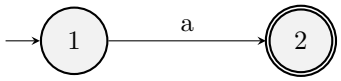
Theory of Computation: Assignment 4 Solutions

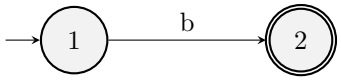
Arjun Chandrasekhar

1.
   - $L = \{w | w$ contains the substring 0101$\}$
     $R = \Sigma^* 0101 \Sigma^*$
   - $L = \{w | w$ has length at least 3 and its third symbol is 0$\}$
     $R = \Sigma\Sigma 1 \Sigma^*$

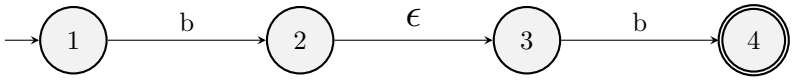
2. To make an NFA for $a(abb)^* \cup b$, first let's make an NFA for a



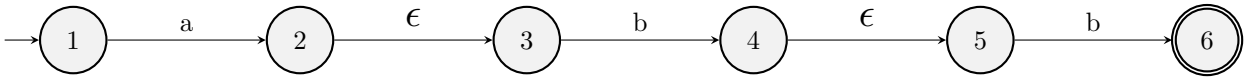
   Next, let's make an NFA for b



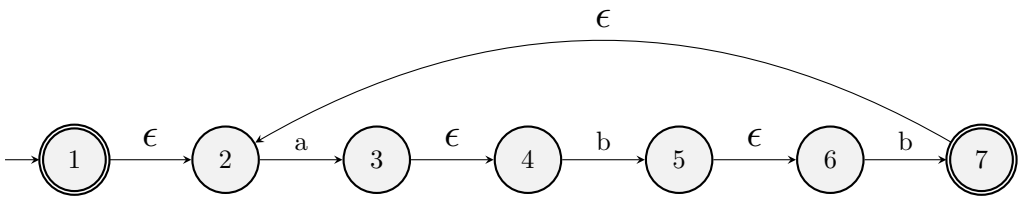
   Next, let's make an NFA for $b \circ b$



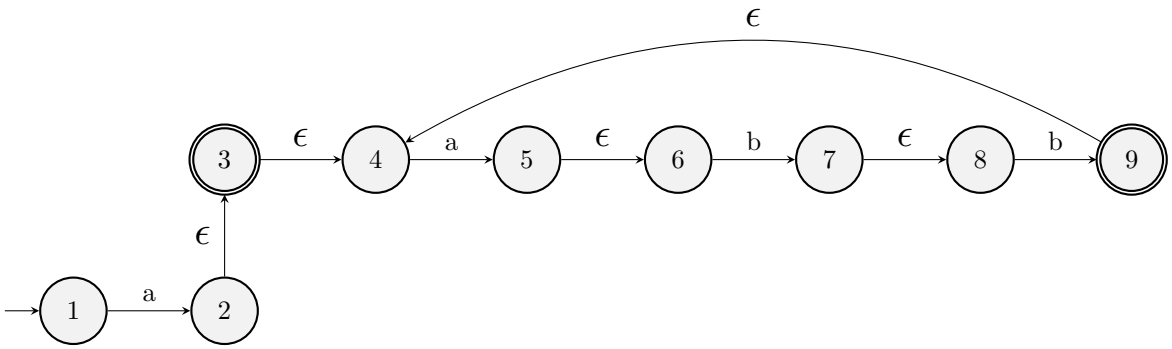
   Next, let's make an NFA for $a \circ b \circ b$



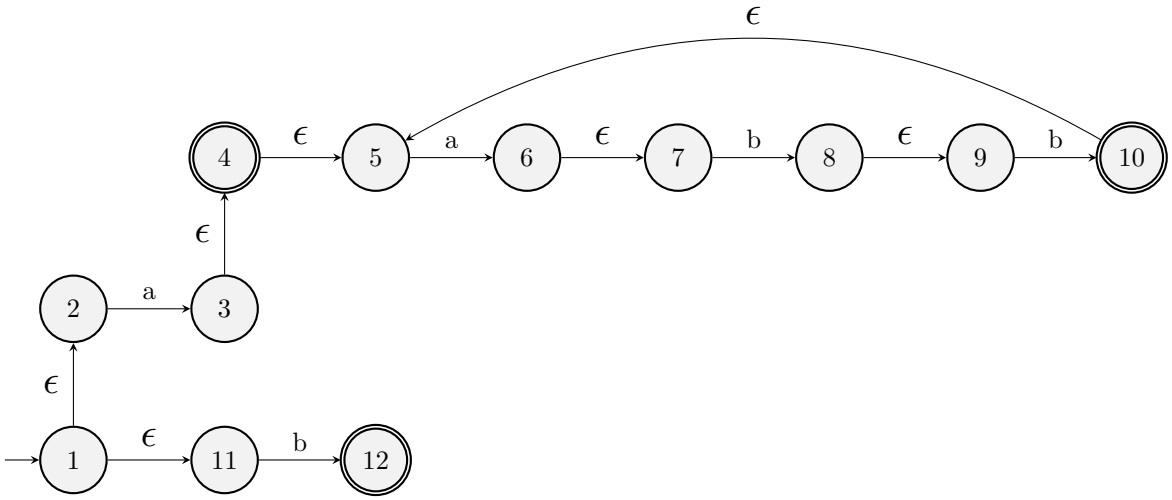
   Next, let's make an NFA for $(a \circ b \circ b)^*$



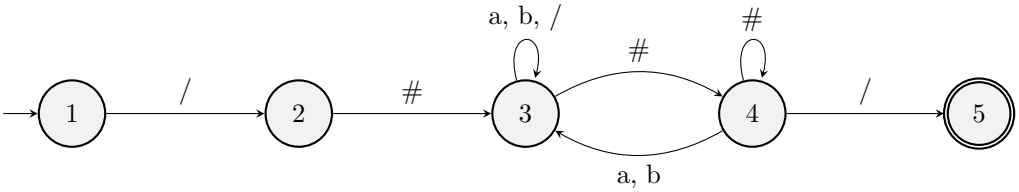
   Next, let's make an NFA for $(a \circ (a \circ b \circ b)^*$



Finally, let's make an NFA for $(a \circ (a \circ b \circ b)^*) \cup b$



3. The following 5-state NFA recognizes C

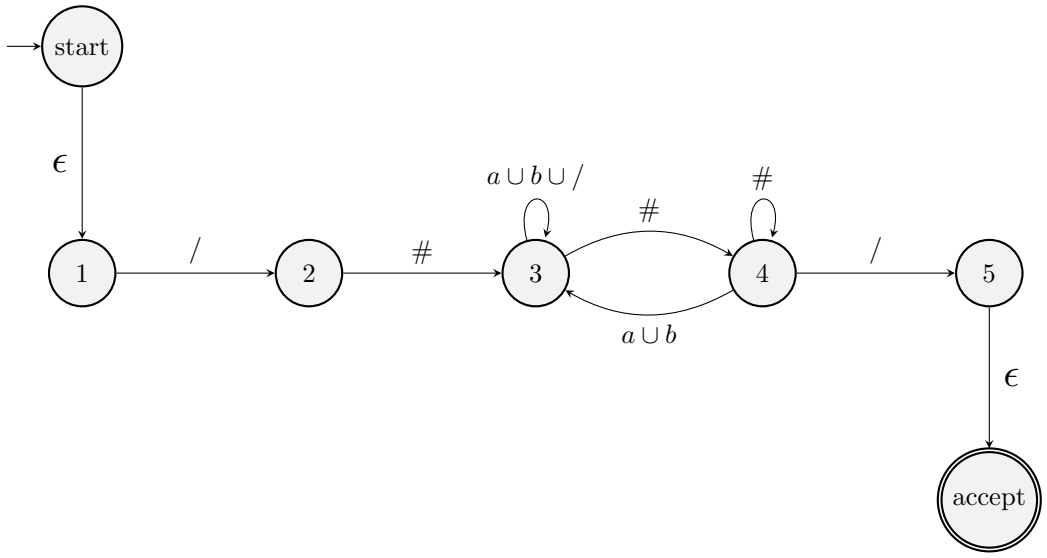


Next we'll convert our NFA into a proper GNFA. This entails the following:

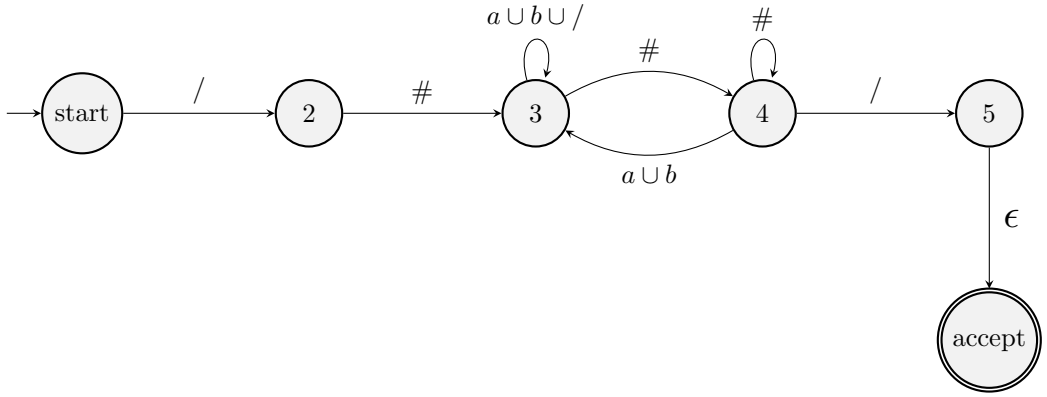
- We need to add a special start state, and an ϵ transitions to the original start state
- We need to add a special accept state, and add ϵ transitions to it from all of the original accept states (or in this case, singular accept state).
- We need to add transitions between every pair of original states (including self-loops), and label each transition with a regex.
  - If there is just one σ between a pair of of states, then our regex for that transition σ
  - If there are multiple transitions $\sigma_1, \sigma_2, \ldots \sigma_n$ between a pair of states, then our regex for that transition is $\sigma_1 \cup \sigma_2 \cdots \cup \sigma_n$

– Technically, if any transition is missing, we are supposed to add an $\emptyset$ transition. However, we will simply omit these transitions to make the state diagram easier to read.

The starting GNFA is as follows:

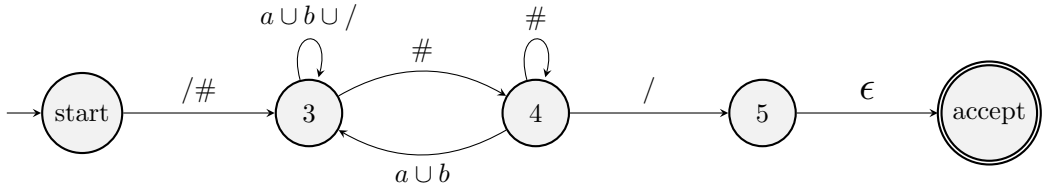


After we rip state 1, the GNFA is as follows:



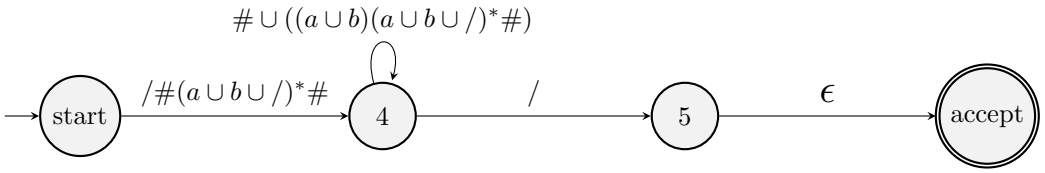
Note that the start $\rightarrow$ 2 transition is technically $\epsilon \circ (\emptyset)^* \circ / = /$

After we rip state 2, the GNFA is as follows:



Note the start $\rightarrow$ 3 transition is technically $/ \circ (\emptyset)^* \circ \# = /\#$

After we rip state 3, the GNFA is as follows



Let's walk through some of the transitions we repaired:

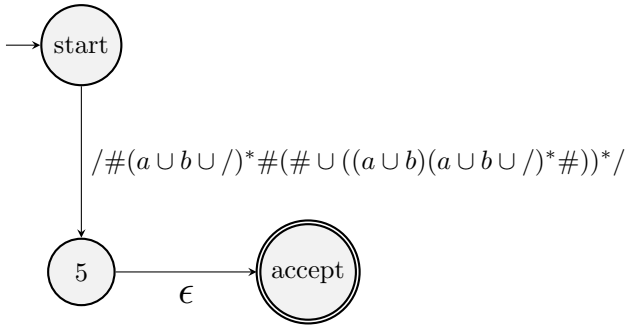
- **start → 4:** There were no direct transitions from *start* to 4 - i.e., the (invisible) transition is $\emptyset$. The other way to get from start to 4, via 3 was as follows: 1. start to 3 (#) 2. 3 back to itself any number of times $(a \cup b \cup /)^*$ and 3. From 3 to 4 (#). Put it all together, and you get

$$\emptyset \cup ((/\#) \circ (a \cup b \cup /)^* \circ (\#)) = /\#(a \cup b \cup /)^*\#$$

- **4 → 4:** There was a direct transition from 4 back to itself (#). The other way to get from 4 back to itself, via 3, was as follows: 1. Go from 4 to 3 $(a \cup b)$ 2. Go from 3 back to itself any number of times $(a \cup b \cup /)^*$ and 3. Go from 3 back to 4 (#). Put it all together, and you get

$$\# \cup ((a \cup b) \circ (a \cup b \cup /)^* \circ (\#)) = \# \cup ((a \cup b)(a \cup b \cup /)^*\#)$$

After we rip state 4, the GNFA is as follows:

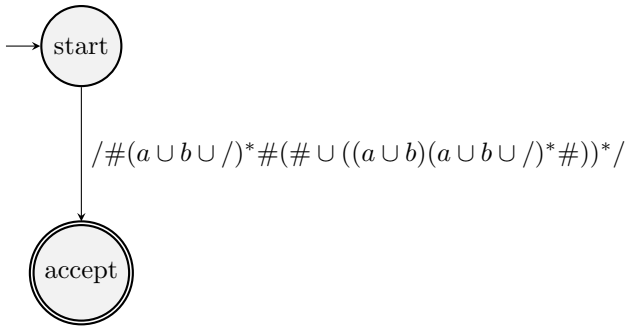


Let's review how we repaired the start → 5 transition:

- There is no direct transition from start to 5; in other words, its an $\emptyset$ transsition.
- The other way to get from start to 5 is via 4
  - The transition from start to 4 is $/\#(a \cup b \cup /)^*\#$
  - The transition that lets us loop from 4 back to itself any number of times is $(\# \cup ((a \cup b)(a \cup b \cup /)^*\#))^*$
  - The transition from 4 to 5 is $/$
- Put it all together and you get

$$\emptyset \cup (/\#(a \cup b \cup /)^*\# \circ (\# \cup ((a \cup b)(a \cup b \cup /)^*\#))^* \circ /) = /\#(a \cup b \cup /)^*\#(\# \cup ((a \cup b)(a \cup b \cup /)^*\#))^*/$$

Finally, we rip state 5, which give the following GNFA:



Note that the transition from start to accept is technically $\emptyset \cup (/\#(a \cup b \cup /)^*\#(\# \cup ((a \cup b)(a \cup b \cup /)^*\#))^*/ \circ \emptyset^* \circ \epsilon)$

Thus, our final regular expression is

$$\boxed{/\#(a \cup b \cup /)^*\#(\# \cup ((a \cup b)(a \cup b \cup /)^*\#))^*/}$$

4. **Approach 1: construct an NFA**. Let's first think about the relationship between L and L^R. Suppose $w \in L$. Then when we run w through the DFA for L, it will transition make its way from the start state to one of the accept states. This means that if we were to read the string in reverse, we would be able to go from one of the accept states back to the start state.

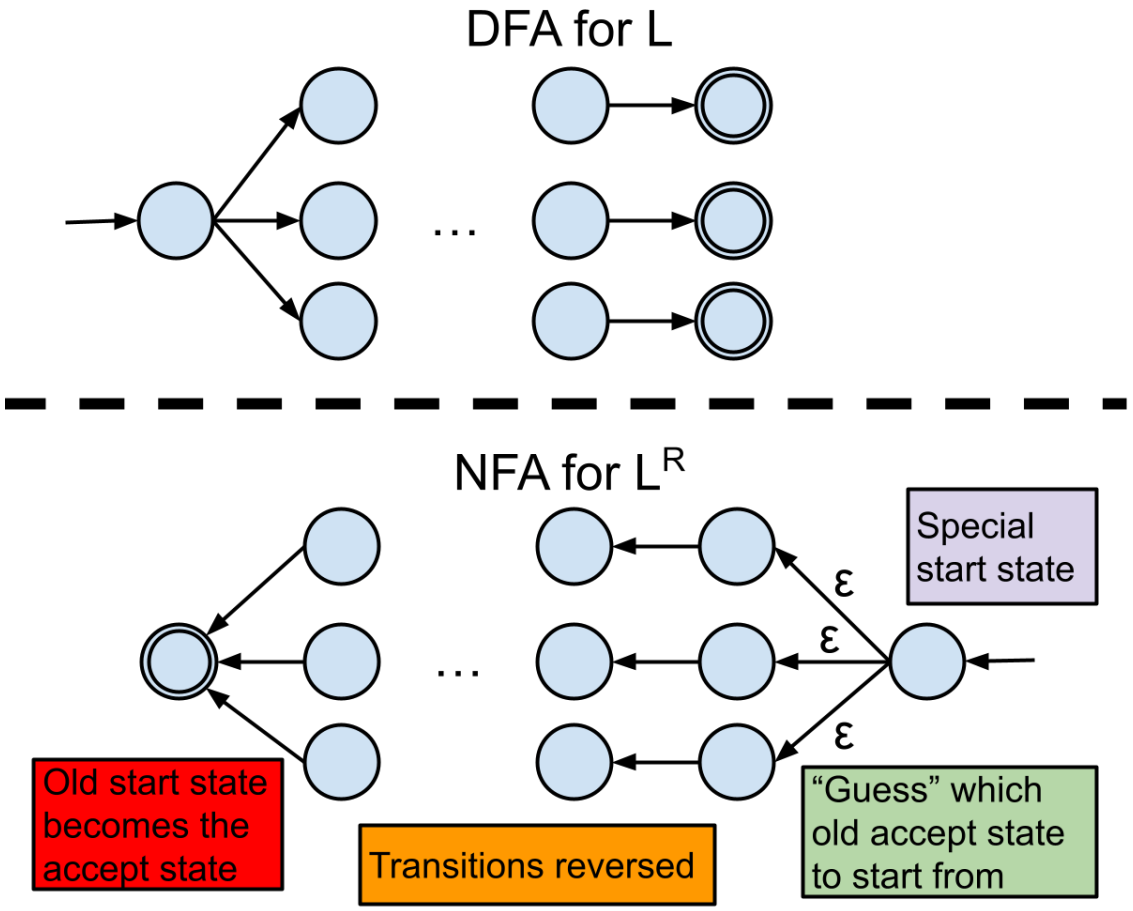
   We will take the original DFA for L, and convert it into an NFA for L^R. We'll make it so that the NFA accepts strings that are able to get from an accept state to the start state; this will imply that when the string is reversed, it would be accepted by the original DFA. We will make the following three modifications to the DFA:

   (a) The original start state becomes the only accept state.

   (b) All of the transitions get reversed.

   (c) We will add a special start state, with ϵ transitions to each start state. This lets the machine "guess" which accept state the string would have ended up in if it weren't reversed.

   Here is a diagram of the proof idea



Formally, suppose L is regular. Then it is is recognized by a DFA $M = (Q, \Sigma, \delta, S, F)$. We'll design an NFA $M^R = (Q^R, \Sigma, \delta^R, S^R, F^R)$ to recognize L^R as follows:

- $Q^R = Q \cup \{S'\}$ - we have all of the original states from M, along with a special new state S'
- $\Sigma = \Sigma$ - same alphabet
- If $\delta(q_i, \sigma) = q_j$, then $q_i \in \delta^R(q_i, \sigma)$ - the transitions in the NFA are the reverse of the transitions in the original machine.
  Additionally, $\delta(S', \epsilon) = F$ - the new state S' has ϵ transitions to all of the original accept states
- $S^R = S'$ our new machine starts in the special state that we added
- $F^R = \{S\}$ - the original start state becomes our only accept state

Approach 2: construct a regular expression. Suppose L is regular. Then L is described by some regular expression R. We will prove by induction that there exists a regular expression R' for L^R.

Base Case: Suppose the size of R is 1.

- **Case 1:** $R = a \in \Sigma$. Then $R' = a$ describes L^R
- **Case 2:** $R = \epsilon$. Then $R' = \epsilon$ describes L^R
- **Case 3:** $R = \emptyset$. Then $R' = \emptyset$ describes L^R

Inductive Case: Suppose all regular expressions of size $\leq n$ can be reversed. More formally, suppose for all regular expressions R of size $\leq n$, there exists a regular expression R' for the language described by R. Now let R be a regular expression of size $n+1$ that describes L.

- **Case 1:** $R = R_1 \cup R_2$. Note that R_1, R_2 have sizes $\leq n$. By our inductive hypothesis, there exist regular expressions R_1', R_2' for the reversals of $L(R_1), L(R_2)$. Then $R_1' \cup R_2'$ describes L^R
- **Case 2:** $R = R_1 \circ R_2$. Note that R_1, R_2 have sizes $\leq n$. By our inductive hypothesis, there exist regular expressions R_1', R_2' for the reversals of $L(R_1), L(R_2)$. Then $R_2' \circ R_1'$ describes L^R. Note that we reversed the order of concatenation.
- **Case 3:** $R = R_1^*$. Note that R_1 has size $\leq n$. By our inductive hypothesis, there exists a regular expression R_1' for the reversal of $L(R_1)$. Then $(R_1')^*$ describes L^R

5. **Approach 1: construct an NFA**. Let's think about the relationship between L and DROP-OUT(L). Suppose we take a string $s \in$ DROP-OUT(L). This means that s is missing one character that would allow it to be in L. If we were to run s through the DFA for L, we would reach a point where one character is missing, and we wouldn't be able to make one of the transitions we need to eventually accept.

   Alternately, consider $w \in L$. Imagine if we ran w through the DFA for L, but "skipped" one character - that is, we made transition for that character without actually reading the character. The letter we *did* read would form a string $s \in$ DROP-OUT(L).

   So how do we recognize if a string $s \in$ DROP-OUT(L)? We run s through the original DFA. However, we give s one chance to "skip" or "simulate" reading the and transitioning on the one missing symbol. In other words, we use an ϵ transition to move from one state to the next, even if the character that we needed for this transition is not present. This way, as long as all but the one dropped-out symbol is present, the string is accepted.

   A naive approach would be to augment the original DFA with ϵ transitions that allow the machine to simulate a missing transition. Unfortunately, putting all these ϵ transitions into the same NFA would allow it to simulate reading several missing characters; we only want to be able to simulate a single missing symbol.

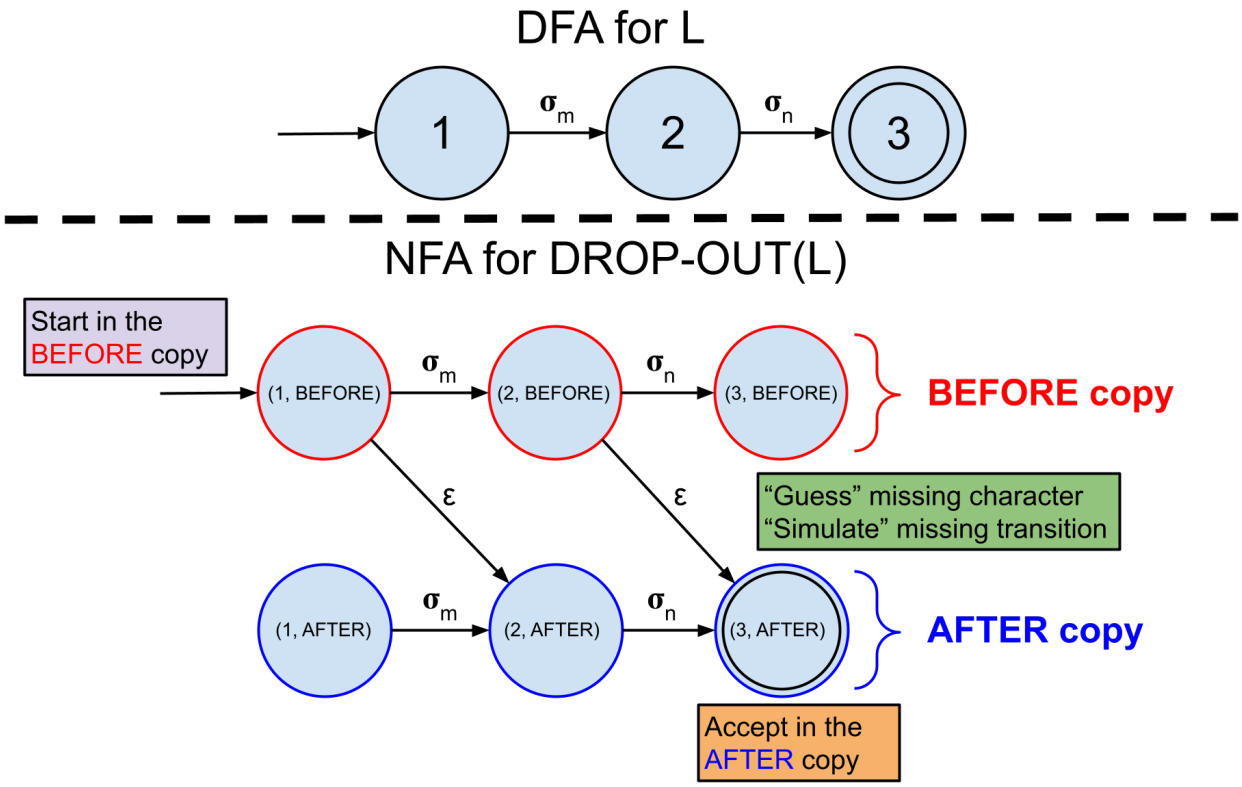
   Instead, we will create two copies of the original DFA: the BEFORE and AFTER copies. The machine will proceed as follows:

   (a) Use the BEFORE copy to read all the symbols that come before the missing symbol

   (b) Guess where and what the missing symbol is. At this point, use an ϵ transition to move into the AFTER copy without consuming a character; this is what we mean by "simulating" reading the missing symbol.

   (c) Use the AFTER copy to read all the symbols that come after the missing symbol. After all this, check if we reached an accept state.

   Here is a diagram of this proof idea



Formally, suppose L is regular. Then there is a DFA $M = (Q, \Sigma, \delta, S, F)$ that recognizes L. We'll construct an NFA $(Q_D, \Sigma, \delta_D, S_D, F_D)$ as follows:

- $Q_D = Q \times \{\text{BEFORE}, \text{AFTER}\}$ - we make two copies of each state; a BEFORE copy, and an AFTER copy
- $\Sigma = \Sigma$ - alphabet is the same
- Suppose $\delta(q_i, \sigma) = q_j$. Then we add the following three transitions:
  - $(q_j, \text{BEFORE}) \in \delta_D((q_i, \text{BEFORE}), \sigma)$ - we preserve this transition in the BEFORE copy
  - $(q_j, \text{AFTER}) \in \delta_D((q_i, \text{AFTER}), \sigma)$ - we preserve this transition in the AFTER copy
  - $(q_j, \text{AFTER}) \in \delta_D((q_i, \text{BEFORE}), \epsilon)$ - we allow the NFA to simulate this transition without consuming a symbol. But once we move from the BEFORE copy to the AFTER copy, we can't guess any more dropped out symbols.
- $S_D = (S, \text{BEFORE})$ we start in the BEFORE copy of the start state
- $F_D = F \times \{\text{AFTER}\}$ - we accept if we are in one of the AFTER copies of the accept states

Approach 2: construct a regular expression. Suppose L is regular, then there exists a regular expression R which describes A. We will create a regular expression R' to describe DROP-OUT(L).

Base Case: Suppose the size of R is 1.

- **Case 1:** $R = a$. Then $R' = \epsilon$ describes DROP-OUT(A)
- **Case 2:** $R = \epsilon$. Then $R' = \emptyset$ describes DROP-OUT(A).
- **Case 3:** $R' = \emptyset$. Then $R' = \emptyset$ describes DROP-OUT(A)

Inductive Case: Assume for all regular expressions R of size $\leq n$, there exists a regular expression R' that describes DROP-OUT($L(R)$). Now let R be a regular expression of size $n+1$ that describes A.

- **Case 1:** $R = R_1 \cup R_2$. Note that R_1 and R_2 have size $\leq n$. By our inductive hypothesis there exist regular expressions R_1', R_2' that describe DROP-OUT($L(R_1)$), DROP-OUT($L(R_2)$), respectively. Then $R_1' \cup R_2'$ describes DROP-OUT(A). Essentially, we drop out a character from either one of the two smaller languages that comprise the union.
- **Case 2:** $R = R_1 \circ R_2$. Note that R_1 and R_2 have size $\leq n$. By our inductive hypothesis there exist regular expressions R_1', R_2' that describe DROP-OUT($L(R_1)$), DROP-OUT($L(R_2)$), respectively. Then $(R_1' \circ R_2) \cup (R_1 \circ R_2')$ describes DROP-OUT(A). Essentially, we drop out a character from the first part of the concatenation and leave the second part unchanged, or we drop out from the second part and leave the first part unchanged.
- **Case 3:** $R = R_1^*$. Note that R_1has size $\leq n$. By our inductive hypothesis there exists a regular expressions R_1' that describe DROP-OUT($L(R_1)$). Then $R_1^* R_1' R_1^*$ describes DROP-OUT(A). Essentially, we still can have any number of copies of R_1, but exactly one of the copies needs to drop a character.