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

[I went over academic integrity rules and guidelines on homeworks. The main guideline is that study groups are OK for understanding lectures and the text, but must stop short of trying to work out the problems ("red zone"). Any Qs in that zone should go to me or the TA. Then I showed how AI can now solve the TopHat problems---well, with one error---so I specifically barred using Gemini or etc. on those.]

Lectures so far have featured "Type Discipline", as exemplified by saying that instructions (p, c, q) have type $Q \times \Sigma \times Q$, or in C++ terms, type `triple<State,char,State>`. Now we will upset this uniformity.

Definition 1: An **NFA with ϵ-transitions** also allows instructions of the form (p, ϵ, q). They enable the machine to go from state p to state q without **processing** a character.

In the Sipser text, this is included in the basic definition of NFA. But IMHO it is useful to keep the concepts separate and use the abbreviation "**NFA$_\epsilon$**" when ϵ-transitions are allowed. They break the type discipline because ϵ is a `string` not a `char`. Sipser continues treating δ as a function rather than a set, giving it range 2^Q, the **power set** of Q, because both $\delta(p, c)$ and $\delta(p, \epsilon)$ can be sets of more than one possible next state---or $\emptyset$ when there is no possible next state. *Teacherly advice*: stick with the set-of-code-triple form. This form is especially nice for the following definition, which applies to DFAs, NFAs, and NFA$_\epsilon$s all in one shot: (**Purple** indicates definitions that are not in the text and not standard nomenclature.)

Definition 2: Say that a finite automaton N ***can process*** a string x ***from*** state p ***to*** state q if there is a sequence of instructions

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

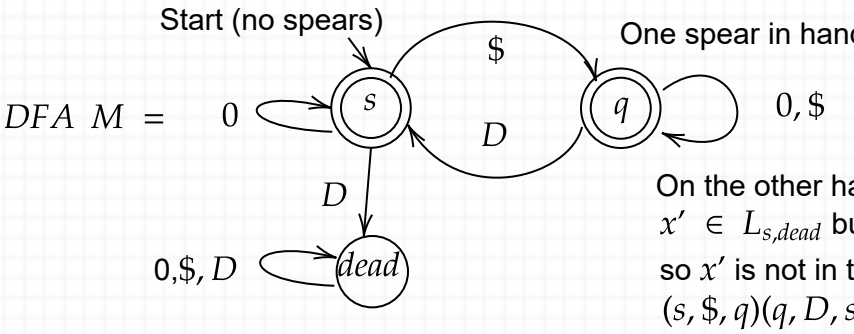
such that $u_1 u_2 \cdots u_m = x$. Here m is at least the length $n = |x|$ of x but can be greater if some of the u_i components are ϵ. Then we write $x \in L_{p,q}$ (with N understood) and formally define:

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

That is, the **language** $L(N)$ of the automaton N is the set of strings x such that N can process x from its start state to some accepting state. The sequence of triples themselves (note that the ends "match like dominoes") is called a **computation** or **computation path**.

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 week's demo.

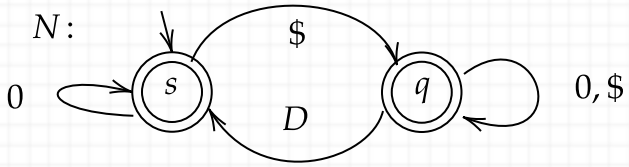
An **accepting computation** on input $x = \$0D$ is $(s,\$,q)(q,0,q)(q,D,s)$.
Thus $x \in L_{s,s}$ and since the start state is accepting, $x \in L(M)$.



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.

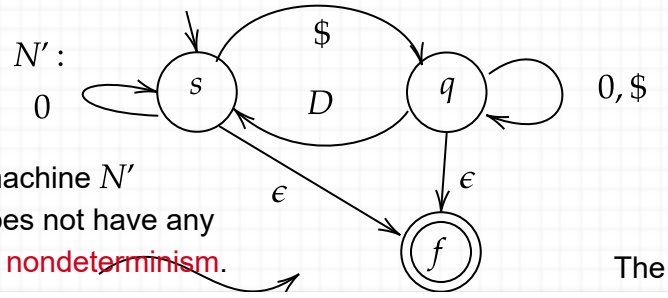
M without the dead state is technically an NFA N. Then string $x' = \$DD$ **cannot be processed.**



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

$(s,\$,q)(q,D,s)(s,D = crash!$ "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 $L_{s,f}$

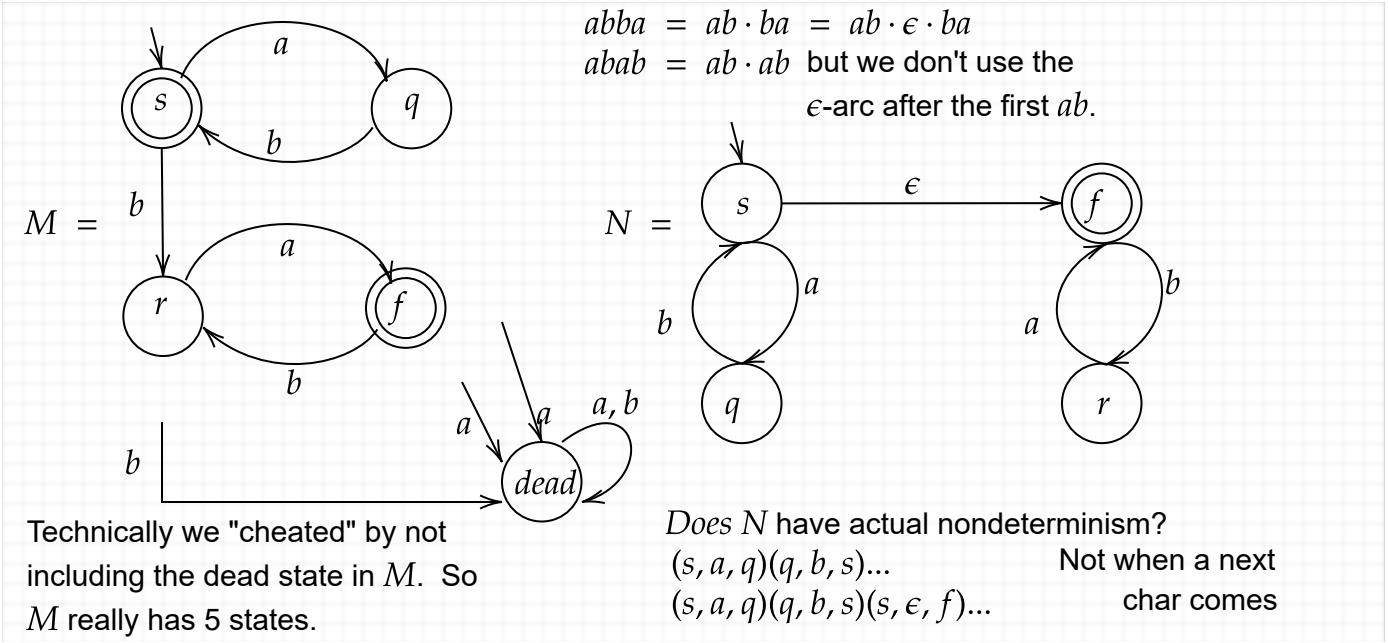
The string $\$DD$ still cannot be processed.

So what are NFAs good for? The very end of last Thursday's third lecture gave a motivation of economizing on the number of states. But even when the NFA has the same number of states, it can be argued as being conceptually clearer.

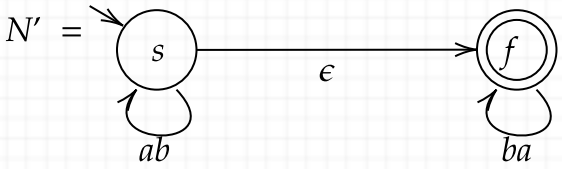
Here is an example. Consider the language of strings over $\Sigma = \{a,b\}$ that begin by repeating ***ab*** zero or more times and then repeat ***ba*** zero or more times (without being allowed to do more of ab after that). Examples: $ababbaba$ is in the language. But $abbaab$ is not, because of the 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 $(ab)^* \cdot (ba)^*$. Here are a DFA and an NFA:



The most economical diagram is at right. The triples (s, ab, s) and (f, ba, f) technically have regular expressions as middle component. We will see these **Generalized NFAs** next week.



Between "NFA" and "DFA", which is the more basic, fundamental concept? The notion of DFA seems simpler, so you might go with that. But let's morph this question into one of object-oriented design philosophy: between `class DFA` and `class NFA`, 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. [Maybe insert discussion of the "`Square` Is-A `Rectangle`?" dilemma, but maintain that `const Square` definitely Is-A `const Rectangle`.]

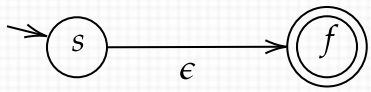
The most instrumental reason to use NFAs, however, is their relationship to **regular expressions.** Between now and the next lecture, we will try to convince you this relationship is "electric" in a literal sense. We've already introduced all the ingredients informally, so let's dive right in to a **formal inductive definition** of them. Intertwined will be an **inductive proof** of the *theorem* that for every **regexp** α we can build an NFA$_\epsilon$ N such that $L(N) =$ the language $L(\alpha)$ denoted by α.

"Defineorem": Regular Expressions and Their Corresponding NFAs (with ϵ-transitions):

(B1) $\varnothing$ is a regexp; $L(\varnothing) = \varnothing$; $N_\varnothing =$

$(\delta = \varnothing)$

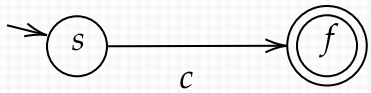
(B2) ϵ is a regexp; $L(\epsilon) = \{\epsilon\}$; $N_\epsilon =$



δ has (s, ϵ, f)

For all chars $c \in \Sigma$:

(B3) c is a regexp; $L(c) = \{c\}$; $N_c =$



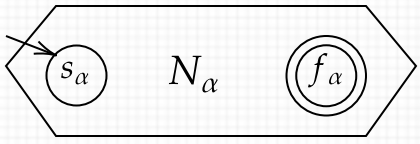
δ has (s, c, f)

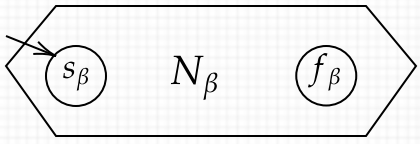
This completes the *basis* of an *inductive definition* of regular expressions. Now let α and β be any two regular expressions, with languages $A = L(\alpha)$ and $B = L(\beta)$. By *inductive hypothesis* (**IH**) we have NFAs N_α and N_β such that $L(N_\alpha) = A$ and $L(N_\beta) = B$. Then:

(I1) $\gamma = \alpha \cup \beta$ is a regexp; $L(\gamma) = A \cup B$.

gamma alpha beta

Now to complete the *induction case* (I1) we need to show how to build an NFA$_\epsilon$ N_γ such that $L(N_\gamma) = L(\gamma)$. What we have to work with is (are) N_α and N_β. We know they have start states we can call s_α and s_β. Taking a cue from the base case NFAs, and mainly for convenience, we may suppose they have unique accepting states f_α and f_β. Besides that, we make no assumptions about their internal structure, so we draw them as "blobs":





The goal is to connect them together to make N_γ with needed properties, also for the cases:

(I2) $\gamma = \alpha \cdot \beta$ is a regexp; $L(\gamma) = A \cdot B$.

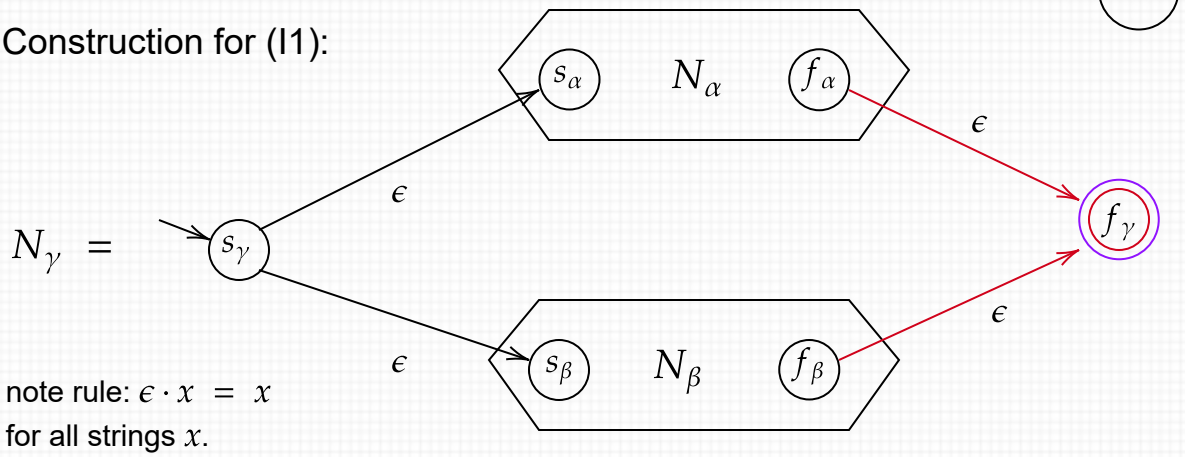
(I3) $\gamma = \alpha^*$ is a regexp; $L(\gamma) = A^*$. (In I3 we have only N_α given.)

$$\textit{This does}: L(N_\gamma) = L(N_\alpha) \cup L(N_\beta)$$

$$\textit{Target}: L(\gamma) = L(\alpha) \cup L(\beta)$$

$$L(\alpha) \cup L(\beta)$$

Construction for (I1):



note rule: $\epsilon \cdot x = x$
for all strings x.

This builds N_γ, but we still need to prove it is correct, i.e., $L(N_\gamma) = L(\gamma)$. *Note the rhythm:*

1. $L(N_\gamma) = L(N_\alpha) \cup L(N_\beta)$ by machine construction;
2. $L(N_\alpha) = L(\alpha)$ and $L(N_\beta) = L(\beta)$ by inductive hypothesis;
3. Thus $L(N_\gamma) = L(\alpha) \cup L(\beta) = L(\alpha \cup \beta) = L(\gamma)$ by definition of γ.

[I will continue as time permits by copy-and-paste and moving things around to do the other two inductive cases to complete the proof. ... As it happened, time ended here. I used the chalkboard for this $\cup$ case but wrote $\gamma = \alpha + \beta$ instead. I put all the following up before drwing the picture:

- The definition $L(\gamma) = L(\alpha) \cup L(\beta)$.
- The **inductuive hypothesis** (**IH**) of their being NFA$_\epsilon$s N_α and N_β for the taking such that $L(N_\alpha) = L(\alpha)$ and $L(N_\beta) = L(\beta)$.
- The goal of building N_γ such that $L(N_\gamma) = L(\gamma)$ needed to complete the **induction step**.
- How you can deduce the goal after building N_γ such that $L(N_\gamma) = L(N_\alpha) \cup L(N_\beta)$.

The statement $L(N_\gamma) = L(N_\alpha) \cup L(N_\beta)$ is what you get from the diagram. The next lecture will pick up with the concatenation and star cases.]