Announcements
1. Homework 2 due today.

An Interesting Exercise
1. Last time we defined a language transformation that produces a new language consisting of the first halves of all strings of even length in another language:

   \[ L_\frac{1}{2} = \{ x \mid \text{there exists } y \text{ such that } |x| = |y| \text{ and } xy \in L \} \]

2. We developed one way to prove that \( L_\frac{1}{2} \) is regular if \( L \) is regular.
   - Suppose that \( D \) is a DFA that accepts \( A \).
   - Informally, I suggested building a new machine, \( N \), that used nondeterminism to simulate two copies of \( D \) one of which would scan the actual input to find out where \( D \) would be after reading some string that might be the first half of one of the even length inputs it would accept. The other copy would guess that state the first copy would eventually reach and then try to guess a string of the same length as the input that would move the machine from that state to a final state. If there was some way for the second simulation to “guess correctly”, then the input should be accepted.
   - Although we describe these simulations as two separate activities, to actually accomplish them in a NFA, they have to be tightly interlocked performing one step of each simulated machine in lock step with the other. This is the only way to make sure that the guessed suffix is the same length as the input.
   - Formally, the construction we imagined could be described as follows. Given \( D = (Q, \Sigma, \delta, s, F) \)

   \[
   N = (\{s_N\} \cup Q \times Q, \Sigma, \delta_N, s_N, \{(m, m, f) \mid m \in Q, f \in F\})
   \]

   where

   \[
   \delta(s_n, \epsilon) = \{(s, m, m) \mid m \in Q\}
   \]

   \[
   \delta((s_x, m, s_y), x_i) = \{\delta(s_x, x_i), m, \delta(s_y, y_i)\} \mid y_i \in \Sigma\}
   \]

3. It is worth discussing one concrete example of the application of the formal transformation from a particular \( D \) to a particular \( N \) the transformation just described would produce.
   - Consider the machine \( D \) shown below which recognizes the language of sequences of 0s and 1s containing a number of 1s that is a multiple of 3.

   ![Diagram](image.png)

   - The machine produced by our transformation would include as states all possible triples of states from the machine shown in the diagram above. Since the original machine contained three states, the states set of the non-deterministic machine produced would have more than 27 states.
   - Now, let’s consider what language we actually get when we apply the \( L_\frac{1}{2} \) transformation to the language of the machine \( D \).
     - Given any string of length greater than 1, we can calculate the number of 1s in the string mod 3. If the answer is 2 we add a single 1, if the answer is 1, we add two 1s, and if the answer is 0, we leave it alone for a moment. Then, regardless of how many 1s we just added, we add enough 0s to double the length of the original string. The result belongs to \( L(D) \).
   So, any string longer than 1 belongs to \( L(D)_\frac{1}{2} \).
– In fact, the only string that is not in $L(D)_2$ is “1”.

• It is pretty clear that it would be easy to build a machine to accept this language. One possibility is shown below.

![Diagram](image)

• This 3 state machine is clearly much simpler than the machine with 28 states our transformation produces, but the two machines recognize the same language.

4. Fortunately, we can continue thinking about $L(D)_2$ because there is another, very different approach we can use.

• Again, suppose that $D$ is a DFA that accepts $L$.

• The idea is to build an NFA $N$ that is able to walk forward as $D$ would from the start state while simultaneously walking backward in $D$ from one of the final states. For each step forward, $N$ also makes one step backward.

• $N$ has to step forward based on the input symbols it encounters, but it can guess what symbols it would have to see while walking backwards to of $A$.

• Instead of working out the details of the formal specification of this machine now, I am going to leave it to you by including it as one of the questions on our next assignment. The goal here is to give you a problem where you don’t have to struggle to find the “trick” that makes it possible to solve. I just gave you the trick. Instead, you just have to turn the idea behind the trick into a precise formal specification.

**More Fun with NFAs**

1. We can use the notion of an NFA to give some simple proofs of a few important closure properties.

• In an earlier class, I sketched how NFAs could be used to show that regular languages are closed under the product operation.

  – Then, I suggested somehow merging the final states of a machine that recognized one language with the start state of a second machine.

  – Now, we can use the magic of $\epsilon$-transitions to connect the appropriate states.

– It is worth noting that this can be formalized by saying that if $M = (Q, \Sigma, \delta, s, F)$ and $M' = (Q', \Sigma, \delta', s', F')$ are DFAs that accept $L$ and $L'$ then $LL' = L(M_P)$ where the NFA $M_P$ is defined as:

$$M_P = (Q \cup Q', \Sigma, \delta_P, s, F')$$

with

- $\delta_P(q, \epsilon) = \{s'\}$ if $q \in F$
- $\delta_P(q, x) = \{\delta(q, x)\}$ if $q \in Q$
- $\delta_P(q, x) = \{\delta'(q, x)\}$ if $q \in Q'$
• We can also use NFAs to give a simpler proof that regular languages are closed under union.
  – We can build a new machine by taking all of the states and transitions of two machines that recognize the languages we are intersecting and adding a new state that will be our start state and have epsilon-transitions to the start states of the sub-machines so that our machine can guess which of the two languages its input belongs to and take the appropriate epsilon-transition into that machine’s start state.

- Technically, we would argue that if \( M = (Q, \Sigma, \delta, s, F) \) and \( M' = (Q', \Sigma, \delta', s', F') \) are DFAs that accept \( L \) and \( L' \) then \( L \cup L' = L(M) \) where the NFA \( M \) is defined by:

\[
M_U = (Q \cup Q' \cup s_U, \Sigma, \delta_U, s_U, F \cup F')
\]

with

* \( \delta_U(s_U, \epsilon) = \{s, s' \} \)
* \( \delta_U(q, x) = \{\delta(q, x)\} \) if \( q \in Q \)
* \( \delta_U(q, x) = \{\delta'(q, x)\} \) if \( q \in Q' \)

- Regular languages are closed under the closure (star) operation.
  – Recall that when we write \( L^n \) we mean \( \{\epsilon\} \) if \( n = 0 \) and \( L^n = LL^{n-1} \) otherwise and that we define

\[
L^* = \bigcup_{n=0}^{\infty} L^n
\]

  – At first, it might seem obvious that the regular languages are closed under the closure operation (both because of its name and) because the closure is a union of products and we have already shown that regular languages are closed under union and products, but...

  – We have shown that regular languages are closed under finite unions. This does not imply they are closed under infinite unions.

  * Every language containing just a single word is regular. Therefore, if the regular languages were closed under infinite unions, we could show that any language was regular by just union-ing together all of the languages consisting of just one word from the possibly non-regular language.

  * It is also worth noting that if we think about the way we proved unions of regular languages were regular, applying the construction to an infinite set of finite automata would lead to an infinite automaton.

  – We can, however, show that the closure of a regular language is regular using a construction involving NFAs with epsilon transitions. Given a machine \( M \) that recognized the language we want the closure of, the idea is to add a new, final start state (to handle the fact that \( \epsilon \) must be in the closure), allow an epsilon-transition from this state to the original start state, and also add epsilon-transitions from all of the final states back to the original start state.
- That is, iv \( M = (Q, \Sigma, \delta, s, F) \) is a DFA that accepts \( L \) then we claim that \( L^* = L(M_C) \) where the NFA \( M_C \) is defined as:

\[
M_C = (Q \cup \{s_C\}, \Sigma, \delta_C, s_C, F \cup s_C)
\]

with

* \( \delta_C(q, \epsilon) = \{s\} \) if \( q \in F \)
* \( \delta_C(s_C, \epsilon) = \{s\} \)
* \( \delta_C(q, x) = \{\delta(q, x)\} \) if \( q \in Q \)

accepts \( L^* \)

### Regular Expressions

1. The closure properties of regular languages provide a way to describe regular languages by building them out of simpler regular languages using the operations union, product and closure.

2. The notation called regular expressions is based on this fact.

**Definition:** Given some finite alphabet \( \Sigma \), we define \( e \) to be a regular expression if \( e \) is

- \( a \) for some \( a \in \Sigma \)
- \( \emptyset \)
- \( \epsilon \)

- \( e_0 \cup e_1 \), where \( e_0 \) and \( e_1 \) are regular expressions
- \( e_0 \circ e_1 = e_0 e_1 \) where \( e_0 \) and \( e_1 \) are regular expressions
- \( e_0^* \) where \( e_0 \) is a regular expression.
- \( (e_0) \) where \( e_0 \) is a regular expression.

3. We view regular expressions as another formalism for describing languages. If \( e \) is a regular expression, the language defined by \( e \) is denoted by \( L(e) \) and defined recursively/inductively as follows:

**Base clauses**

- \( L(x) \) for some \( a \in \Sigma \) is just \( \{a\} \)
- \( L(\emptyset) \) is \( \emptyset \)
- \( L(\epsilon) \) is \( \{\epsilon\} \)

**Recursive clauses**

- \( L(e_0 \cup e_1) \) is \( L(e_0) \cup L(e_1) \)
- \( L(e_0 \circ e_1) \) is \( L(e_0)L(e_1) \)
- \( L(e_0^*) \) is \( L(e_0)^* \)
- \( L((e_0)) \) is \( L(e_0) \)

4. Given the closure properties we have just shown, it is clear that all regular expressions describe regular languages.

5. Here are some languages one might want to describe with regular expressions

- Binary strings of length 2 or less: \( (0 \cup 1 \cup \epsilon)(0 \cup 1 \cup \epsilon) \).
- Binary strings that end in 110: \( (0 \cup 1)^*110 \).
- Binary strings that don’t end in 110:
- Binary strings that are multiples of 3?

6. The last example is very challenging. It raises the question of whether or not every regular language can be described by a regular expression.