Lecture 4: Non-Regular Languages

Ryan Bernstein

Computer Science 311
Spring 2016

1 Introductory Remarks

- Assignment 1 is available on the course web page. It's due April 19th
- Assignment 0 has been graded, but grades haven't been entered. That should be done by tonight, and I'll hand them back on Thursday.

1.1 Errata

To start, a minor, but important correction. When discussing NFAs with ϵ-transitions, I gave you a formal definition of an NFA as a 5-tuple $(Q, \Sigma_\epsilon, \delta, q_{start}, F)$. Notably we said that the alphabet of this NFA was Σ_ϵ, which is created by taking the union of the input alphabet Σ and the singleton set ϵ.

In a formally-described NFA, the second element, Σ, is *only* supposed to represent the input alphabet — that is, the set of all possible characters that appear in any string in the language of this NFA. As the input alphabet, Σ should *not* include ϵ. We do have to make a minor modification to our NFA to account for ϵ-transitions, but this modification is actually to δ.

In an NFA with ϵ-transitions, we say that $\delta : (Q \times \Sigma_\epsilon) \to \mathbb{P}(Q)$. In an NFA *without* ϵ-transitions, we simply say that $\delta : (Q \times \Sigma) \to \mathbb{P}(Q)$. In either case, Σ itself remains unchanged.

This distinction will come up again when we discuss other types of nondeterministic machines on Thursday.

1.2 Recapitulation

Last lecture, we introduced the concept of a *regular expression.* Regular expressions allow us to succinctly express regular languages using a string of either mathematical symbols (for uses in theory) or printable characters (for uses in industry).

We gave a recursive definition of a regular expression, which in turn gave us a recursive definition of a regular language. We say that a regular language is one of six possibilities:

- $\emptyset$, the empty set
- $\{a\}$ for some *specific* single character $a \in \Sigma$
- ϵ, the empty string

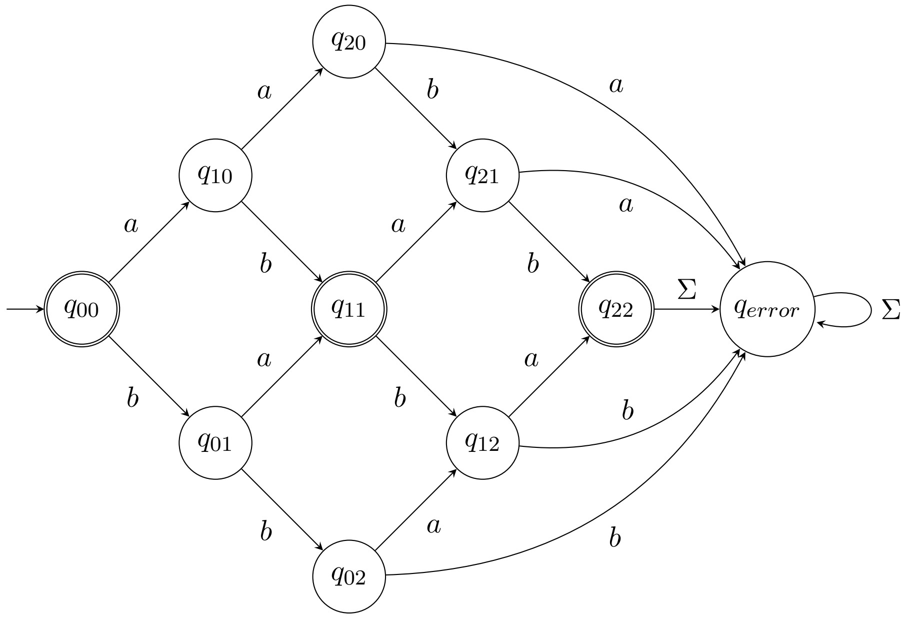
- $A \cup B$ where A and B are other, smaller regular languages
- $A \circ B$ where A and B are other, smaller regular languages
- A^*, where A is another, smaller regular language

2 Limits of Regular Languages

2.1 Limits in the Concrete

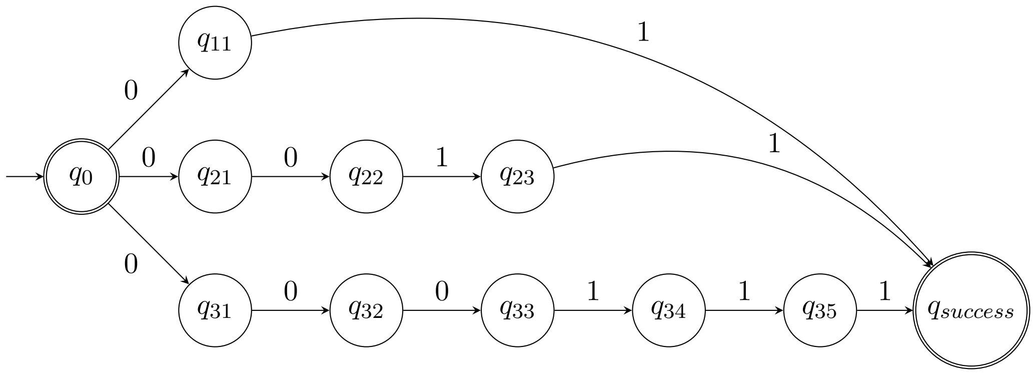
Let's get a little more practice building finite state machines. We'll explore how the construction process works, and in doing so, discover a bit more about what it's capable of.

Example 1 Construct a DFA that decides $A = \{s \in \{a, b\}^* \mid s \text{ contains } i \text{ } a\text{s and } i \text{ } b\text{s for some } i \in \{0, 1, 2\}\}$



How does this scale? Note that for a given upper bound n, we need $(n+1)^2 + 1$ states in our DFA. If we wanted to generalize this to other alphabets, we might say $(n+1)^{|\Sigma|} + 1$. This is because we can have between n different values of each character before hitting our error state.

Example 2 Construct an NFA that decides $B = \{0^n 1^n \mid 0 \leq n \leq 3\}$.



Even with an NFA, it's still clear that this machine grows pretty quickly as well. Each increase in n requires the addition of another branch, which is $2n - 1$ states in length. Unlike the first one, however, that this isn't a minimal DFA, so we could potentially grow by a smaller amount, but the important thing to note is that a finite automaton for this language grows proportionally to the upper bound on n.

2.2 Limits in the Abstract

We've now seen three types of finite automata that we can use to decide regular languages: DFAs, NFAs, and generalized NFAs.. We know that if a language is regular, then some such machine exists to decide it, and this is where the limits of regular languages come from.

At any point during a computation in one of these machines, the only *context* that I'm allowed to save — that is to say, the only details that I'm allowed to "remember" — are a state (in a DFA) or a set of states (in an NFA or GNFA). This is especially limiting because we have a finite number of states to choose from (or in a nondeterministic machine, a nondeterministic number of subsets of Q). We only remember the state(s) that we're currently in, not how we got here. This means that if I want to use some property — say, the length, or the number of as — that I've seen so far, I need to have enough states to track that property.

The number of states required for the languages we've just seen is some function of an upper bound n, where n is dependent on the language in question. What happens if we want to remove this upper bound? What if we wanted to decide languages like:

- $\{s \in \{a, b\}^* \mid s$ contains the same number of as and bs$\}$
- $\{0^n 1^n$ for any $n \in \mathbb{N}\}$

Since the number of states required in each machine is a function of n, leaving n unbounded means that the number of states is also unbounded. All of the machines that we've seen so far are examples of *finite* automata, so named because they have a finite number of states. This is the limiting factor of regular languages: they are unable to *count infinitely.* We'll formalize this property and use it to develop a test that allows us to prove that a language is non-regular.

3 The Pumping Lemma

Let's look back at our formal definition of computation in a DFA:

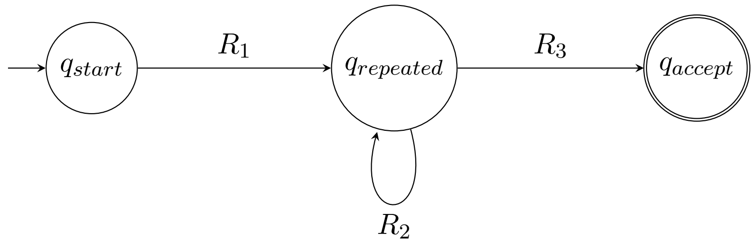
> A DFA $(Q, \Sigma, \delta, q_0, F)$ accepts s if Q contains a sequence of states $(r_0, r_1, \ldots, r_n)$ such that:
>
> 1. $r_0 = q_0$
> 2. $\delta(q_i, s_{i+1}) = r_{i+1}$ for all $i \in [0, n)$
> 3. $r_n \in F$

We know that any string passed to a DFA will pass through a series of states. Since this machine is deterministic and contains no ϵ-transitions, we know exactly how long this sequence will be for any string s: since we begin in the start state and take one transition for every letter in s, we'll pass through exactly $|s| + 1$ states.

We know that our set of states is finite. What happens, then, if we pass a very long string into our DFA? Specifically, if $|s| > |Q|$, we know that some state in our sequence appears more than once, by the Pigeonhole Principle.

It's here that the idea of a GNFA becomes helpful once again. If you don't remember *all* of the details, that's okay. The important part is just that transitions in a GNFA are labeled with regular expressions, rather than single characters. We take a transition in a GNFA if we pass in a substring that matches the regular expression with which that transition is annotated.

Since we know that a loop or cycle appears somewhere in our sequence of states, the GNFA path that our string travels through would look something like this:



Since we know that 1) this string that was accepted and 2) a loop or cycle exists, we know that for whatever our repeated state was, none of R_1, R_2, or R_3 is $\emptyset$.

Remember that since any string longer than $|Q|$ *must* visit a state more than once, *every* sufficiently long string in our language will have this form. The repeated state may not be the same one, but we know that some repeated state for which we can draw a diagram like this exists.

We therefore know that any string longer than $|Q|$ in our language can be divided into three parts xyz such that:

1. x matches R_1
2. y matches R_2
3. z matches R_3

Last time we used a GNFA in this manner, we reduced a path much like this one into a single regular expression of the form $R_1 \circ R_2^* \circ R_3$. Why did we take the Kleene closure of R_2?

We know two things about q_{repeat}:

1. There is a path out of q_{repeat} that leads to an accepting state
2. There is a path out of q_{repeat} that leads, directly or indirectly, back to q_{repeat}

This means that any string w should be in the language if w starts with x, ends with z, and either:

- Omits y entirely, in which case we skip the loop and proceed straight along the accepting path R_3, or
- Contains more repetitions of y, in which case we traverse the loop more than once

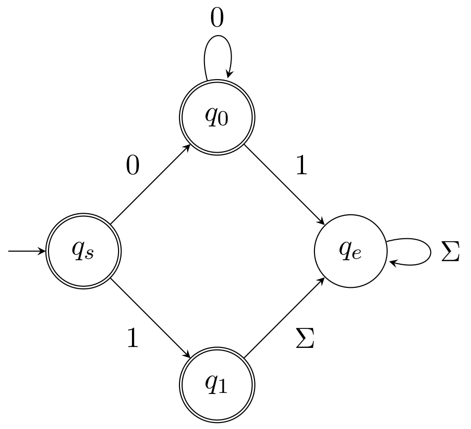
We'll now formalize this into a concrete rule.

3.0.1 The Pumping Lemma, Formalized

If L is a regular language, then there exists some length p, called the *pumping length*, such that any string $s \in L$ with a length greater than or equal to p can be split into three parts $s = xyz$ that satisfy the following conditions:

1. For any $i \geq 0$, $xy^iz \in L$
2. $|y| > 0$
3. $|xy| \leq p$

Why do we declare p as a separate variable, rather than simply using $|Q|$? Consider the following DFA, which decides $\{0\}^* \cup \{1\}$:



This DFA has four states, but by inspection, we can see that any string $s \in L(M)$ must enter a loop after the second character. To summarize, $|Q|$ need not be equal to p; it is simply an upper bound.

We insist that $|y| > 0$ because we know that our loop must exist. And since we've established $|Q|$ as our upper bound on p, we know that we must traverse that loop within the first $|Q|$ characters.

3.1 Using the Pumping Lemma

We've just derived an important property of regular languages. How can we use this to prove that a language is *not* regular? The answer is a pretty standard proof by contradiction: we assume that a language *is* regular, and then use the Pumping Lemma to show that this property does not hold.

A tricky part of this application is the fact that we only know that three parts x, y, and z exist in s — we don't get to choose what they are, because we instantiate them existentially.

Fortunately, we know that this property holds for *any* string s. This means we can choose a string that limits the possibilities for x, y, an z.

Example 1 Use the Pumping Lemma to show that $A = \{0^n1^n \mid n \in \mathbb{N}\}$ is not regular.

Assume that A is regular. The string 0^p1^p is longer than p, so we should be able to divide it into three parts xyz such that the required properties hold.

Since we mandate that $|xy| \leq p$, we know that y is composed entirely of zeroes. If we "pump up" from xyz to xy^2z, we therefore have more zeroes and ones. This means that xy^2z is not an element of A. By the Pumping Lemma, we can therefore conclude that A is not regular.

Example 2 Use the Pumping Lemma to show that $B = \{s \in \{a, b\} \mid s$ contains the same number of as and bs$\}$ is not regular.

We can actually use the exact same string and strategy as we did in the last example here. Note that $\{a^nb^n\} = \{a^*b^*\} \cap B$. We know that regular languages are closed under intersection (at least if we've finished Assignment 1), so if $\{a^nb^n\}$ is not regular and $\{a^*b^*\}$ is, we can conclude that B is not regular.

Of course, we can use the Pumping Lemma even if we *haven't* yet proven closure under intersection. Since we get to choose any string s we want, we can again pick a^pb^p — since it has an equal number of as and bs, it should be an element of B.

Example 3 Use the Pumping Lemma to show that $C = \{1^{n^2} \mid n \in \mathbb{N}\}$ is not regular.

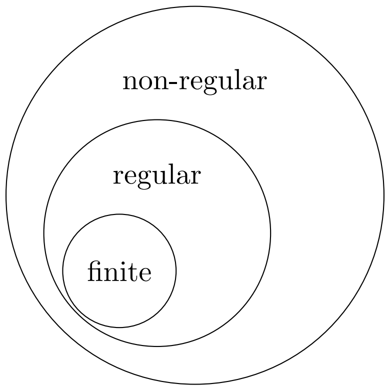
Let $s = 1^{p^2}$.

We don't know how long y is, but we do know that the next string in the language will be $s^{(p+1)^2}$. Any string in between these two lengths will not be an element of the language.

We know that the maximal possible length of y is p, but we also know that $(p + 1)^2 = p^2 + 2p + 1$. $xyyz = 1^{p^2+|y|}$. Since $|y| < (2p + 1)$, we know that $xyyz \in C$.

4 The (Currently Incomplete) Chomsky Hierarchy of Languages

This allows us to introduce what's known as the Chomsky Hierarchy of Languages, which allows us to categorize languages based on their properties. As the word "hierarchy" implies, we're getting broader in scope, and the larger classes encompass the smaller ones. We can therefore draw a Venn diagram for our language classes like this:



This diagram is incomplete. We'll be adding more classes over the course of the term. For now, though, it gives us a good intuitive idea of where this course is going.

4.1 Caveat: Sets of Sets

Remember that when we partition languages like this, we're dealing with sets of languages. Languages are sets themselves. A subset of a language within a given class does *not* necessarily have to be in a subclass. As an example, recall the second definition of a language that we were given:

> A *language* is a subset of Σ^*.

Σ^* is itself a regular language. We can draw a very simple one-state DFA that decides it. This means that all languages, *even the ones that are not regular*, are subsets of a regular language.