CS 4510 Automata and Complexity January 26

Lecture 5

Lecturer: Frederic Faulkner *Scribe(s):*

1 Regular Expressions

We have already seen two ways to represent regular languages, namely, deterministic finite automata and non-deterministic finite automata. Today we will discuss a third way: regular expressions. Regular expressions effectively provide a way to build up large regular sets out of smaller ones. Starting with just four rather small regular sets, $\emptyset$, $\{\varepsilon\}$, $\{0\}$ and $\{1\}$, we can build every regular language by means of $\cup$, $\circ$ and $*$.

It can help to think of a regular expression as a template for the strings in the language. For example, consider the following familiar language: $L_1 = \{x \mid x \text{ is a binary string that ends in } 101\}$. Every string in this language has the form $xxxxxxxxxx101$, where the x's represent an arbitrary binary string. But we know that the set of arbitrary binary strings is a regular set! Typically we denote it Σ^*. So $\Sigma^* \circ \{101\}$ is the same set as L_1.

(Note: For regular expressions, we typically omit both the set brackets and the $\circ$ operator, so we would write it as Σ^*101.)

Let's try another. Let $L_2 = \{x \mid x \text{ is a binary string of length at least 2 that starts and ends with the same character}\}$. Either the string looks like $0xxx..xxxx0$ or $1xx...xxx1$. Again, the x's can be any arbitrary string, so we end up with the two sets $\{0\} \circ \Sigma^* \circ \{0\}$ and $\{1\} \circ \Sigma^* \circ \{1\}$. How do we combine them? Well, with $\cup$, of course. After removing the brackets and $\circ$'s, we get the final expression $0\Sigma^*0 \cup 1\Sigma^*1$.

Definition 1 *Regular expressions over an alphabet Σ are defined recursively, as follows:*

- *$\emptyset$ is a regular expression*
- *ε is a regular expression*
- *For each $a \in \Sigma$, a is a regular expression. (So for binary regular expressions, for example, 0 and 1 are defined by this step to be regular expressions)*
- *If R_1 and R_2 are regular expressions, then $R_1 \cup R_2$ is a regular expression*
- *If R_1 and R_2 are regular expressions, then $(R_1) \circ (R_2)$ is a regular expression.*
- *If R is a regular expression, then $(R)^*$ is a regular expression.*

For anyone who has studied regular expressions (or perhaps *regex*), the ones that we study in class may feel a bit stark, in that we will only use the three operators $\cup$, $\circ$, and $*$. We will see soon that these three operators are sufficient to express all regular languages, which means that most other operators you may be familiar with are actually just shorthand for some combination of $\cup, \circ$ and $*$.

For example, if R is a regular expression, $R?$ often is used to represent zero or one occurrences of a string matching R. We can say $R \cup \varepsilon$ to represent the same thing.

Similarly, R^+ is often used to mean one or more occurrences of a string matching R. We can express the same thing like so: RR^*.

Note the difference between, say, 00^* and $(00)^*$. The first matches the strings of one or more zeroes, while the second matches all even length strings of zeroes. This is because regular expressions have an order of precedence. Remember **PEMDAS** from grade school? **P**arentheses are the highest precedence, then **E**xponents, then **M**ultiplication/**D**ivision, and finally **A**ddition/**S**ubtraction. Similarly, for regular expressions, parentheses are highest, then Kleene Star, then concatenation, then finally union. ("()" $\geq$"$*$"$\geq$"$\circ$"$\geq$"$\cup$".)

Theorem 2 *A language L is regular if and only if there is a regular expression R that expresses L.*

Proof. The backwards direction, that every regular expression has a corresponding NFA (and therefore DFA), has a fairly simple proof, especially since we have already shown how to construct NFA's for the union, concatenation and Kleene star of regular languages. The full proof is in the book, but the idea is to work through the six bullet points in the formal definition above, showing the corresponding NFA for each item in the definition. (I.e. construct NFA's for $\emptyset$, then for $\{\varepsilon\}$, then for $a \in \Sigma$, etc.)

We will focus on the other half of the proof by demonstrating a method to convert an DFA to a regular expression. To do so, we will need an automata called a *Generalized Non-Deterministic Finite Automata (GNFA).*

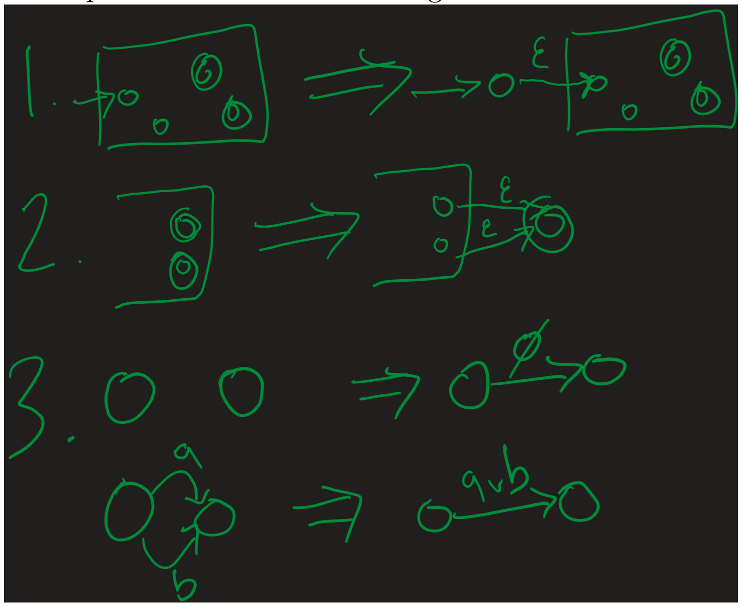
Definition 3 *A GNFA is an NFA whose transitions can be labeled with any regular expression rather than just ε or single characters. In addition, we require that*

- *there are no transitions into the start state*
- *there is exactly one final state*
- *except for the start and accept states, there is exactly one transition from each state to each other state*

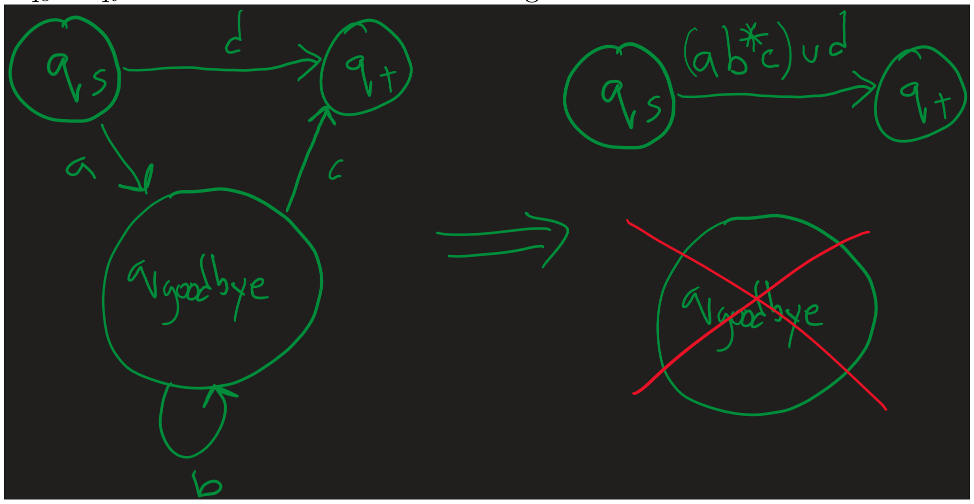
The most notable difference between an NFA and a GNFA is that GNFA transitions can contain full regular expressions. Thus a GNFA can read more than one character per transition; in a single step it can read any number of consecutive characters that together match the corresponding regular expression.

Here is how we will convert a DFA to a regular expression: we will first convert the DFA to a GNFA with 2 additional states. Then, we will remove states from the GNFA one state at a time while taking care not to change the language accepted by the GNFA. Finally, when we have removed enough states that there are only two states and one transition remaining, the label on the transition will be a regular expression equivalent to the starting DFA.

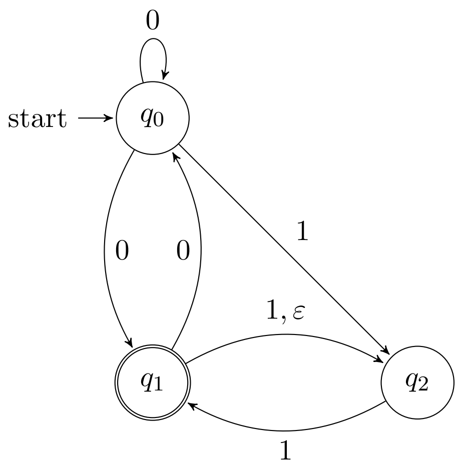
First we convert the DFA to a GNFA. To prevent incoming arrows to the start state, add a new start state with an ε-transition to the old start state. Then, to ensure that there is only one accepting state, we add a new accepting state; the old accepting states are no longer accepting, but there is an ε-transition from each old accepting state to the new accepting state. Finally, we must ensure there is exactly one transition from each state to each other state. For each missing transition, add a new transition marked with $\emptyset$. (This doesn't change the language, since $\emptyset$-transitions cannot be taken.) Then, if there are, say two transitions a and b between two states, they become a single transition marked by the regular expression $a \cup b$. See the image below.



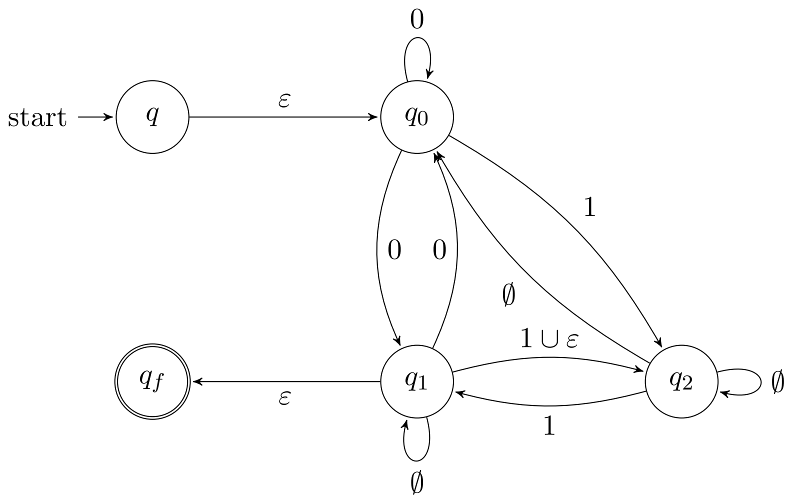
Now we can remove each state one at a time except for the new start state and final state. Each time we remove a state, we update every other transition in the machine. Suppose we remove the state $q_{goodbye}$. We now must update every transition between each pair of states. Suppose that the transition from q_s to q_t was labelled by regular expression d. In addition, the transitions from q_s to $q_{goodbye}$, from $q_{goodbye}$ to itself, and $q_{goodbye}$ to q_t were marked with the regular expressions a, b, and c respectively. Then the new transition from q_s to q_t would be $ab^*c \cup d$. See the image below.



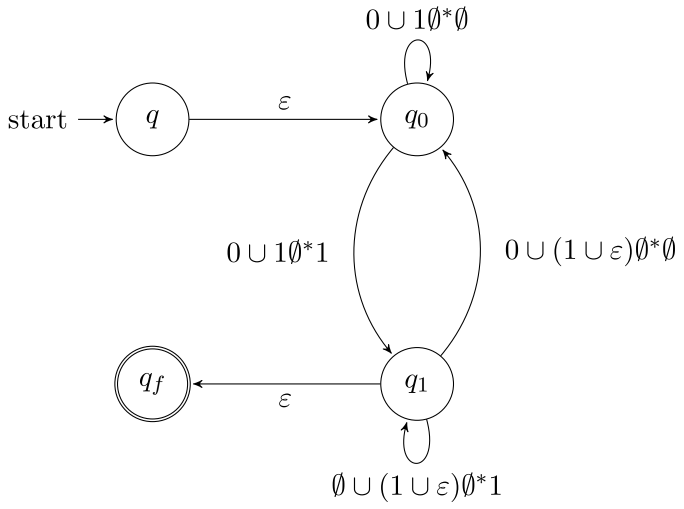
We will walk through the first couple steps of an example.Here is an NFA that we have seen before:



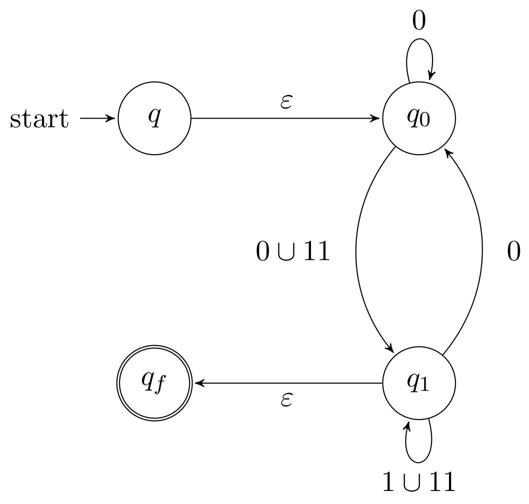
First we convert it to a $GNFA$:



Now we will show what happens when we remove $q2$:


start
q
q_0
q_1
q_f
ε
$0 \cup 1\emptyset^*\emptyset$
$0 \cup 1\emptyset^*1$
$0 \cup (1 \cup \varepsilon)\emptyset^*\emptyset$
ε
$\emptyset \cup (1 \cup \varepsilon)\emptyset^*1$

Which, after simplifying the regular expressions, becomes:



The rest of the conversion is left as an exercise for the reader.