CSE396 Lecture Thu. 2/11: NFAs and Regular Expressions

The formal definition of a *finite automaton* is a 5-tuple (i.e., an object) $N = (Q, \Sigma, \delta, s, F)$ where:

- Q is a finite set of *states*
- Σ is the *input alphabe*t
- s, a member of Q, is the *start state* (also called q_0)
- F, a subset of Q, is the set of *accepting* states (also called *final* states)
- δ is a finite set of *instructions* (also called *transitions*) of the form (p, c, q) where $p, q \in Q$ and $c \in \Sigma$; an **NFA with ϵ-transitions** (**NFA$_\epsilon$**) also allows (p, ϵ, q).

The machine is *deterministic* (is a DFA) if $(\forall p \in Q)(\forall c \in \Sigma)(\exists! q \in Q) : (p, c, q) \in \delta$. Else it is "properly" *nondeterministic* (is properly an NFA).

So DFA is a special case of an NFA. When we have a DFA M, we can regard δ as a function from $Q \times \Sigma$ to Q. With an NFA, we could regard δ as a function from $Q \times \Sigma$ to 2^Q, which is the set of all subsets of Q and called the *power set* of Q. But in all cases I prefer to think of δ as a set of instructions.

I have actually not yet formally defined the language of a machine; we appealed first to intuition. We can give a rigorous definition that applies equally well to NFAs and DFAs. This introduces a notion that is not defined as-such in the text but will be especially handy when we come to "GNFAs" later on. (**Purple** indicates definitions that are not standard nomenclature.)

Definition 1: Say that and NFA (or DFA) N ***can process*** a string x ***from*** state p ***to*** state q if there is a sequence of instructions

$$(p, u_1, q_1)(q_1, u_2, q_2)(q_2, u_3, q_3) \cdots (q_{m-2}, u_{m-1}, q_{m-1})(q_{m-1}, u_m, q)$$

such that $u_1 u_2 \cdots u_m = x$. Since we haven't exemplified ϵ-transitions yet, you can think of each u_i being a character, so that the length m of the computation is the same as the length of x (which we usually call $n = |x|$); later with GNFAs, each u_i may also be a longer substring. The sequence itself is called a **computation** or **computation path**.

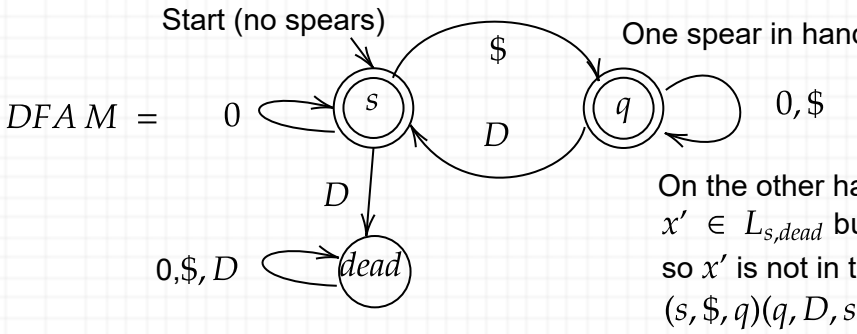
Then we write $x \in L_{p,q}$ (with N understood). Now formally define:

$$L(N) = \cup_{f \in F} L_{s,f}.$$

If N has only one accepting state f (a design goal we can meet for NFAs but often not for DFAs) then the language is just $L_{s,f}$. An example of a DFA that needs to have two accepting states is the "spears and dragons" game that was shown in last Thursday's demo.

An accepting computation on input $x = \$0D$ is $(s,\$,q)(q,0,q)(q,D,s)$.
Thus $x \in L_{s,s}$ and since the start state is accepting, $x \in L(M)$.

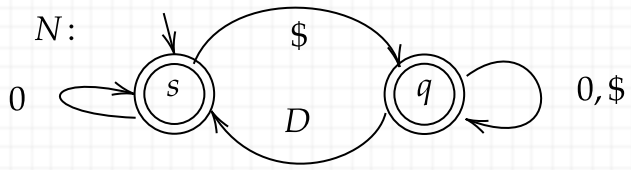
$F = \{s,q\}$



On the other hand, consider $x' = \$DD$
$x' \in L_{s,dead}$ but $dead$ is not accepting
so x' is not in the language.
$(s,\$,q)(q,D,s)(s,D,dead)$

Without the dead state and arc to it, the NFA N on input $x = \$DD$ would "crash" in state s. Even though s is an accepting state (and even though this would count as legal termination by a Turing machine), not all of x would be processed, so it does not count in the FA's language. With the dead state present, x gets processed to $dead$, but $dead \notin F$ so $x \notin L(N)$ still.

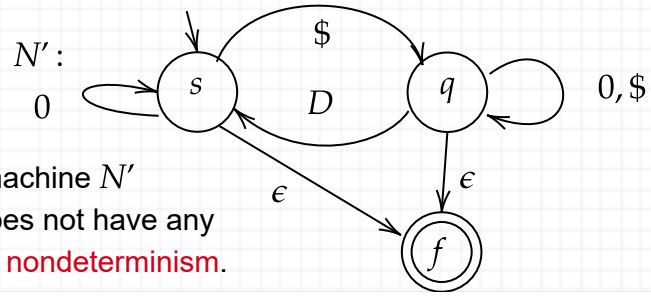
M without the dead state is technically an NFA N. The string $x' = \$DD$ **cannot be processed.**



This also means that for an NFA, having all states in F does not mean that the language is all strings.

$(s,\$,q)(q,D,s)crash!$ "Crashing" in an accepting state is not accepting the string.

If you add ϵ-arcs to make a single accepting state, it is also technically an NFA:



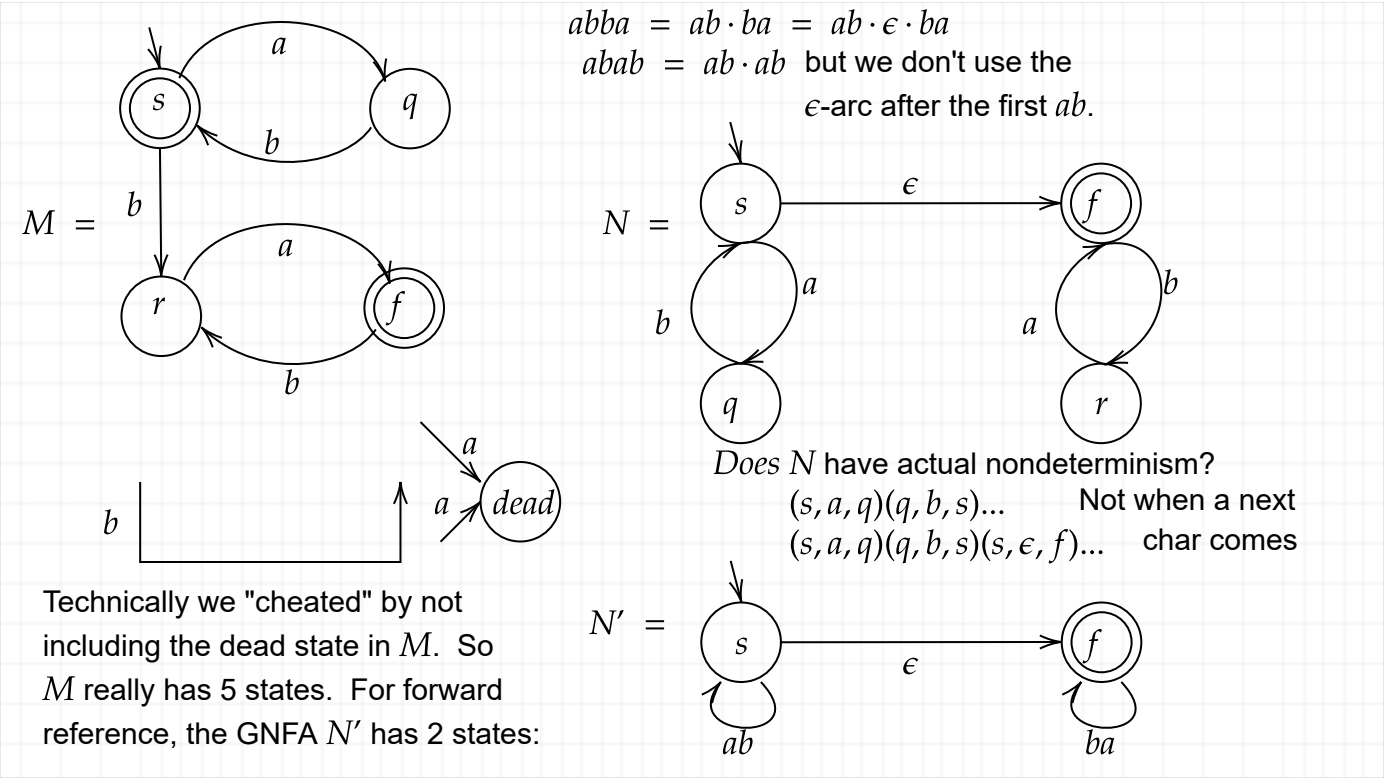
The machine N' still does not have any actual nondeterminism.

The above accepting computation on the string $x = \$0D$ now technically becomes
$(s,\$,q)(q,0,q)(q,D,s)(s,\epsilon,f)$
The language of N' is just $L_{s,f}$

So what are NFAs good for? The very end of Tuesday's lecture gave a motivation of reducing the number of states. But even when the NFA has the same number of states, it can be argued as being conceptually clearer. Here is an example. Consider the language of strings over $\Sigma = \{a,b\}$ that begin by repeating ab zero or more times and then repeat ba zero or more times (without being allowed to do more of ab after that). Examples: $ababbaba$ is in the language. But $abbaab$ is not, because of the last ab. The string $abab$ by itself is OK, because the "zero option" is allowed for the ba part. Likewise, $baba$

uses zero-option for the first part. But $baab$ is not allowed, because it gets the parts in the wrong order. And how about ϵ? It is in because the "zero option" is allowed for both parts.

As a first example of a **regular expression** that does some grouping, this language can be denoted by $(ab)^*(ba)^*$. Here are a DFA and an NFA:



Here's a purely "philosophical" argument---well, object oriented design philosophy: Between "DFA" and "NFA", which is the more basic concept? The notion of a DFA is simpler, one might say. But consider how we might program them in an O-O language such as C++. Which should be the base class?

```
class ??? {
   set<State> Q;
   set<char> Sigma;
   State s;
   set<State> F;
   set<triple<State,char,State> > delta;
};
```

My position: NFA should be the base class, because a DFA "Is-A" NFA. In this O-O sense, "NFA" is the more basic concept.
[Insert discussion of the "`Square` Is-A `Rectangle`?" dilemma, but maintain that `const Square` definitely Is-A `const Rectangle`.]

The most instrumental reason to use NFAs, however, is their relationship to *regular expressions*---which the next lecture will try to convince you is "electric." Let's first give an informal definition of regular expression, before the formal one to come then.

Definition: A regular expression (with powering allowed as an abbreviation) has the same syntax as an ordinary algebraic expression in $+$, $\cdot$, and powers, where the powers can be either numeric or *. The differences are:

- In place of zero, we have the constant $\emptyset$, which denotes the empty language $\emptyset$.
- In place of one, we have the constant ϵ, which denotes the language $\{\epsilon\}$ whose only member is the empty string.
- In place of constants, we have the letters of the alphabet Σ.
- The operation $+$ means union (the text uses $\cup$).
- The operation $\cdot$ means concatenation of languages, and numerical powers iterate that.
- The star power *, which is named "Kleene star" after Stephen Kleene, is the union of all the numerical powers and means "zero or more" occurrences of what is powered. That is:
  $A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots = \{\epsilon\} \cup A \cup A \cdot A \cup A \cdot A \cdot A \cup \cdots$

Extra (actually, continuing an example from the end of my first course lecture):
In abstract math, B^A denotes the set of functions $f : A \to B$, and by rule, $|B^A| = |B|^{|A|}$ which is just a power of numbers. So 0^0 equals the cardinality of the set of functions from $\emptyset$ to $\emptyset$. Now:

The empty function $\emptyset$ is a function from $\emptyset$ to $\emptyset$.

And it is the only function from $\emptyset$ to $\emptyset$. This is "Zen" but it is real. So $0^0 = 1$. Since $\{\epsilon\}$ is like 1, this can be argued to justify $\emptyset^0 = \{\epsilon\}$. But I will try a third way to make it intuitive. First, let's finally get around to defining the (*Kleene*) *star* operation, named for Stephen Kleene (1909--1994):

$$A^* = \bigcup_{i=0}^{\infty} A^i = \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots$$

It is the set of all strings formed by concatenating zero or more strings from A. Now here is the intuition for why "concatenating zero strings from A" yields ϵ, i.e., why A^* always includes ϵ even when $A = \emptyset$.

Suppose we've designed a security system for a building that periodically runs a status check, say if it detects the possibility of there being an intruder or some other breakdown. The system gets feedback for the check from various cameras and sensors and monitors. Let A be the language of strings representing internal audits of sensory data that pass the status check. Since the check can run multiple times, we can picture it being inside an event-driven `while` loop. Then A^* is the language of

inputs that will pass every check, no matter how many times the check is activated. So, finally, what happens if:

- $A = \varnothing$, meaning we are sure to **fail** the check if it is activated; **but**
- the check is never activated---the while loop runs **0** times and falls through!

The upshot is that the system **passes**, with the **empty string** of sensor data. Because it is a *pass*, not a *fail*, the language of inputs that pass "every" check (of 0 checks) is $\{\epsilon\}$, not $\varnothing$. So $\varnothing^0 = \{\epsilon\}$.

[See my blog article https://rjlipton.wordpress.com/2015/02/23/the-right-stuff-of-emptiness/ for a story with my father that I did not have time to tell, which gives other intuition for why $\{\epsilon\}$ versus $\varnothing$ can literally be a life-or-death difference. The continuation of that article gets more complicated, however.]