1 Lecture summary

- We showed that even a computational model as powerful as Python has languages it cannot compute
- We discussed inductive definitions and defined delta-hat
- We talked about the idea that when constructing automata, it is helpful to write down a fact for every state. This helps you come up with the machine and also prove it is correct.

2 Noncomputability

- We defined a spec as a language, i.e. a set of strings
- We said that a program satisfies a spec L if it outputs "yes" on every string in the language L and halts and outputs "no" on any string not in L

  (Technical Note: this is a slightly different definition of "satisfying a spec" than what I gave in class; in 4810 you would learn that this definition is "decides" while the other definition is "recognizes"; the halting problem given below is recognizable but not decidable; all of this is outside the scope of this course, but feel free to ask if curious)
- Claim: there are specs that don't have programs that satisfy them.
- Proof: the set of specs is uncountable, but the set of programs is a subset of the set of strings, which is countable. Therefore there cannot be a surjection from the set of programs to the set of strings.
- Example of non-computable spec: "halting problem"

$$HP = \{x \mid \text{if interpreted as a python program, } x \text{ doesn't run forever}\}$$

- Idea behind proof that HP is noncomputable: if HP was computable, there would be a program P that decides it. Is the following program in HP?

```
def diabolical(x):
  if P(diabolical) == "yes":
    while True: pass # (run forever)
  else:
    print "no"
```

3 Inductive definitions

We can define the set of strings Σ^* as follows:

1. the empty string ϵ is in Σ^*
2. if x is in Σ^* and a is in Σ then $xa \in \Sigma^*$
3. strings formed using rules 1 and 2 are the only strings in Σ^*

This formalizes the idea that there are two kinds of strings in the world: empty strings, and strings of the form xa.

Using this idea, we can define a function f inductively by specifying its output on the two kinds of strings. While defining $f(xa)$, we can assume we've already defined $f(x)$. In this sense these definitions are inductive or recursive.

4 Definition of delta-hat

The transition function for a DFA is a function

$$\delta : Q \times \Sigma \to Q$$

This means it only tells us how to process a single character. It is useful to define an extended transition function that processes a whole string. Just as $\delta(q, a)$ tells where the machine transitions from state q on input a, $\hat{\delta}(q, x)$ tells us where the machine transitions to after processing the whole string x starting from state q. It is defined inductively/recursively as follows:

$$\begin{aligned}
&\hat{\delta} : Q \times \Sigma * \to Q \\
&\hat{\delta} : (q, \epsilon) \mapsto q \\
&\hat{\delta} : (q, xa) \mapsto \delta(\delta - hat(q, x), a)
\end{aligned}$$

5 Associating facts with states

When designing a DFA, it is useful to write down a fact that you know about the input if processing that input lands you in that state.

Example 1: networking. In the initial state I know that there is no connection. After receiving a "CONNECT" message, I know that someone wants to open a connection. After receiving an "ACKNOWLEGEMENT" message I then know that the session is established. A "DATA" message keeps me in the same state, and a CLOSE message returns me to the initial state.

Example 2: Suppose we wanted to create a machine to recognize binary strings that are multiples of 3. The important facts about binary for this example are:

- the empty string represents 0
- if x represents n, then $x0$ represents $2n$
- if x represents n, then $x1$ represents $2n + 1$

For example: 1101 represents 13 because 110 represents 6 and $2 \cdot 6 + 1 = 13$.

It is not clear how to build such a machine. We'd like to know that if we're in the final state, then x is a multiple of 3. So we can try creating a state q_0 that represents the fact "x is a multiple of 3".

The starting state is where we end up after processing the empty string. In this case, we know that ϵ represents 0, which is a multiple of 3, so we can start in q_0.

We're not done, because we haven't written transitions for all states on all characters. Let's consider them. What happens if $\hat{\delta}(q_0, x) = q_0$ and we see a 0? Well, since $\hat{\delta}(q, x) = q_0$, we know x represents a multiple of 3. Lets say x is $3k$. Then $x0$ represents $2x$ which is $2 \cdot (3k)$ which is itself a multiple of 3. So $\hat{\delta}(q0, x0)$ should be q_0; this means $\delta(q0, 0)$ should be 0.

If we see a 1 on the other hand, then we have $6k + 1$, so we shouldn't transition to q_0. So let's create a new state q_1. One way we might describe this state is by saying that if $\hat{\delta}(q_0, x) = q_1$ then x represents a multiple of 3 plus 1 (i.e. x represents $3k + 1$ for some k)

Continuing this process yields a machine with 3 states. I encourage you to finish it yourself.