CS 361 Meeting 8 — 2/26/20

Announcements

1. Homework 3 due Friday.
2. Revised office hours on Thursday (1:00-2:30 and 4:00-4:30)

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 Σ, we define e to be a regular expression if e is

   - a for some $a \in \Sigma$
   - $\emptyset$
   - ε
   - $e_0 \cup e_1$, 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 $\emptyset$
   - $L(\varepsilon)$ 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. Here are some examples of regular expressions (and the strings they describe):

   - $b^*\#a^*$ - The language $L_{ba} = \{b^n\#a^m \mid m, n \geq 0\}$
   - $(1 \mid 0)^*$ or equivalently $(1 \cup 0)^*$ - The language of all binary strings
   - $(1^*0)^*$ - The language of binary strings that don't end with a 1.
   - $(0 \cup 10^*1)^*$ - The language of binary strings with even parity (I'm just not very original!)

5. 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.
6. 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:

     $$(0 \cup 1 \cup \epsilon)(0 \cup 1 \cup \epsilon) \cup (0 \cup 1)^*(1 \cup 010 \cup 00)$$

   - Binary strings that are multiples of 3:?

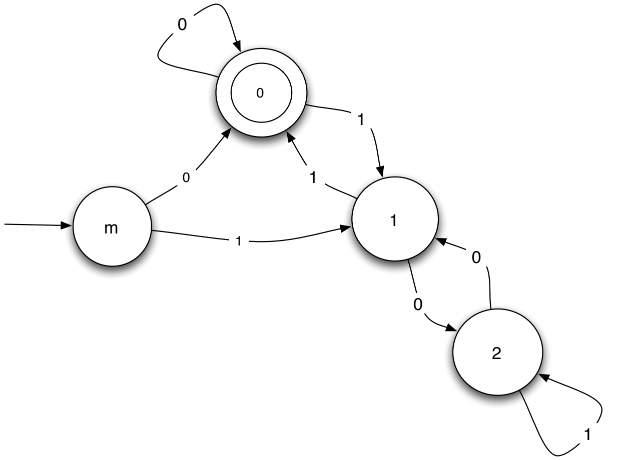
7. The last example raises the question of whether or not every regular language can be described by a regular expression.

Generalize Nondeterministic Finite Automata

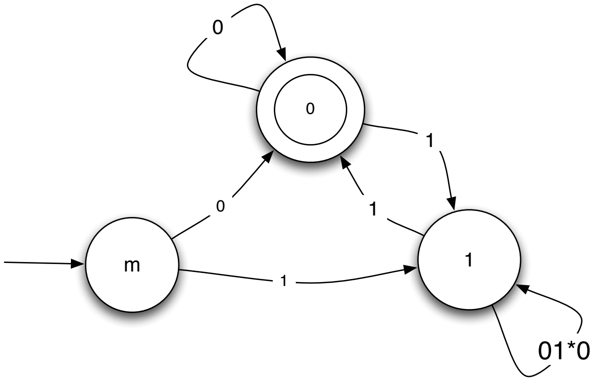
1. The book presents an algorithm that translates the description of a DFA into a regular expression describing the same language. The existence of and correctness of this algorithm proves that all regular languages are described by some regular expression.

Click here to view the slides for this class

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:



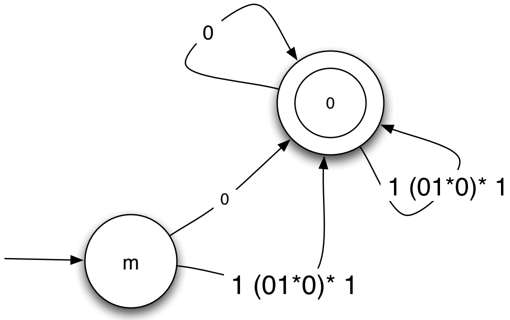
3. Looking at the diagram for this machine, it is clear that for the machine to go from state 1 to state 2 and then get back to state 1 again, it must encounter an input substring described by the regular expression 01*0. Given this fact, if we don't really want to have to think about state 2, we could 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.

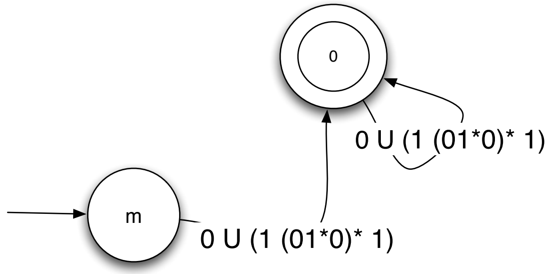
5. Just as we were able to eliminate state 2 in our diagram by adding an edge labeled with a regular expression to account for its absence we can also eliminate state 1.

   - 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 1(01*0)*1.
   - To follow the path from 0 back to itself, we similarly must see a string that matches 1(01*0)*1.
   - This leads to the GNFA shown below.



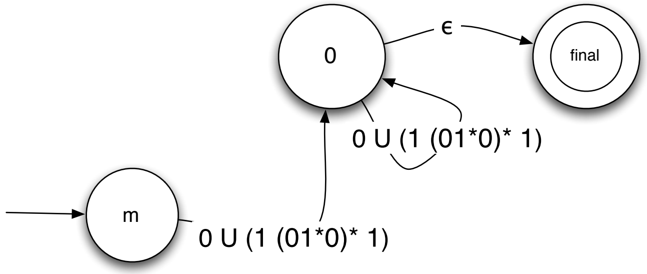
6. We now have multiple edges from m to 0 and from 0 back to itself. In an NFA, this would not bother us. In a GNFA, however, since we have

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.

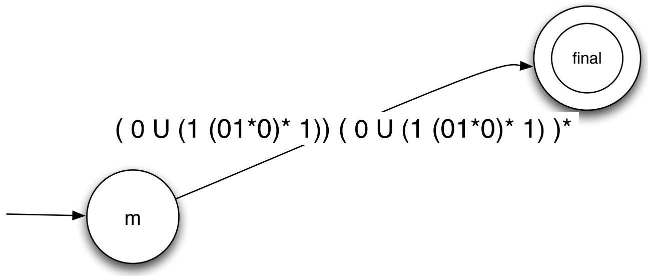


7. At this point, you should be able to tell what regular expression describes the language of this machine. But it would be nice if we could continue the approach of removing nodes from the machine until we got to the point where we had a single edge labeled with the desired regular expression. This 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 ϵ-transitions, we can fix this by creating a single, external final state with ϵ-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

$$(0 \cup (1(01^*0)^*1))(0 \cup (1(01^*0)^*1))^*$$

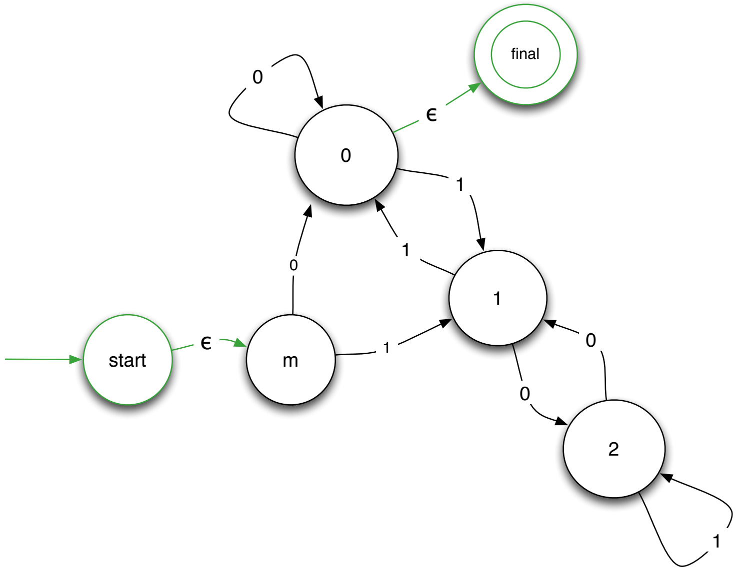
actually does describe the language of binary numbers divisible by 3.

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

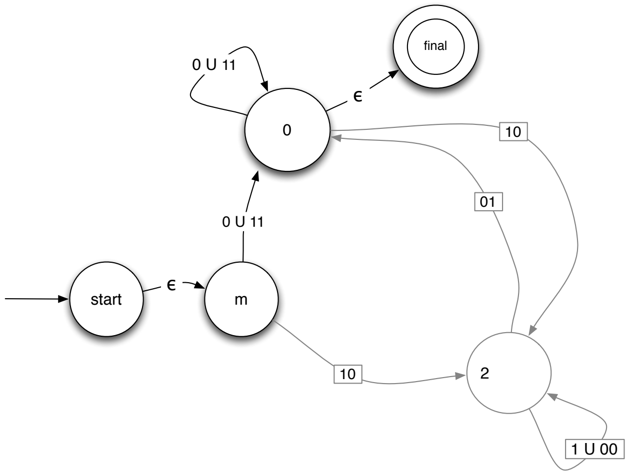
    - First, rather than waiting to add a single, separate final state when they get in trouble, the algorithm starts by adding both a new start state and a new final state and connecting these new states to the original start state and final states with epsilon-transitions.
    - Second, so that they don't 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

12. 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.

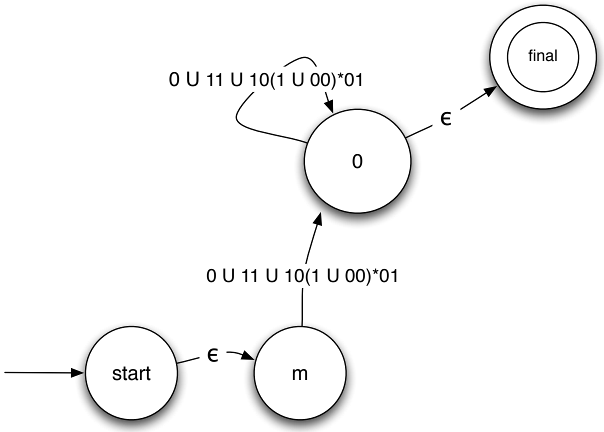
13. As our first step, rather than waiting until we get in trouble, we immediately augment the machine with a new start state and a new final state. We add epsilon-transitions from the new start state to the old one and from all old final states to the new one.



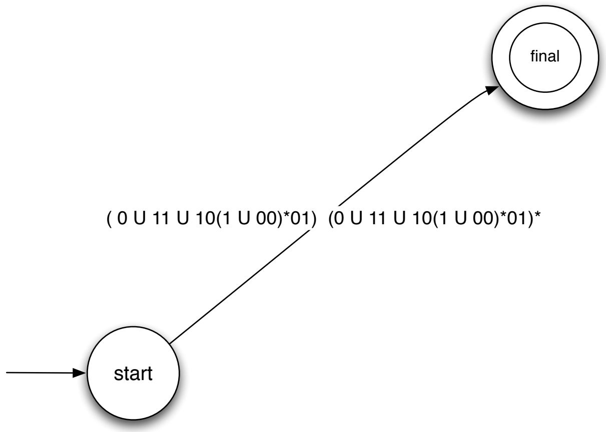
14. Now, instead of removing state 2, let's try removing state 1 first. This will require adding edges from m to 2, from 2 to 0, and from 0 to 2. We will also have to update the label of the edges from m to 0, and the loops from 0 to itself and from 2 to itself. The result looks like:



15. 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.



16. 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:



17. 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!

18. 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 you understanding and convince yourselves that the algorithm can be applied to any DFA.

Languages that are not regular

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

$$(0 \cup (1(01^*0)^*1))(0 \cup (1(01^*0)^*1))^*$$

and

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

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 two regular expressions actually do describe the same languages.

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 $\{a+b=c \mid a,b,c \in \{0,1\}^*$ 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:

   > $\{a+b = c \mid a,b,c \in \{1\}^*$ 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

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

   is regular.