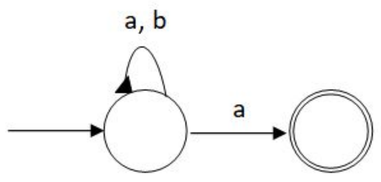
-- NFA vs DFA --
We can define two types of automata.

1) DFA (deterministic finite automata)
   → everything we have seen thus far
   → comprised of 5 attributes M = (Q, Σ, □, s, F)
   - Q is the set of all states
   - Σ is the alphabet of the language
   - □ is the list of transitions (q_1, σ, q_2) where q_1, q_2 ∈ Q and σ ∈ Σ
   - s is the start state
   - F is the set of final states
2) NFA (non-deterministic finite automata)
   → an extension of a DFA (meaning every DFA is also an NFA), where we can have empty transitions and nondeterministic transitions. For example, consider a language over {a, b} where the string must end with an ‘a’

DFA:



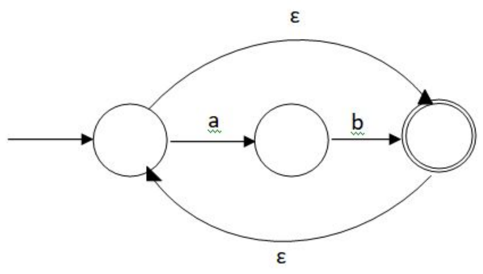
Equivalent NFA:



The NFA in this example accepts the same strings as the DFA. The NFA also has ***non-deterministic*** transitions because on an ‘a’ transition from the start state, we could ***either*** move to the start state ***or*** the end state.

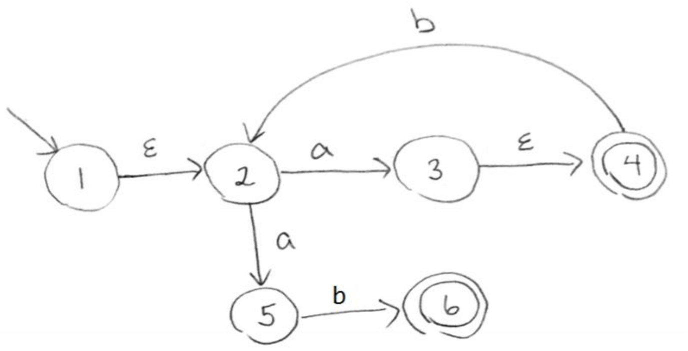
When determining whether or not a string is accepted by an NFA we must look at ***ALL*** the possible states we could end on (as opposed to one) and see if at least one of those states is a final state

An example NFA with empty transitions (epsilon transitions)



This NFA is equivalent to $(ab)^*$

-- NFA to DFA Conversion --
Consider the following NFA from class



Since NFA's and DFA's can be equivalent if they accept the same languages, there must be a way to convert between the two. We will attempt this with the subset construction algorithm.

We start on state 1. Without any character inputs, where are all the possible states we could be positioned on? We could be on state 1 or empty transition to state 2. We then build set (state) {1, 2} and ask ourselves what states can I transition to via an 'a'? Via a 'b'?

- {1, 2}
  $\rightarrow_{(a)}$ = {3, 4, 5}
  $\rightarrow_{(b)}$ = ∅

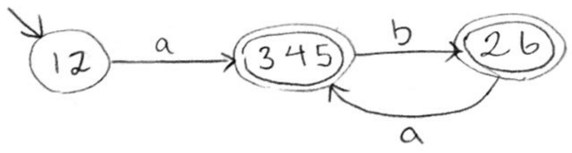
Taking an 'a' from state 1 or 2 can get us to states 3 or 5. Then if we take another empty transition from state 3, we could also be on state 4. We create new set {3, 4, 5}.

Now we repeat the same steps over again for all new sets (states) we create until we can no longer make new sets (states).

- {3, 4, 5}
  $\rightarrow_{(a)} = \varnothing$
  $\rightarrow_{(b)} = \{2, 6\}$

- {2, 6}
  $\rightarrow_{(a)} = \{3, 4, 5\}$
  $\rightarrow_{(b)} = \varnothing$

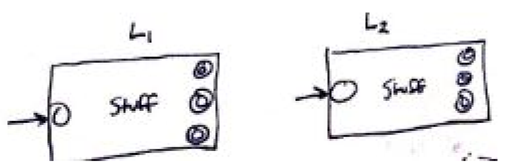
Then using the sets we created as our DFA states and drawing the transitions we noted above, we get an equivalent DFA.



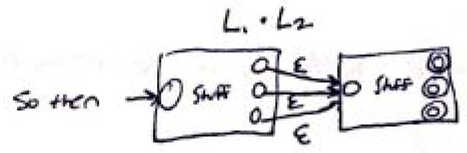
The final states of the DFA are states {3, 4, 5} and {2, 6} because they contains at least one final state from the original NFA (4 and 6).

-- Regular Expressions to NFAs --
Lastly, we can convert from regular expressions to equivalent NFAs. If we think about regular expressions as compositions of languages using concatenation, union, and kleene star, the steps become simple.

If we have regular expression L that can be broken into the ***concatenation*** of two smaller languages $L_1 \bullet L_2$, (for example splitting language L = ab into L_1 = a and L_2 = b) we could represent L_1 and L_2 as NFAs

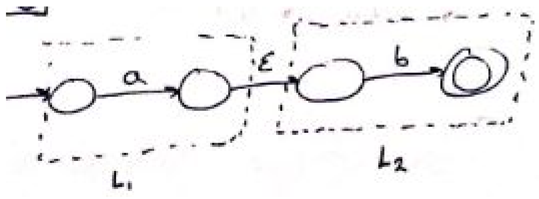


and concatenate them via epsilon transitions.

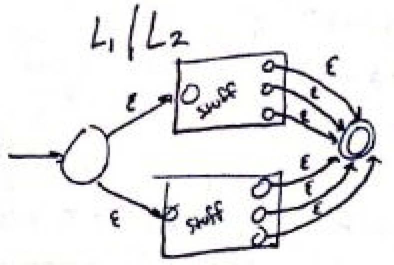


We make the final states from L_1 regular states and make empty transitions from the former L_1 final states to the start state of L_2, leaving the final states of L_2 as our final states of L.

For the example langage ab, this rule would give us

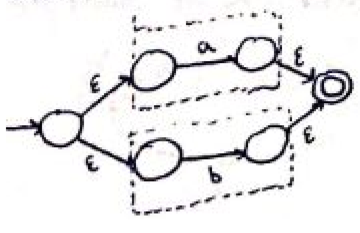


We could do something similar for the ***union*** of languages L = L_1 | L_2

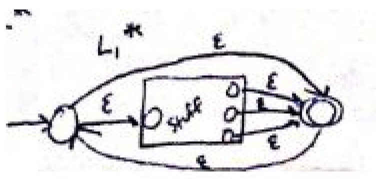


Here we add two new states. One becomes the new start state and empty transitions to the start states of L_1 and L_2. The other new state becomes the final state of L and the old final states of L_1 and L_2 become regular states.

For the example language a | b, this rule would give us

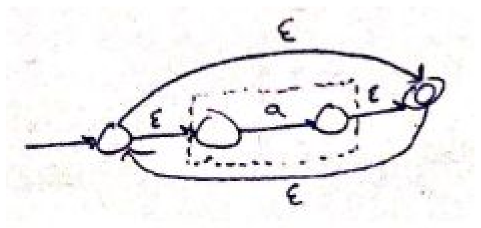


We can also do something similar with the kleene star, $L = L^*$



Here we again add two new states. One becomes the new start state and we make an empty transition to the old start state. The other new state becomes the final state and we make empty transitions from the old final states to the new one. Lastly, since we want to accept any number of occurrences of the language, we make epsilon transitions from the start to end state and from the end to start state.

For the example language a^*, this rule would give us



-- Challenge --

1) Earlier we saw a DFA for the set of strings that end with an 'a'.
   a) Write a DFA for a set of strings where 'a' is the second to last letter
   b) 3rd to last?
   c) Think about how many states would your DFA have for the set of strings where 'a' is the nth to last element

2) Earlier we saw a DFA that showed us whether a number was divisible by 2. Over that same alphabet, could you find a DFA that accepts strings that are divisible by 3? (Hint: it may help to remember the divisibility trick where if the sum of the digits is divisible by 3, then the original number is also divisible)

3) Write a DFA for the following languages over the alphabet {a, b}. The notation $\#_a$ means "the number of a's."
   a) The set of strings where $\#_b < 4$
   b) The set of strings where $\#_a > 2$ and $\#_b < 4$

c) If we included the character c in our alphabet and added the constraint where $\#_c = 3$, think about how many states the DFA must have. If we add more characters and constraints, is there a rule for knowing how many states there will be?

d) The set of strings where $\#_a$ is odd and $\#_b$ is even.

e) The set of strings where $\#_a \equiv 2 \pmod 3$ and $\#_b \equiv 1 \pmod 3$

f) Again, if we add character c in our alphabet and added the constraint where $\#_c \equiv 0 \pmod 2$, how many states would we have?

4) What would an NFA look like for the regular expressions L = a? or L = a+ as opposed to our traditional concatenation, union, and star operations?