CS 4510 Automata and Complexity	January 24
Lecture 4	
Lecturer: Frederic Faulkner	*Scribe(s):*

1 Nondeterminism

Last class we introduced the notion of non-determinism. Recall that an NFA differs from a DFA in three ways:

- It can have "missing" transitions
- It can multiple transitions for the same state and input
- It can have optional ε-transitions that don't read the next character of the input yet.

Since an NFA, unlike a DFA, has to make "decisions" as it runs, we also had to redefine what it means for a nondeterministic machine to accept a string. Recall that an NFA N accepts a string x if *any* sequence of choices it makes while processing x leads to an accept state. Thus, an NFA rejects x only if *every* sequence of choices either ends in an reject state or gets "stranded" (i.e. doesn't have a transition for the current input.)

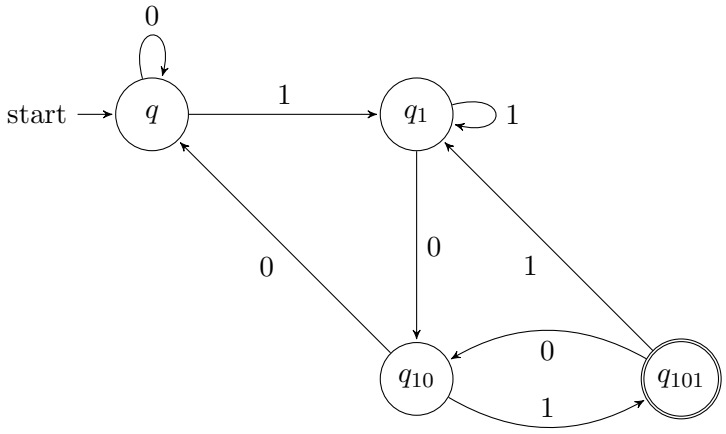
Here are the formal definitions related to NFA's. (NOTE: the definitions given in last class's lecture notes are slightly incorrect as they don't mention ε-transitions. They are correct here.)

Definition 1 *Formally, a nondeterministic finite automata N is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$ where*

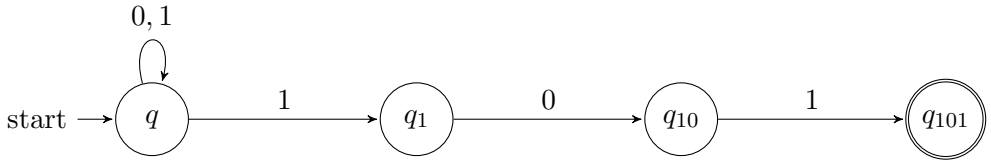
- *Q is the set of states*
- *Σ is the input alphabet*
- *$q_0 \in Q$ is the start state*
- *$F \subseteq Q$ is the set of final states*
- *$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ is the transition function*

Definition 2 *For a given NFA N with states Q, define the ε-closure of a subset S of Q to be the set of a states reachable from states in S by ε-transitions. We denote that as $E(S) = S \cup \{q' |$ there is a sequence of ε transitions from some state $q \in S$ to $q'\}$. An NFA N is said to accept x, if there exists a sequence $q_0, ..., q_{|x|}$ of states of N such that q_0 is the initial state of N, $q_{|x|}$ is an accepting state of N, and $q_{i+1} \in E(\delta(q_i, x[i]))$ for $i \in [0...|x|-1]$ where δ is the transition function of N.*

Now let's see some non-determinism in action! We can use it to simplify tasks that we have already done previously. For example, here is the DFA for the strings that end in 101:



And here is a simpler NFA which does the same thing:



Consider how this string processes the input 110101. It can do the following:

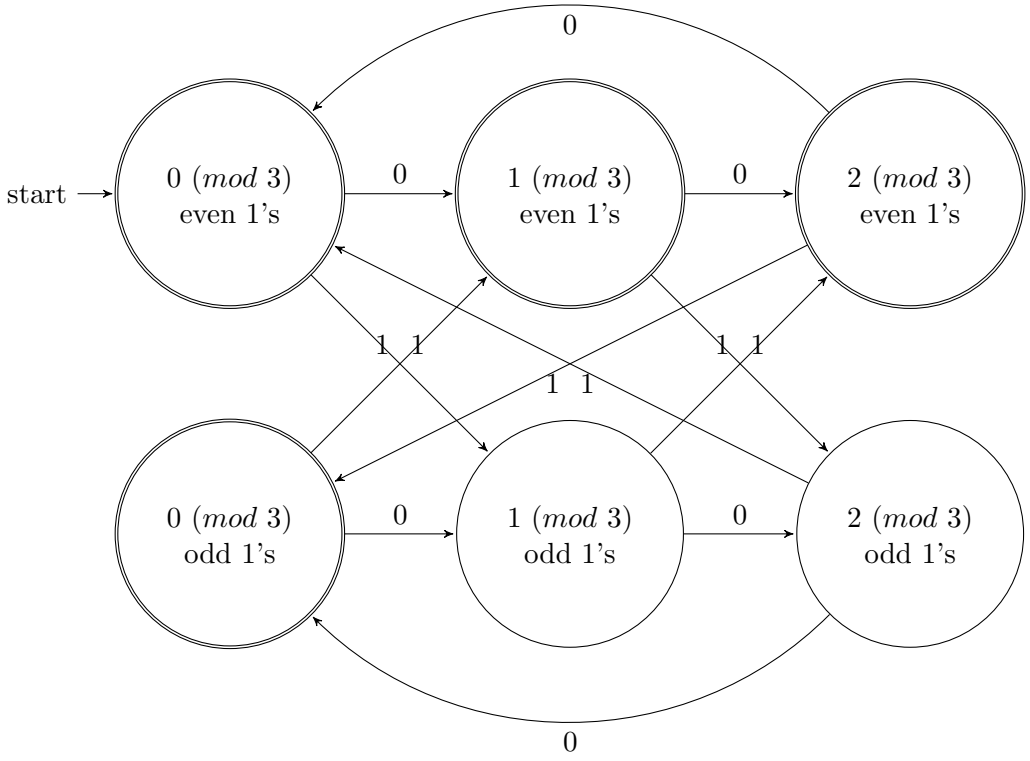
- $q \xrightarrow{1} q_1$ (no loops, get stranded)
- $q \xrightarrow{1} q \xrightarrow{1} q_1 \xrightarrow{0} q_{10} \xrightarrow{1} q_{101}$ (by looping once, gets stranded because there are still two characters left in the string but no available transitions)
- $q \xrightarrow{1} q \xrightarrow{1} q \xrightarrow{0} q \xrightarrow{1} q_1 \xrightarrow{0} q_{10} \xrightarrow{1} q_{101}$ (loop three times, end in accept state)

Since it is possible for us to end in an accept state after processing all the input, we accept 110101, like we wanted to. (Note the importance of processing all of the input: the input 1010 technically can get "stranded" in the accept state before reading the final 0; this doesn't count as ending in an accept state, so 1010 will not be accepted.)

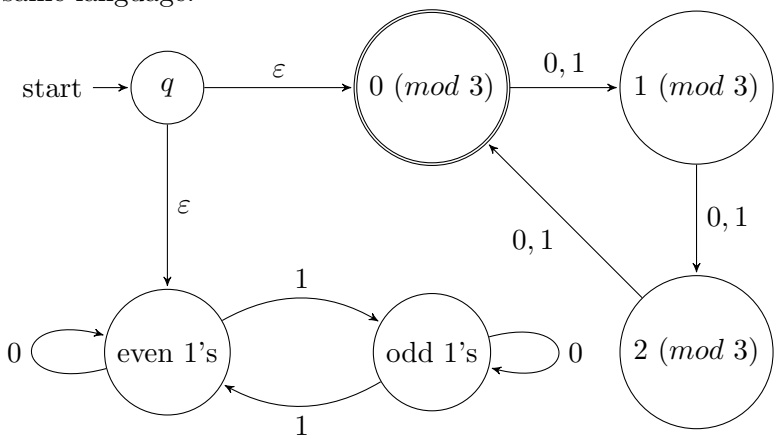
In class, we created a DFA for the language

$$\{x | x \text{ is a binary string that has an even number of 1's or length divisible by 3, or both}\}$$

We created it by first creating separate DFA's to track the two separate characteristics of having an even number of ones or length divisible by 3, and then combining them. The result looked something like this:



However, we can use ε-transitions to make a much easier-to-read NFA that recognizes the same language.



Here, the machine uses nondeterminism once (i.e. it makes one "choice"). It guesses whether it should keep track of the length of the string or the number of ones. Then it just runs the corresponding DFA for that language. For strings not in the language, neither DFA will accept; but for strings in the language at least one of the DFA's will accept.

You might notice that this lends itself to a much more straightforward proof of the theorem that we proved last week:

Theorem 3 *If L_1 is regular and L_2 is regular, then $L_1 \cup L_2$ is regular.*

Proof. Since L_1 and L_2 are regular, they have DFA's D_1 and D_2. Create an NFA for

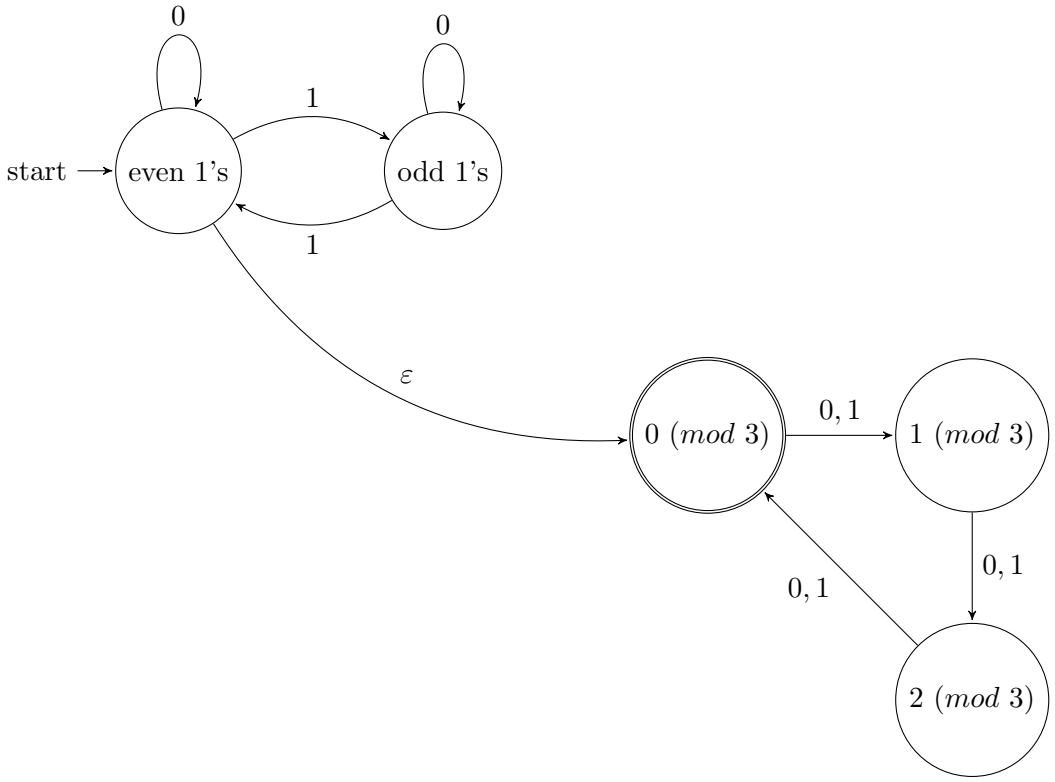
$L_1 \cup L_2$ by adding a new start state q, and adding two ε transitions from q to the old start states of D_1 and D_2.

Now we can also do the proofs that we omitted in the previous notes for the closure of regular languages under concatenation and Kleene star.

Theorem 4 *If L_1 is regular and L_2 is regular, then $L_1 \circ L_2$ is regular.*

Proof. Let D_1 and D_2 be DFA's for L_1 and L_2. Then, construct an NFA N for $L_1 \circ L_2$ by setting the start state to be the old start state of D_1 and connecting each of the old final states of D_1 to the old start state of D_2 via ε-transitions. Then the final states of N are the final states of D_2.

Here is an implementation of this construction using the same two DFA's that we just used for the union example:

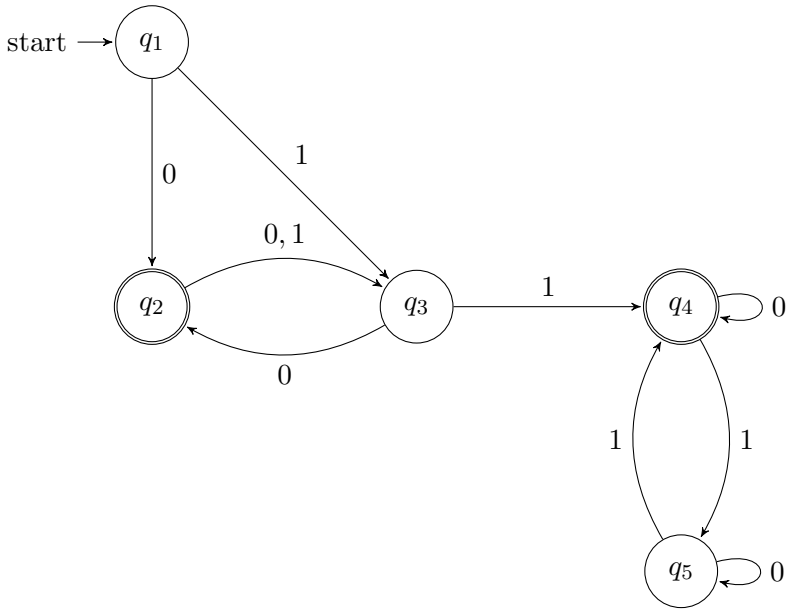


Notice that the ε-transition between the two machines allows the machine to decide at what point the string from the first language ends and the second one begins.

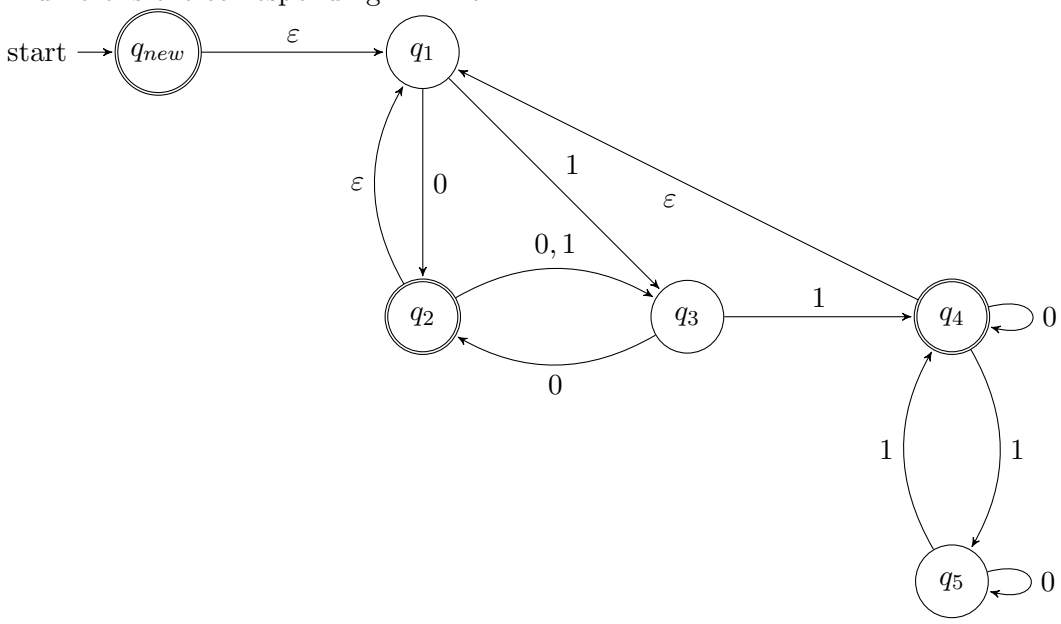
Theorem 5 *If L is regular, then L^* is regular.*

Proof. Given D, a DFA for L, we convert it into an NFA for N by connecting each final state to the start state via ε-transition. Then we add a new start state which is also an accepting state, and connect it to the old starting state via ε-transition. (We do this to accept the empty string.)

As an example, here is some DFA for some language L:



And here is the corresponding NFA for L^*:



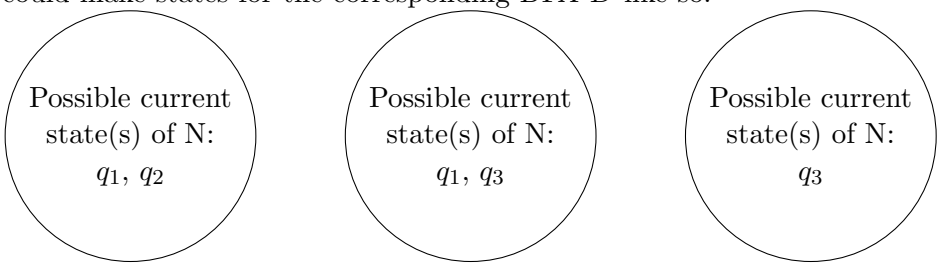
But wait! you object. *How does this prove that the languages are regular? I thought regular languages had DFA's, not NFA's...*Good question! It is not obvious yet what relationship exists between NFA's and DFA's. Adding non-determinism to an automata makes them easier for humans to design; does it also make them more powerful? It turns out that it does not. Any language with an NFA that recognizes it is guaranteed to *also* be recognized by some DFA.

Theorem 6 *A language is regular if and only if it is recognized by an NFA.*

Proof. Clearly all regular languages have NFA's, since all regular languages have DFA's

and it is trivial to convert a DFA to a NFA. Thus we must now show that we can "remove" the need for non-determinism by showing how to convert an NFA to a DFA.

The trick is that we can keep track of every state that we could conceivably be in so far as we read the input. Thus, suppose that an NFA N had three states: q_1, q_2, and q_3. Then we could make states for the corresponding DFA D like so:



And so on. (There would be eight states in total.) Suppose after reading, say, 101, that we are in the left most state. That would mean that, no matter what choices that N made, after reading 101 it is possible that N is in state q_1 or q_2, but that there is no way for it to be in state q_3. The transition function works as follows: given an input a and a set of possible states for N, check to see which states you can reach from the old states by reading in an a and then taking any number of ε-transitions. The start state of D will be the start state of N plus any states that the start state can reach by ε-transitions, and the final states will be all sets of states of N that include a final state.

Formally, let N be an NFA defined as $(Q, \Sigma, q_0, F, \delta)$. Then let D be a DFA defined as $(P(Q), \Sigma, E(q_0), G, \gamma)$, where $E()$ is the ε-closure function defined earlier, $G = \{S \subseteq Q | S \cap F \neq \emptyset\}$, and $\gamma : P(Q) \times \Sigma \to P(Q)$ is defined as $\gamma(S, a) = \cup_{q \in S} E(\delta(q, a))$.