CSE396 Lecture Tue. 1/27: Deterministic Finite Automata

We will give the dry formal definition before trying to liven it up in a few ways. Note I will have a few cosmetic differences from the text.

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*. [Text says 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. Arguments to δ are pairs (q, c). Outputs are states r.

This "tuple" style of definition was introduced in the 1930s by French mathematicians writing under the fictional name Nicolas Bourbaki. A textbook by John Martin which we used before Mike Sipser's text came out made a joke that if you readily understand definitions like that, you muct be a mathematician. What I think it means, however, is that the Bourbakists were trying to do object-oriented programming before computers were invented. We can render the definition as:

```
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?
}
```

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 things stand above with `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);
```

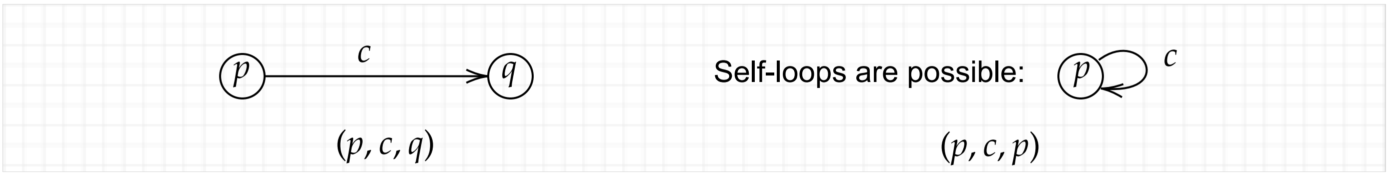
Or, more cleanly (but also more fussily), one can define a separate *function-object* class, say `Delta`, with a method `apply(State p, char c)`, and have `Delta delta`; be the class field. However, I will favor a third way that harmonizes better with the upcoming definition of NFAs and reflects the 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 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:

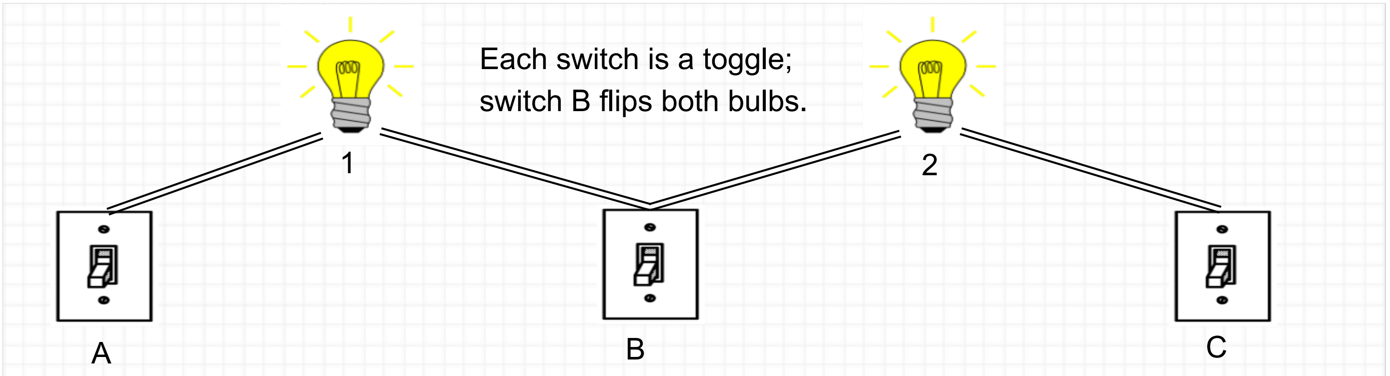


There is a nice web applet for drawing DFAs, http://madebyevan.com/fsm/ by Evan Wallace, but it does not execute the machines you draw. That's where the *Turing Kit* comes in.

Before we go to the demo, one further remark about design principles. The definition says Q is a set of "states" without saying what those are, and my Java/C++ mockup code left `State` undefined. The text first exemplifies states as being observable conditions of a machine (an automatic door), but we will often want to think in terms of internal "states of mind" while processing a stream of data. Much more than any text I know, I want every "state" to have a comment or name signifying its purpose, much like commenting a line of code. The student who programmed the *Turing Kit* agreed wholeheartedly---its most overt difference from other machine apps one can find is the rich naming and tagging facility.

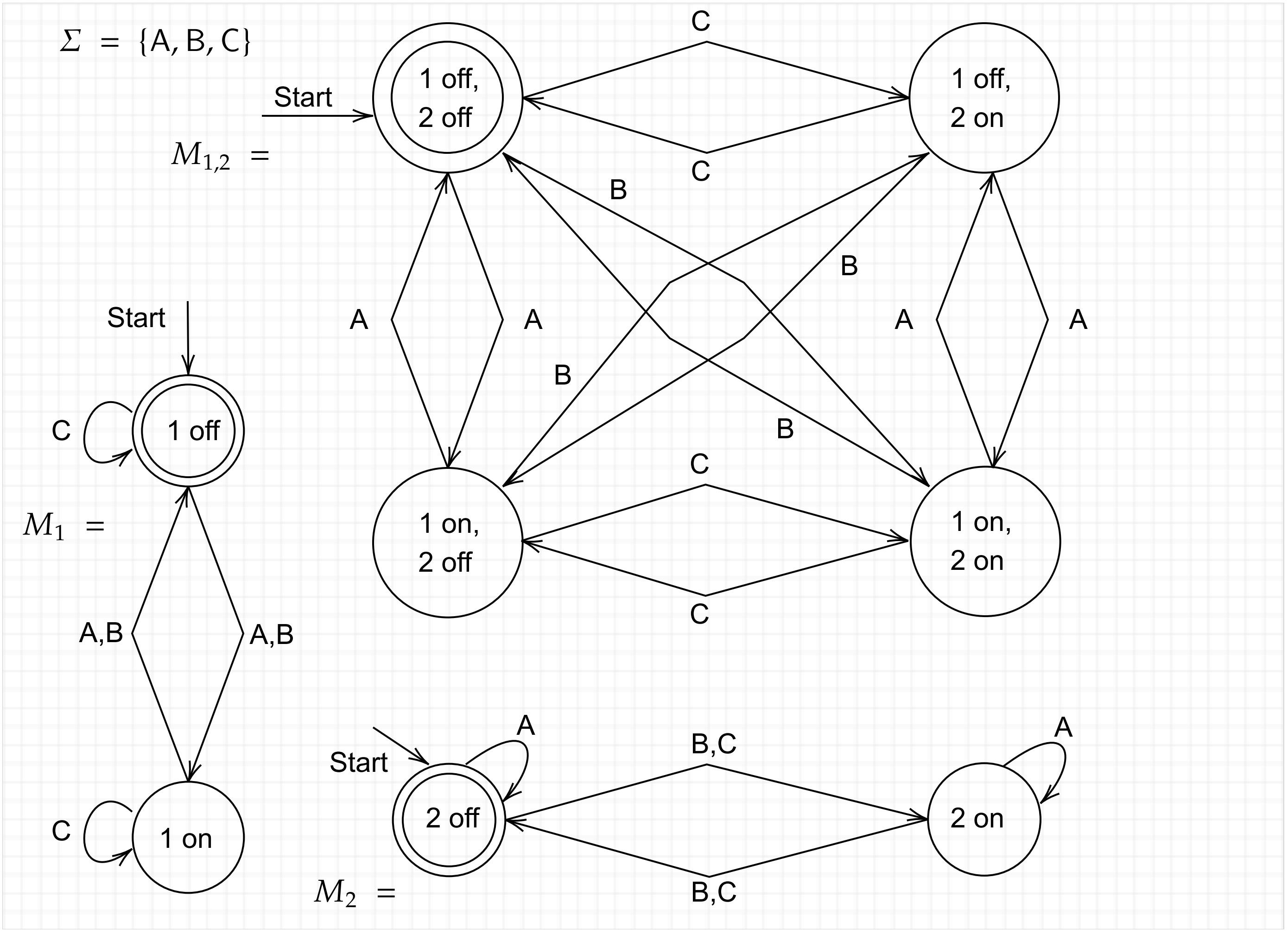
Examples of DFAs

(Light bulbs and switches from http://clipart-library.com/free/light-bulb-clipart-transparent.html)



We want L to be the set of streams of actions over the day that, assuming both lights were initially off, leave them both off. What should be the alphabet for these actions?

What should the states of the system be? Include 8 switch combos or just the lights?



The DFA M_{12} is the "Cartesian Product for AND" of the DFA M_1 tracking light bulb 1 and the DFA M_2 tracking light bulb 2. The set of ordered pairs {(1off,2off), (1off,2on),(1on,2off),(1on,2on)} is the ordinary Cartesian product of the set {1off,1on} of states of bulb 1 and the set {2off,2on} for bulb 2.

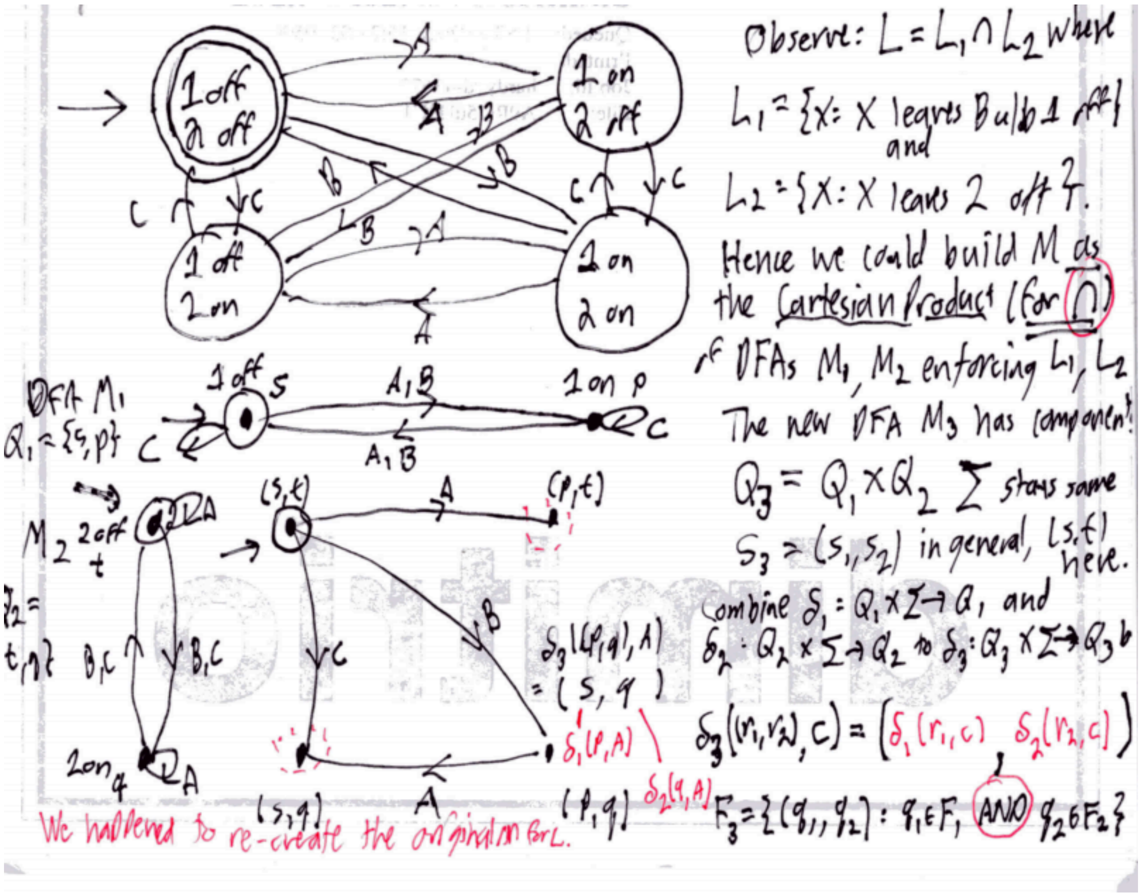
[Cartesian products come later in the text after regular expressions are defined, and more will be said about them either in lecture or week-3 recitations---including notes at the end here.]

[Here the *Turing Kit* demo worked on the first try. I used it for the rest of the lecture---including a preview example of a Turing machinr. On a Windows PC it now seems that no setup is required at all: just unzip the **`.jar`** file linked from the course webpage (no need to bother with the setup instructions link now), navigate your comamnd line to it, and enter the one command

```
java -cp TKIT70.jar Main
```

Then load `DragonSL.tmt` to see the first demo machine and do `View → Auto Resize` to work around a window-sizing bug. Use of the Turing Kit software is strictly optional---it even supports printing machine diagrams natively, but its Postscript job handling has been wonky in the past. You can of course just take a snip or screenshot of any machine you design for HW and just paste it in to the file for the rest of your assignment.]

Added for Recitation:
How do we get M_{12} from M_1 and M_2? Old notes with the general formula: A goes across and C down now, and we'll call the new machine M_3:



Observe: $L = L_1 \cap L_2$ where $L_1 = \{x : x$ leaves Bulb 1 off$\}$ and $L_2 = \{x : x$ leaves 2 off$\}$.
Hence we could build M as the Cartesian Product (for $\cap$) of DFAs M_1, M_2 enforcing L_1, L_2.
The new DFA M_3 has components:
$Q_3 = Q_1 \times Q_2$ Σ stays same
$S_3 = (s_1, s_2)$ in general, (s,t) here.
Combine $\delta_1 : Q_1 \times \Sigma \to Q_1$ and $\delta_2 : Q_2 \times \Sigma \to Q_2$ to $\delta_3 : Q_3 \times \Sigma \to Q_3$ b
$\delta_3((r_1, r_2), c) = (\delta_1(r_1, c), \delta_2(r_2, c))$
$F_3 = \{(q_1, q_2) : q_1 \in F_1$ AND $q_2 \in F_2\}$

The new machine starts up in the state (1off,2off). If switch A is flipped, it goes to the state

δ_3((1off,2off),A) = (δ_1(1off,A),δ_2(2off,A)) = (1on,2off)
δ_3((1on,2off),B) = (δ_1(1on,B),δ_2(2off,B)) = (1off,2on)
δ_3((1off,2on),C) = (δ_1(1off,C),δ_2(2on,C)) = (1off,2off)

Because the operation is AND, the composite final states are $\{(q_1, q_2) : q_1 \in F_1 \; AND \; q_2 \in F_2\}$.
[This equals $F_1 \times F_2$, but this only holds good for AND.]

Suppose the desired final condition was that exactly one of the two lights be left on, with the idea that it would be safe to leave the building in the evening and the custodian could flick all the lights off. Then we have $F_3' = \{(q_1, q_2): q_1 \in F_1\ XOR\ q_2 \in F_2\}$: XOR because we want exactly one light to be off, the other on. If we call the new machine M_3', then we have

$$L(M_3') = \left\{x \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}^* : x \in L(M_1)\ XOR\ x \in L(M_2)\right\} = L(M_1) \triangle L(M_2)$$

$$= (L(M_1) \setminus L(M_2)) \cup (L(M_2) \setminus L(M_1)).$$