Announcements
1. Homework 3 due Friday at noon (not 5 or Saturday or Sunday).

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 $a \in \Sigma$
- $\emptyset$
- $\varepsilon$
- $e_0 \cup e_1$, 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. Given the closure properties we have just shown, it is clear that all regular expressions describe regular languages. It is also true, though far from clear, that every regular language can be described by some regular expression. Our first goal to day will be to justify this claim.

Generalize Nondeterministic Finite Automata
1. The book presents an algorithm that translates the description of a DFA into a regular expression describing the same language. The existence of and correctness of this algorithm proves that all regular languages are described by some regular expression.

2. To introduce this algorithm, let’s think about how we would convert the “divisible by 3” FDA we have considered previously into a regular expression:

3. Looking at the diagram for this machine, it is clear that for the machine to go from state 1 to state 2 and then get back to state 1 again, it must encounter an input substring described by the regular expression $01^*0$. Given this fact, if we don’t really want to have to think about state 2, we could use the following diagram to capture the behavior of the machine.
4. This diagram is an example of what the text calls a *generalized non-deterministic finite automata* or GNFA. It is basically a NFA where instead of labeling transitions with simple symbols, we allow ourselves to use regular expressions. The idea is that the machine can move from one state to another if it finds a sequence of input symbols that match the regular expression on the edge connecting the states. The word “may” is critical here. Like an NFA, we assume this machine is very clever at guessing which strings to match with the regular expressions labeling its edges.

5. Just as we were able to eliminate state 2 in our diagram by adding an edge labeled with a regular expression to account for its absence we can also eliminate state 1.

   - In the reduced version of the machine, there is a path from m to 0 through 1 and there is also a path from 0 back to itself through 1. We will need to account for both paths with new edges.
   - To follow the path from m to 0 we must see a string that matches \((01^*0)^*1\).
   - To follow the path from 0 back to itself, we similarly must see a string that matches \((01^*0)^*1\).
   - This leads to the GNFA shown below.

6. We now have multiple edges from m to 0 and from 0 back to itself. In an NFA, this would not bother us. In a GNFA, however, since we have the power to use regular expressions as labels, we can eliminate such edges by creating a single edge labeled with the union of the regular expressions on the existing edges. Doing this to our machine yields.

7. At this point, you should be able to tell what regular expression describes the language of this machine. But it would be nice if we could continue the approach of removing nodes from the machine until we got to the point where we had a single edge labeled with the desired regular expression. This is hard to do when the machine reaches the point that all we have left is the only start state and the only final state and they are different states.

8. Given that we have \(\epsilon\)-transitions, we can fix this by creating a single, external final state with \(\epsilon\)-transitions going from what would normally be our final states to this new state.

9. Now we can use the same approach we used to eliminate states 1 and 2 to eliminate state 0 giving:
10. Amazingly, if you think about it you will (may?) realize that

\[(0 \cup (1(01^{*}0)^{*}1))(0 \cup (1(01^{*}0)^{*}1))^{*}\]

actually does describe the language of binary numbers divisible by 3.

11. Our text presents an algorithm that translates the description of a DFA into a regular expression describing the same language. The existence of and correctness of this algorithm proves that all regular languages are described by some regular expression.

12. The algorithm in the book takes the basic approach that we just followed, but streamlines things in several ways.

- First, rather than waiting to add a single, separate final state when they get in trouble, the algorithm starts by adding both a new start state and a new final state and connecting these new states to the original start state and final states with epsilon-transitions.
- Second, so that they don’t have to handle the merging of edges as a special case, they immediately add edges between all states not connected directly by edges (except for their new start and final state) that are labeled with the regular expression \( \emptyset \). They can get away with this because such edges act as if they are not there. In class and when doing your homework, it is not worth adding these edges. Just merge or add edges when appropriate.

13. With this in mind, let’s consider the “multiples of 3” machine again. To make it interesting, we can remove states in a different order. Also, I will ask you to help by telling me which edges I will have to add or augment when I remove an existing state and by telling me what the labels on these edges should be.

14. As our first step, rather than waiting until we get in trouble, we immediately augment the machine with a new start state and a new final state. We add epsilon-transitions from the new start state to the old one and from all old final states to the new one.

15. Now, instead of removing state 2, let’s try removing state 1 first. This will require adding edges from \( m \) to 2, from 2 to 0, and from 0 to 2. We will also have to update the label of the edges from \( m \) to 0, and the loops from 0 to itself and from 2 to itself. The result looks like:
16. Next we will remove state 2. This requires updating the labels of the edges from m to 0 and from 0 to itself to reflect the paths between these sources and destinations that currently pass through 2.

18. If you have a really good memory, you will have already noticed that we obtained a different regular expression by performing this sequence of state eliminations that we did last time. Eliminating 2 then 1 then 0 and m gave us:

\[(0 \cup (1(01^*0)^1))(0 \cup (1(01^*0)^1))^*\]

Eliminating 1 then 2 then 0 and m gave us:

\[(0 \cup 11 \cup 10(1 \cup 00)^*01)(0 \cup 11 \cup 10(1 \cup 00)^*01)^*\]

Hopefully, these two regular expressions describe the same sets!

17. Finally, since the only edge from start to m is an epsilon-transition, it is clear that we can remove both states m and 0 to obtain:

19. My hope is that this practice gives you a clear enough understanding of how to use GNFAs to extract a regular expression that describes the language of a DFA. The book gives a more formal presentation (almost a proof). You should reread (or read) that section now to solidify your understanding and convince yourselves that the algorithm can be applied to any DFA.

**Languages that are not regular**

1. We have derived two distinct regular expressions that (should) describe the same language — binary representations of numbers divisible by 3:

\[(0 \cup (1(01^*0)^1))(0 \cup (1(01^*0)^1))^*\]
and 
\[(0 \cup 11 \cup 10(1 \cup 00)\ast 01)(0 \cup 11 \cup 10(1 \cup 00)\ast 01)\ast\]

2. After spending hours making up the slides showing how to extract a regular expression for a language from a DFA that recognizes the language, I could not help thinking that it would be nice to have some sort of “regular expression checker” that would tell me for sure that the regular expressions
\[(0 \cup (101\ast)\ast 1)((0 \cup (1(01\ast)\ast 1)))\ast\]
and
\[(0 \cup 11 \cup 10(1 \cup 00)\ast 01)(0 \cup 11 \cup 10(1 \cup 00)\ast 01)\ast\]
actually do describe the same languages.

- I knew that different reduction orders would normally generate different regular expressions, but I also knew how easy it is (at least for me) to make mistakes performing the steps in such an algorithm.

3. If you think about it, you will realize that what I really wanted was a **decider** for the language:

\[L_{EQ-RE} = \{ e = e' \mid e \& e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e') \}\]

4. This language is a bit more interesting than most of the examples we have been talking about so far this semester. Certainly, it would be harder for you to write a program that decided whether an input belonged to this language than it would be to decide if a binary string represented a number divisible by 3.

5. If this problem does not impress you, consider the similar problem for a language somewhat richer than the language of regular expressions:

\[L_{EQ-Java} = \{ j = j' \mid j \& j' \text{ are Java programs that behave identically} \}\]

Those in the know might even suspect that writing a program to recognize strings that belong to this language is more than difficult.

6. For now, let’s stick to regular languages and ask whether a set like

\[L_{EQ-RE} = \{ e = e' \mid e \& e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e') \}\]

is regular.

7. In fact, let’s start with something even easier. In your last homework assignment I mentioned that \{ \text{a} + \text{b} = \text{c} \mid \text{a, b, c} \in \{0, 1\}\ast \text{ and the sum of the numbers represented by a and b in binary notation is the number represented by c } \} was not regular. Let’s consider an even simpler representation of addition:

\[\{ \text{a} + \text{b} = \text{c} \mid \text{a, b, c} \in \{1\}\ast \text{ and the sum of the numbers represented by a and b in unary notation is the number represented by c } \}\]

8. That is, we would like to determine whether the language

\[L_{UnaryAdd} = \{1^a + 1^b = 1^c \mid 1^k \text{ refers to a string of } k \text{ 1s and } a + b = c \}\]

is regular.

9. It is surprisingly easy to show that this language is not regular.

- If it were, then there would be some DFA M such that \(L_{UnaryAdd} = L(M)\). If \(M = (Q, \Sigma, \delta, s, F)\), then let \(n = |Q|\). That is, call the number of states in this machine \(n\).
- This machine must accept the input \(1^{2n+1} = 1^{2n+1}\).
- Since the machine only has \(n\) states, at some point while processing the prefix \(1^n\) the machine M must pass through the same state \(q\) twice. (In fact, it must do this within the first \(n\) symbols.)
- This is the key observation, because it implies that if we delete (or duplicate) the symbols between the first occurrence of \(q\) and the second, the machine cannot tell the difference. It must therefore accept both the original string and the shortened (lengthened) string even though the shortened (lengthened) string should not be in the language.
• To show this a bit more precisely, let’s give name to the state that the machine passes through as it processes the prefix $1^n$. In particular, let $q_0 = s$ and $q_i = \hat{\delta}(s, 1^i)$ so that in general, $q_i$ is the state the machine is in after processing $i$ characters of the input string.

• The sequence of states $(q_0, q_1, ..., q_n)$ has length $n + 1$. Since $|Q| = n$, at least two of the states in $(q_0, q_1, ..., q_n)$ must actually be the same state. That is, there is at least one repeated state in the sequence of states the machine passes through while processing $1^n$.

Suppose that the first repeated state appears first at steps $k$ and $l$ with $k < l$. That is, $q_k = q_l$ (or $\hat{\delta}(s, 1^k) = \hat{\delta}(s, 1^l)$). Then the last $l - k$ symbols in $1^l$ must just take the machine from state $q_k$ back to state $q_k$. In other words, the path from $s$ to some final state that the machine follows while processing $1^{2n} + 1 = 1^{2n+1}$ includes a loop.

The following GNFA-like diagram illustrates the situation. The path the machine takes while processing the input $1^{2n} + 1 = 1^{2n+1}$ can be broken down into three sub-paths corresponding to the substrings $1^k$, $1^{l-k}$, and $1^{2n-l} + 1 = 1^{2n+1}$ and the second sub-path traverses a loop in the DFA’s transition diagram.

• The diagram also makes it clear that it is possible for the machine to travel from its start state to one of its final states along a path that skips the loop. That is, if a string starts with $1^k$ and ends with $1^{2n-l} + 1 = 1^{2n+1}$ it will bring the machine to a final state.

• Therefore, the string $1^k1^{2n-l} + 1 = 1^{2n+1}$ or $1^{2n-l+k} + 1 = 1^{2n+1}$ must be accepted by $M$. We assumed, however, that $M$ only accepted strings of the form $1^a + 1^b = 1^c$ where $a + b = c$. Given that $M$ accepts $1^{2n-l+k} + 1 = 1^{2n+1}$ where $a = 2n - l + k$, $b = 1$, and $c = 2n + 1$, we would be led to conclude that $2n - l + k + 1 = 2n + 1$, but this is only true when $l = k$, and our assumption was that $l$ and $k$ were distinct positions in the sequence $(q_0, q_i, ..., q_n)$. Thus, we have a contradiction that allows us to conclude that our assumption — that $L(M) = L_{UnaryAdd}$ must be incorrect.

• This implies that $L_{UnaryAdd}$ must not be regular.