Lecture 1: DFAs and Regular Languages

Ryan Bernstein

1 Introductory Remarks

1.1 Administrative Details

- Assignment 0 should now be available on the course web site. This assignment covers the review topics that we discussed on Tuesday. I meant to assign this on Tuesday, but forgot to mention it. This will be due one week from today.
  - I prefer submissions on paper, but you can submit electronically as well
  - As mentioned, typed or handwritten submissions are fine so long as they're legible
  - We'll be going over solutions in class the day homework is due, so as mentioned in the syllabus, late submissions won't be accepted without prior permission. I like to think I'm pretty reasonable, so if you have personal circumstances that you think are going to prevent you from finishing on time, let me know.
- In many lectures, starting today, I'll be handing out a worksheet. This worksheet isn't collected or graded. It's just a practice exercise. These worksheets will be available on the course web site in the "Worksheets" tab. Downloading a clean copy is a great way to study for the midterm/final.
- As always, please let me know if you have comments or suggestions for ways that I can help you learn better. Don't hesitate to interrupt me if you have questions or if you feel something is unclear.

1.2 Recapitulation

Last lecture, we did some review of concepts that we've seen in class like CS 250 and CS 251. These included:

- Collection types:
  - Sets
  - Sequences
  - Tuples
- Functions
- Proof strategies:
  - Proof by Contradiction

  - Proof by Induction

We also introduced a couple of new concepts:

- Alphabets, which are finite, nonempty sets of tokens
- Strings or words, which are sequences of characters from some alphabet
- Languages, which are sets of strings or words. We also defined a language as a subset of Σ^*.
- Operations that we perform on strings alphabets and languages, including:
  - Concatenation, in which we create a sequence by mashing two elements of a set together
  - The Kleene closure, in which we concatenate zero or more elements of some set

2 State Machines in General

I've been studying for CS for awhile, and I'm starting to burn out. I've decided to get out of the game and start a new business venture in the wonderful world of tango dancing.

I know literally nothing about tango other than that I've heard the phrase "it takes two to tango". Therefore, I'd like to ensure that:

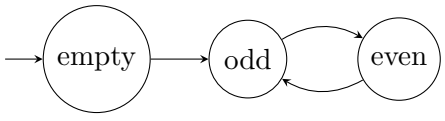
- I have a non-zero number of participants
- The number of participants is even

I have no idea how wildly successful my first event is going to be, so I'll be leaving registration open for an as-yet-undetermined period of time to see how it goes. Since I've sworn off of computers forever, I'll be accepting these registrations on paper. As they roll in, I'll pass them back to you. At any point, I may ask you whether or not our registration conditions have been satisfied to see if we can proceed.

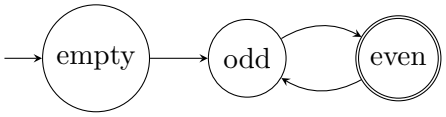
There are a couple of ways that you could approach this task. One way — which is the method that we'd use if we were tracking these registrations in some database — would be to simply accept all registrations, count them, and then see if the result is divisible by two.

We don't actually care about any properties of the count beyond the ones that we've created, though. So a simpler (but equally correct) method of accomplishing the same task would be to track *only* these properties. We discard the value of the count and track only its equality with zero and its divisibility by two.

To do this, we'd start by saying that our count was zero. When we see our first registration, we know that our count was odd. When we saw our second, we'd know that the count was even. With each additional registration, we'd switch back and forth between these two states. We could draw what's called a *state diagram* for this process like so:



The arrow pointing into the "empty" state indicates that this is our *start state*. If we want one or more states to represent true or correct behavior, we draw it in a double circle. Since only the even state matches all of our criteria, that's the only one we update.



We've just built a *finite state machine*, which is our first example of an abstract machine.

3 DFAs

We're going to be looking at a specific type of finite state machine called a *deterministic finite automata*, or DFAs for short. DFAs are used to implement predicates that take strings and either *accept* or *reject* them. Every machine M decides some set of strings, which is called the language of that machine. We sometimes refer to this language as $L(M)$.

When accepting or rejecting a string, *a DFA is never wrong*. This is because, as the notation indicates, $L(M)$ is a function of M. In other words, the language is defined by the machine, and not the other way around. $L(M)$ may not be the language that you intended to decide, but no string that is rejected by M is in $L(M)$, and no string that is accepted by M is *not* in $L(M)$.

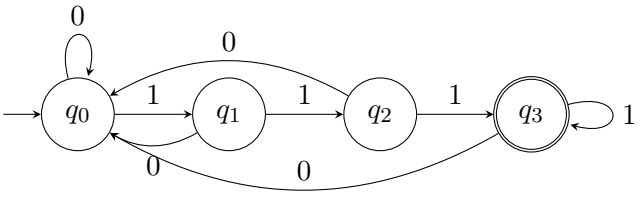
If we look back at the machine we've just created, we can think of this machine as deciding a language over some singleton alphabet (since we treat every registration exactly the same). We can then summarize the language of the machine as follows:

$$L(M) = \{s \mid |s| \text{ is even}\}$$

Note that in general, if our alphabet contains more than one token, we'll need to annotate each transition with the token that induces it.

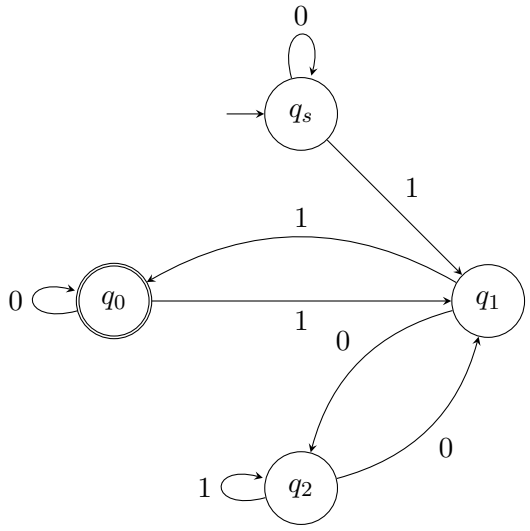
3.1 Examples of DFAs

3.1.1 Strings that End in 111



3.1.2 Binary Multiples of Three

Drawing this DFA is much easier if we consider that every binary string represents some integer number, and that for any $x \in \mathbb{Z}$, $x \ \% \ 3 \in \{0, 1, 2\}$. Taking an existing binary number and adding a 1 to the end is equivalent to multiplying it by two; adding a 0 to the end is equivalent to multiplying it be two and then adding one.



3.1.3 Worksheet Exercise: Strings in $\{a, b\}^*$ that start and end with the same letter

3.2 Determinism

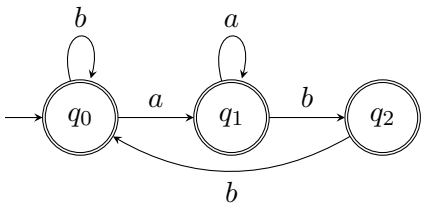
Our textbook mandates that to be a *deterministic* finite automaton, a machine must:

- Have a deterministic set of state transitions, which means that we have only one transition for each character from each state
- Define a transition for *every* character from *every* state. In other words, if we were to represent our transitions as a function, we can have no empty rows — we must define an output in the range for every element of the domain.

The second property is somewhat debatable. Having domain elements for which our function has no defined result doesn't make that function nondeterministic so long as those elements *always* have no defined value — it's still legal for me to go nowhere, provided that's the only place I go. This textbook does require every element of the domain to have a mapping for a DFA to be fully-specified.

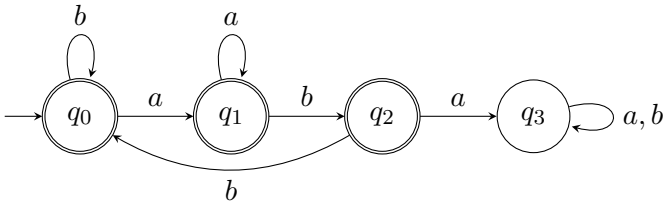
3.3 DFAs with Error States

Let's say we wanted to construct a DFA M such that $L(M) = \{s \in \{a, b\} \mid s \text{ does not include the substring } aba\}$. A tempting way to do it would be drawing something like this:



Since we define no transition from state q_2 for the input a, we could say that the machine crashes, and the string is rejected by default.

However, since the textbook mandates that we include a transition for every possible state/input pair, we do have to include a transition here. We can simulate a crash by adding an inescapable error state, as shown below:



3.4 Formalizing a DFA

Drawing a DFA works well for small, simple languages like the ones that we've looked at here. What happens, though, if we have a DFA that is unmanageably large? By formalizing a DFA as a collection of mathematical constructs, we can make it easier to implement and scale. What are the components of a DFA that we need to formalize?

A DFA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a set of states
- Σ is an alphabet
- $\delta : (Q \times \Sigma) \to Q$ is a transition function
- q_0 is the initial state, or start state. $q_0 \in Q$.
- F is the set of accepting states, also known as *final* states. The term "accepting" is probably clearer, since evaluation of the string can end in any state, regardless of whether or not it is an accepting state. $F \subseteq Q$.

3.5 Examples of Formal Definitions of DFAs

3.5.1 Binary Strings Ending in 111

We have a state for the number of ones at the end of the string that we've seen so far. This can be 0, 1, 2, or 3, so $Q = \{q_0, q_1, q_2, q_3\}$.

Since these are binary strings, $\Sigma = \{0, 1\}$.

If we want to describe the transition function as a computation, we can do this as follows:

- From any state q_i, if we see a 0, transition to q_0.
- From some state q_i for $i \in [0, 2]$, if we see a 1, transition to q_{i+1}
- From q_3, if we see a 1, loop back to q_3.

We can also express this as a table:

q	a	q_{next}
q_0	0	q_0
q_0	1	q_1
q_1	0	q_0
q_1	1	q_2
q_2	0	q_0
q_2	1	q_3
q_3	0	q_0
q_3	1	q_3

The start state is q_0.

$F = \{q_3\}$.

3.5.2 Worksheet Exercise: Binary Multiples of Three

3.6 DFAs as a Model of Computation

Since DFAs are our first abstract machines, they also provide our first model of computation. As we discussed last time, *any* computable problem can be reduced to problems of set membership. Since we can encode anything that can be encoded using binary, this means that we can reduce any computable problem to a language decision.

DFAs aren't capable of computing *every* computable problem, but they do work for a subset thereof. We can formalize DFA computation by saying that 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. $r_n \in F$

4 Regular Languages

Now that we understand how DFAs work, we can introduce our first language class. We say that a language is *regular* if and only if there exists a DFA that recognizes it.

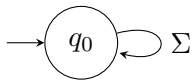
Proving that a language is regular is then as simple as constructing a DFA that recognizes it. But it's important to note that failing to construct a DFA that recognizes does *not* prove irregularity of a language. To do that, we'd need to somehow prove that it's actually impossible for such a DFA to exist.

We'll provide a method of proving irregularity of a language later on in the term, along with a better definition of what actually makes the class of regular languages significant. But for now, the important thing is that we can prove the regularity of any language for which we can construct a DFA.

4.1 Quick Proofs

4.1.1 $\emptyset$ is a Regular Language

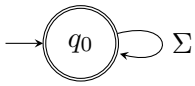
The empty set is a regular language, since we can construct the following DFA:



As shorthand, we've simply written Σ on the transition so that this DFA is alphabet-agnostic.

4.1.2 Σ^* is a Regular Language

By the same logic, we can prove that Σ^* is regular by constructing a very similar DFA:



5 Proof by Construction

With this in mind, we can introduce a new proof strategy called proof by construction. Thanks to our definition of a regular language, proving the regularity of a single language is a pretty straightforward conditional proof: we simply build a DFA that recognizes that language.

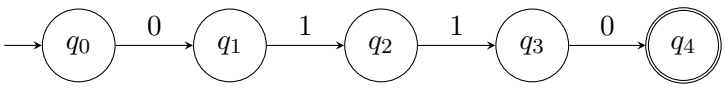
Constructive proofs allow us to generalize this to prove the regularity of whole sets of languages, even if those sets have infinite cardinality. To do this, we simply describe a method or algorithm that one could follow to create a DFA for any language in the set. Obviously, this only works for a set of languages that are closely related enough for such an algorithm to exist.

5.1 Example: Proof that a Singleton Language is Regular

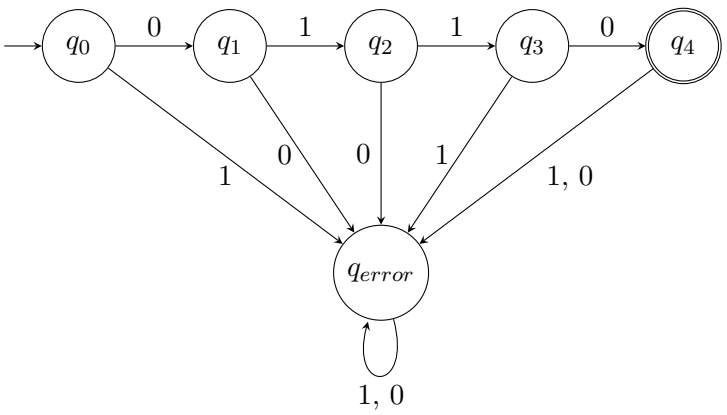
As an example of how constructive proofs are formulated, we'll prove that any language that contains only a single string is regular. We'll look at an example to see how it works, and then we'll formalize this process so that we can make a general constructive proof.

5.1.1 Example Walkthrough

Consider the language $L = \{0110\}$. The start of drawing a DFA for L is pretty intuitive: we simply lay out a transition for each character, with states between them as appropriate.



This isn't a correctly specified DFA, though, since we haven't defined a transition for every character from every state. Since this language only contains one string, though, any other transition results in an unsalvageable error. We can add an error state to catch this.



5.1.2 Proof

Let $L = \{s\}$ be a language containing only one string, s. We can then construct a DFA that decides L using the following method.

Consider s as a sequence of characters, $(a_1, a_2, \ \ldots, a_n)$. Then:

- $Q = \{q_0\} \cup \{q_1, q_2, \ \ldots, q_n\} \cup \{q_{error}\}$
- $\Sigma = \{x \mid x \text{ appears in } s\}$
- $q_0 = q_0$
- $F = \{q_n\}$

We can define δ as the following function, which takes a pair (q, x) such that $q \in Q$ and $x \in \Sigma$.

- If $q = q_i$ for some $i \in [0, n)$, then:
  - If $x = a_{i+1}$, transition to q_{i+1}
  - Otherwise, transition to q_{error}
- If $q = q_n$, transition to q_{error}, regardless of the value of x
- If $q = q_{error}$, loop back to q_{error}, regardless of the value of x

Since we have constructed a DFA that decides L, L is a regular language.

6 Closure Properties

To write the proof that's the last worksheet exercise, we should take a moment and discuss closure properties of a set. A set is *closed* under some operation if performing that operation on any element of the set yields another element of the set — that is to say, you can never escape the set by performing that operation on one of its elements. For instance:

- $\mathbb{N}$ is closed under addition and multiplication, but not subtraction or division
- $\mathbb{N}$ is closed under addition, multiplication, and subtraction, but not division

- $\mathbb{Q}$ and $\mathbb{R}$ are closed under addition, multiplication, subtraction, and division, because:
  - We can add and subtract fractions and ensure that the result is always a fraction by finding a common denominator
  - Multiplying fractions is as easy as taking the product of the numerators over the product of the denominators, which also yields a fraction
  - We can divide fractions by multiplying by a reciprocal

6.1 Worksheet Exercise: Prove that Regular Languages are Closed under Complement

At this point, we'll use a constructive proof to prove that the regular languages are closed under the complement operation. This means that the complement of a regular language is always regular.

We'll be proving more closure properties of regular languages next week, but only after we learn some new techniques that make these proofs much easier.