CSE491/596, Fall 2023: First Lecture 8/28/23.

[Go over syllabus---discuss differences: --Undecidability, **++Quantum**]
[Give out "Survey"]

Try out three ways of teaching: 1. Presentation 2. Overheads, 3. ~~Whiteboard~~

Presentation---let's jump right in to the definition of DFAs even before you read it in text notes. (Well, the main knock against it is that it can feel like lecturing by reading from a textbook...)

We will give the dry formal definition before trying to liven it up in a few ways. Note a few cosmetic differences from the text and other sources. An **alphabet** is a set of characters, which can be strung together to make **strings** over that alphabet. The **binary alphabet** $\{0, 1\}$ will serve most purposes in this course; some example strings are 0110101000101, 1111, "0" and "1" by themselves, and the **empty string** ϵ. (lowercase Greek epsilon)

A **deterministic finite automaton** (**DFA**) is a 5-tuple $M = (Q, \Sigma, \delta, s, F)$ where:

- Q is a finite set of *states*.
- Σ is a finite *alphabet*.
- s, a member of Q, is the *start state*. [Most texts say q_0.]
- F, a subset of Q, is the set of *desired final states*, also called *accepting states*.
- δ is a function from $Q \times \Sigma$ to Q.

```
class DFA {
   set<State> Q;
   set<char> Sigma;
   State s;                          //start state
   set<State> F;                     //accepting states
   //State delta(State p, char c);   //is this sensible?
   set<triple<State, char, State> > delta;
}
```

Indeed, in the *Turing Kit* software---written in Java by Mark Grimaldi while a student in this course in 1997---there is such a class. One change needed in "delta", however, motivates ways in which C# and Scala (among others) veered off from the original Java. As `delta`, it is a *class method* which makes it the same function for every DFA instance. It needs to be an *instance method*. In C++, one could do this "primitively" by making a pointer-to-member function field:

```
   State (*delta)(State p, char c);
```

Or, more cleanly (but also more fussily), one can define a separate *function-object* class, say `Delta`, with a method `apply(State p, char c)`, and have `Delta delta`; be the class field. However, I

will favor a third way that will harmonize better with next week's definition of NFAs and that reflects the idea of a program being a set of *instructions.* The abstract fact is that every function f can be identified with the set of ordered pairs (a, b) such that $f(a) = b$. The `delta` function in this case has two arguments, so we get ordered triples instead of pairs. We can just treat these triples as instance data by writing:

```
set<triple<State, char, State> > delta;
```

Every DFA instance will then automatically have its own set. Thus I prefer the definition of DFA to specify:

- δ, the set of *instructions*, aka. *tuples*, is a subset of $(Q \times \Sigma) \times Q$.
- In a DFA, for every $p \in Q$ and $c \in \Sigma$, there is a unique $q \in Q$ such that $(p, c, q) \in \delta$.

Relaxing the last clause will define an NFA ("without ϵ-arcs"). Another reason to think of instructions is how the machines look graphically:

There is a nice web applet for drawing DFAs, http://madebyevan.com/fsm/ by Evan Wallace, but it does not execute the machines you draw. That's where the *Turing Kit* comes in.

Here are pictures of two DFAs, which will also make the segue to using the overhead projector: