**CS 208 : Automata Theory and Logic** $19^{th} February, 2024$

# Lecture - 21

Topic: Equivalence between RegEx and NFA/DFA

*Scribed by*: Yash Jonjale (22b0990)
*Checked and compiled by*:



## 1 Some Recap

### 1.1 DFAs, NFAs, and $\epsilon$-NFAs

We have discussed that in terms of their expressive power(The set of languages that can be expressed by the machine), **DFAs, NFAs and $\epsilon - NFAs$ are all equivalent** to each other. And the languages represented by such machines are called **Regular Language**s.

### 1.2 Kleene Closure

In short, it is nothing but closure under the **concatenation** operator. For a set $S$, Kleene closure of $S$ **is given by $S^*$.**

### 1.3 RegEx

RegEx is a syntax whose semantics represent a **language**. For example, $(a+b)^*$ (syntax) represents a set of strings(language). Now, it is a natural question if RegEx represents Regular Languages. Well, RegEx is Regular Expressions, so you can guess!

## 2 Expressing RegEx as NFAs

We will cover only the intuition and not the full formal proof. Let's start with an example!

$$\Sigma = \{0, 1\}$$
$$L = ((0.1)^* + (1.0)^*)^* \cdot (1.1 + 0.0)^*$$

Now, you can make a **parsed tree** of this regular expression as they are defined recursively, so the parsed is nothing but the **recursion tree**, which is binary nature due to the operators being binary.

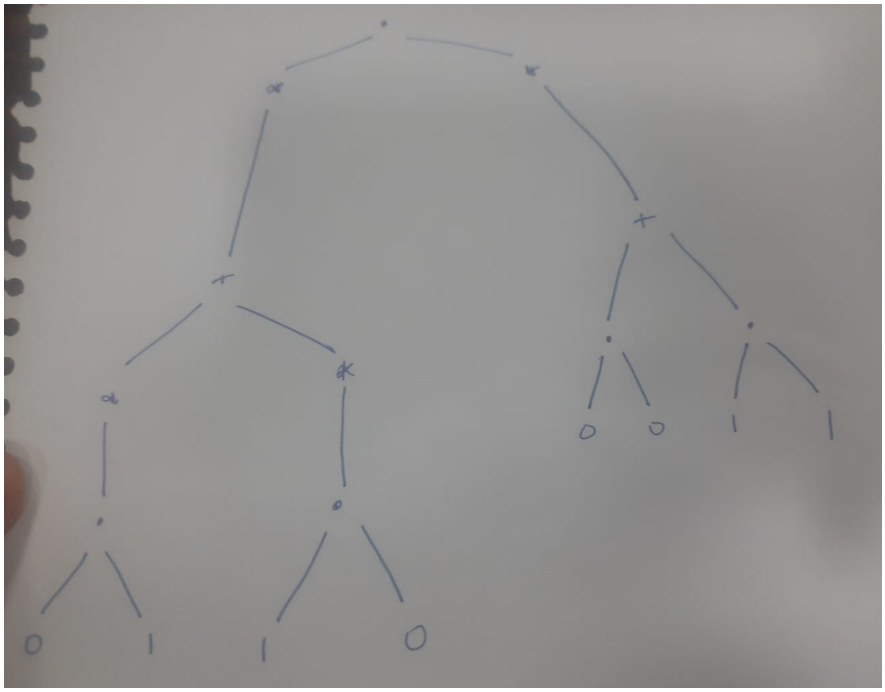


Figure 1: Parsed Tree

Now, we will apply induction on the height of this parsed tree(**structural induction**), assuming the NFA exists for the children node and applying a **combination** operation on the two children NFAs to form the parent NFA.

For this example, we will become from the leaf nodes, moving up to the root. The leaves (height = 0) are either the string "0" or "1". So, we will be first constructing NFAs for them.

**NFA for "0":**

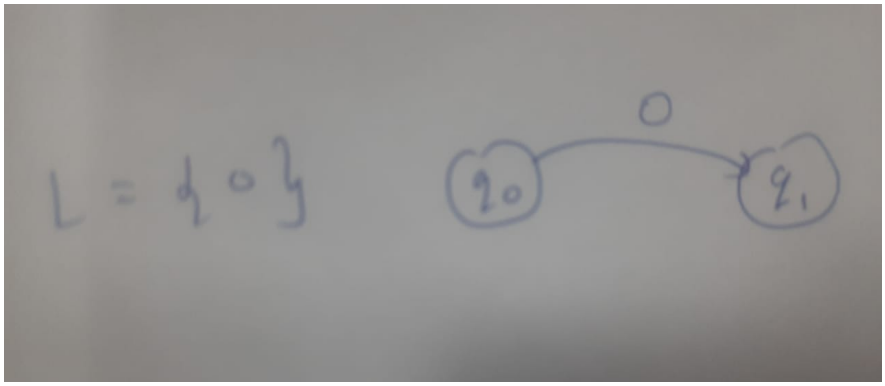


Figure 2: A simple figure

**NFA for "1":**

So, we now have NFAs for all leaves. Now, we will construct NFAs for the level with height = 1.

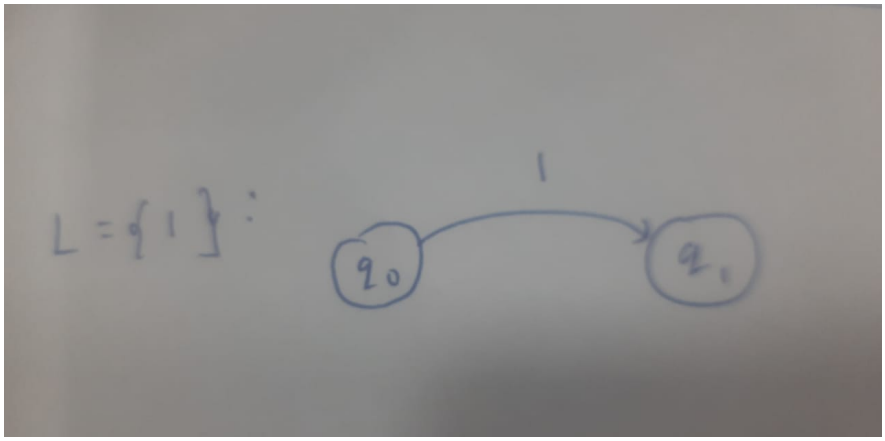


Figure 3: A simple figure

We just have the CONCAT(left.right) at height=1.

To construct an NFA for a **CONCAT(left.right) node**, you will have to check for the left string[1], and immediately after that, you will check for the right string, to do this you will just create an $\epsilon$-edges from the final states of the left string NFA to the starting state of the right string NFA.

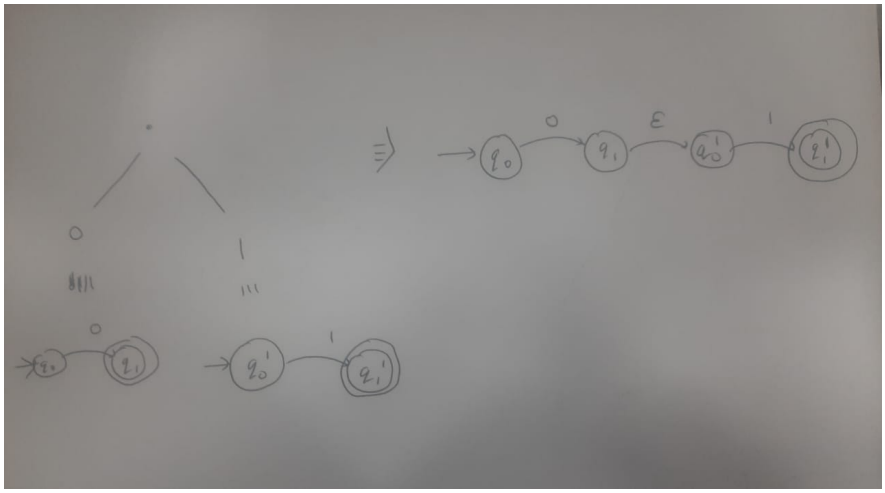


Figure 4:

Let's jump to height = 2, we find a (*) node and a (+) node. How would you combine the children NFAs of these nodes to form the NFA for these nodes?

**For a (*) node:** We have to accept a looped concatenation of the strings from the same language. Recollect the jump instruction in MIPS! So, we just create an $\epsilon$-edge from all the final states of the child NFA to its starting state.

[1]Meaning string belonging to the left language

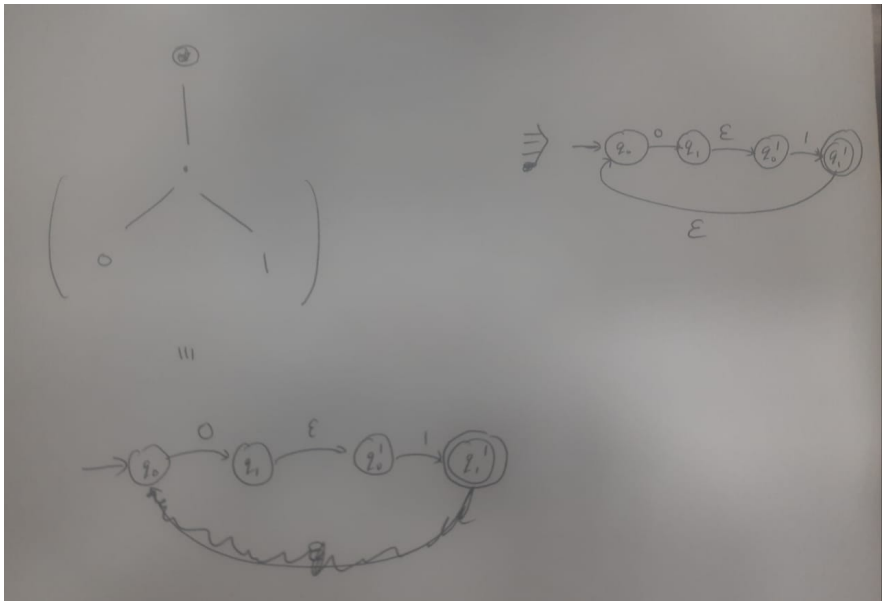

Figure 5: A simple figure

**For a (left+right) node:** We have the left NFA and right NFA to use as tools. In the final NFA, we are to accept strings from both languages, so we have to guess if the string probably belongs to the left language or the right language. So, we just make a node, which will be our start node for the final NFA, and make $\epsilon$-edges from the new starting node to the previous starting nodes.

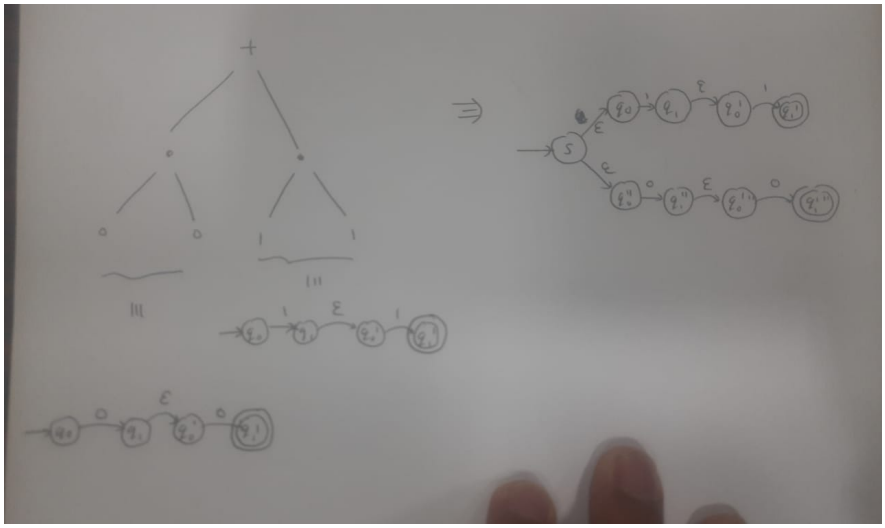

Figure 6:

We have covered all the cases of a node - $*$, $\cdot$, $+$ and the leaf nodes, so we keep doing this for every height, and we come up with the final NFA!

Formally, to prove this algorithm (implementation is essentially DP!), you would use strong induction on the height of the node; a node can either be $*$, $\cdot$, $+$ or a leaf node. So, you combine the

children NFAs using the combination operation given above to produce the NFA for the considered node. This was your inductive step. For the base case, your node is a leaf node, i.e. just a character from the alphabet($\Sigma$). One can trivially construct an NFA for this. Just take one start state and one final state, and create an edge that takes the character for which you are constructing the NFA.

This result tells us that we RegEx can represent only languages that are regular, but this does not say if RegEx can represent all regular languages.

## 3 Expressing NFAs as RegEx

Well, this is some interesting intuition! So, you have to construct a RegEx from an NFA. How would you do it? Can we generalise edges for whole languages[2] instead of just characters? Can't we just represent every NFA with a start and an end state, with an edge, that takes the language of the NFA as an input from the start to end state? Well, we can! We will be using such ideas to illustrate this equivalence.

So, let's take the example of this NFA. We will derive a RegEx for this NFA using some transformations. As for the NFA, we will treat every edge taking strings[3] as input, instead of just characters. **Notice this constraint that $\epsilon$-edges are only allowed as outgoing edges from the start node and incoming edges to the final node, and there is only one final state. This can be produced by $\epsilon$-closures and the introduction of $\epsilon$-edges, respectively.**

$$\Sigma = \{a, b\}$$

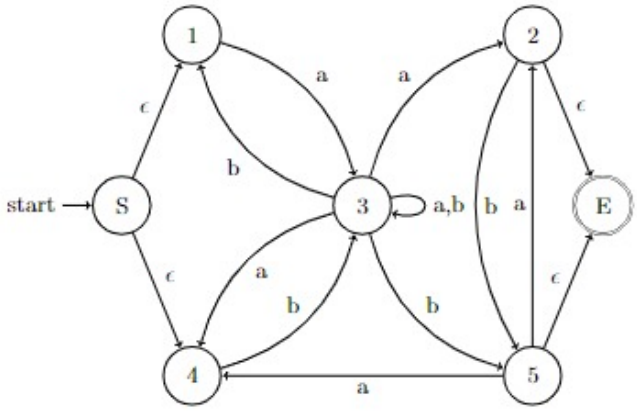


Figure 7:

We will be reducing this NFA to another NFA that accepts the same regular language, and the edges represent regular languages instead of RegEx. We will keep doing this until we reach an NFA such that there is just one final and one start state, so the edge's Regex will represent the initial NFA's regular language. The question is, how do we make such reductions? So, in this example,

[2]Basically, a function $\delta : Q \times L \longrightarrow Q$, where L is a language, so the function takes a string at a particular state and goes to another state.

[3]the character strings, eg. "a" for a

we will try to remove one of the nodes, combine its incoming and outgoing edges, and create a new edge for every pair that takes a regular language in the form of a RegEx. Now, we carry out strong induction(correctness of the algorithm) on the number of nodes in the NFA so a RegEx can represent the reduced NFA. Since the initial and reduced NFAs represent the same regular language, a RegEx exists for every regular language. Let's do this step by step! We will be removing the q3 node.

Step 1: Combine the incoming, outgoing edges and self-loop of q3 that connect q1 and q4 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q1 to q4.

$$q1 \xrightarrow{a} q3 \xrightarrow{a,b} q3 \xrightarrow{a} q4$$

Now, let's simplify this-

$$q3 \xrightarrow{a,b} q3$$

As the above edge takes two characters - 'a' and 'b', we can represent this edge by the RegEx (a+b). But it also is a self-loop, so we can take this as $(a + b)^*$

$$q3 \xrightarrow{(a+b)^*} q3'$$

Now, coming back to the entire transition, we can express it as

$$q1 \xrightarrow{a} q3 \xrightarrow{(a+b)^*} q3' \xrightarrow{a} q4$$

Also, edges in a sequence, as above, can be combined into one edge by the CONCAT(.) operator. The following9 is the final transformed NFA. We delete the outgoing edge that was considered iff becomes redundant[4]

$$q1 \xrightarrow{a.(a+b)^*.a} q4$$

Now, we repeat the same procedure for combining incoming and outgoing edges for the following steps.

Step 2: Combine the incoming, outgoing edges and self-loop of q3 that connect q1 and q1 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q1 to q1.

Step 3: Combine the incoming, outgoing edges and self-loop of q3 that connect q1 and q2 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q1 to q2.

Step 4: Combine the incoming, outgoing edges and self-loop of q3 that connect q1 and q5 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q1 to q5.

Step 5: Combine the incoming, outgoing edges and self-loop of q3 that connect q2 and q2 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q2 to q2.

[4]i.e., that is, all incoming edges have been paired up with this outgoing edge.

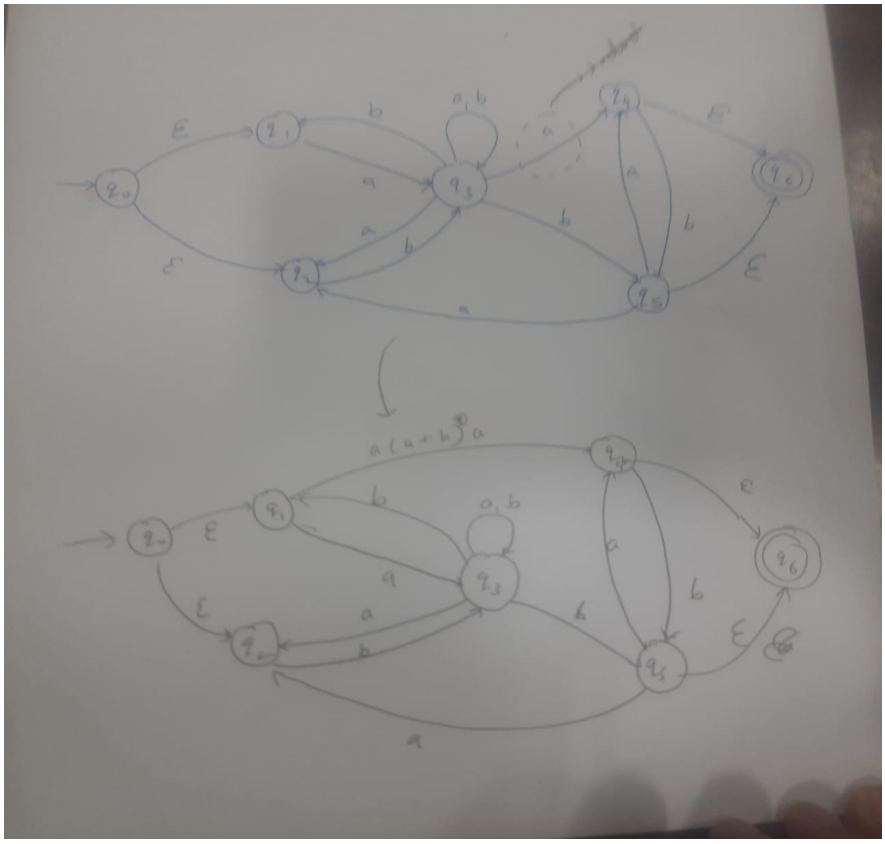

Figure 8: Step 1

Step 6: Combine the incoming, outgoing edges and self-loop of q3 that connect q2 and q1 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q2 to q1.

Step 7: Combine the incoming, outgoing edges and self-loop of q3 that connect q2 and q4 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q2 to q4.

Step 8: Combine the incoming, outgoing edges and self-loop of q3 that connect q2 and q5 and represent it as an edge, and construct a RegEx for this such that all strings in it cause a transition from q2 to q5.

Now, the following is the Final Reduced NFA:

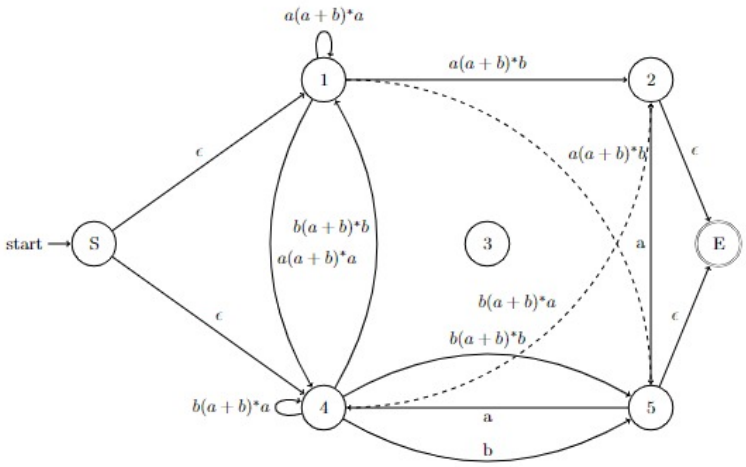


Figure 9: Final Reduced NFA

So, what now? Our motive was to generate a RegEx for our initial NFA! And we have said that, upon these reductions, the regular language accepted remains the same. So, even after the removal of one node, the regular language accepted remains the same. So, we keep removing nodes with the help of these reductions and reach to a state where there is just a start state and an end state, with an edge from the start to the end state. And we know that edges are labelled as languages with regular expressions. So, this RegEx on this final edge represents the regular language for the initial NFA.

**Some caveats:** what if the node being removed has an $\epsilon$-edge? We will treat it as an empty string, as usual.