SOLUTIONS, Midterm 1, CS301, Spring '09, McConnell

1. Why is it that if L is a regular language, then its complement, $\overline{L}$ is also regular?

   Solution: *Given a DFA for L, convert every accept state to a non-accept state and vice-versa, and you have a DFA for the complement.*

   Some people made the mistake of claiming that you can apply this trick to an NFA for L.

2. Why is it that if L_1 and L_2 are regular languages, then the set $L_1 - L_2$ of words that are in L_1 but not in L_2 are also a regular language.

   Solution 1: $L_1 - L_2 = L_1 \cap \overline{L_2}$. *We have shown that the complement of a regular language is regular, so* $\overline{L_2}$ *is regular. We have shown that the intersection of two regular languages is regular, so* $L_1 \cap \overline{L_2}$ *is regular.*

   Solution 2: *Given DFA's for* L_1 *and* L_2*, put your fingers on the start states of DFA's for the two languages. Each time you read a letter, move your fingers to the required states. For every state of your fingers, each letter dictates uniquely which new state they get into. Your fingers are therefore a finite state machine, a DFA. Accept if your finger in the DFA for* L_1 *ends in an accept state and your finger in the DFA for* L_2 *ends in a non-accept state.*

3. True or false? If L is a regular language and $L' \subseteq L$, then L' is regular. Prove your answer.

   Solution: $\{a,b\}^*$ *is a regular, and every language on* $\{a,b\}$ *is a subset of it. If the claim were true, then every language on* $\{a,b\}$ *would be regular. We've seen some languages on* $\{a,b\}$*, such as* $\{a^n b^n | n \geq 0\}$*, that are non-regular.*

4. Using mathematical induction, show that $\sum_{i=1}^{n} i = n(n+1)/2$.

   Solution: *B.C.:* $n = 1$*: the lefthand side is 1 and the righthand side is* $1(2)/2 = 1$.

   I.H. Suppose $n > 1$ *and* $\sum_{i=1}^{n-1} i = (n-1)n/2$.

   I.S. $\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n = (n-1)n/2 + n = ((n-1)n + 2n)/2 = ((n-1)+2)n/2 = n(n+1)/2$.

5. Give an inductive definition of the language of balanced parentheses.

   Solution: *B.C. "()" is in the language.*

   I.S. If w_1 *and* w_2 *are in the language, then so are* $w_1 w_2$ *and* (w_1).

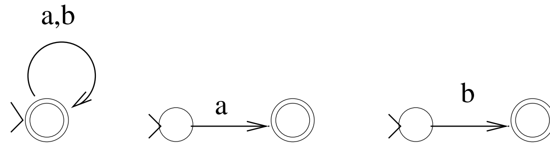
6. Using the pumping lemma, show that the language of balanced parentheses is not regular.

   Solution: *Let's prove it isn't regular by assuming the opposite and obtaining a contradiction.*

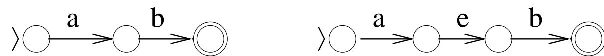
   Suppose that it is regular. There is a DFA for it. Let n be the number of states in it. Since the language is regular, the pumping lemma applies to every string in the language. Since $(^n)^n$ *is in the language, some portion of the first n parentheses can be pumped. This gives a string that's obviously not in the language. Therefore, the pumping lemma doesn't apply to every string in the language – a contradiction.*

7. Recall that our proof that if a language can be expressed by a regular expression, then it is also the language accepted by an NFA. The proof is a constructive one: it gives an algorithm that inputs a regular expression and outputs an NFA that accepts the language.

   The base cases of the proof are the following:

   The algorithm is useful not just for proof, but to be implemented on the computer. To show that you understand this, if I give you the regular expression ab and asked for the NFA it produces, it would be incorrect to give the lefthand NFA; the algorithm produces the righthand one. The lefthand one reflects a common-sense shortcut that works in this example, but we haven't described when the shortcut can be applied, so the lefthand drawing doesn't show that you know a set of rules that will work on *any* NFA.

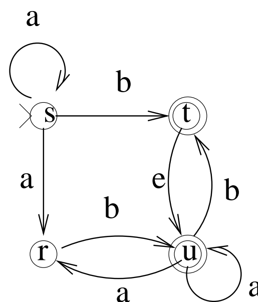
   

   One criticism I have of the way the book illustrates the algorithm on an example is their use of the sloppy lefthand figure, without an accompanying analysis of when the shortcut it illustrates can be applied. You would need such an analysis to implement an algorithm that made use of the shortcut.

   Show that you understand *exactly* how the steps of the algorithm work by showing *exactly* what NFA it produces for the regular expression $(ab(b \cup ba)^*b)^*$.

   Solution: *See class notes from 3/10. Few people missed this problem.*

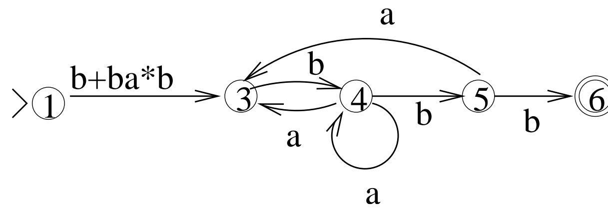
8. Using the algorithm we studied for converting an NFA to a DFA, convert the following NFA to a DFA.

   To make it possible to grade fairly, I must mark it wrong if it has a mistake. I usually make a mistake on my first attempt on a problem like this, but I know how to find them and correct them if I am patient. To avoid getting it marked wrong when you understand the algorithm, you should probably spend at least as long checking your work as you did constructing your answer. To help you check your work, I will tell you that the final DFA has four states, two of which are accepting.

   Solution: *See class notes from 3/10.*

9. The algorithm we studied for convering an NFA to a regular expression works by eliminating states one-by-one from an expression diagram. Suppose that at some point you have the following expression diagram, where "+" denotes "∪":



   (a) Draw the expression diagram that results when you eliminate state 3.

   Solution: *See class notes from 3/10.*

   (b) Draw the expression diagram that results when you eliminate state 4 from the diagram that results from the previous step.

   Once again, allow plenty of time to check your result. To help you check, I will tell you that the second answer has three transitions. The regular expression on one of them has 17 characters, counting necessary parentheses, the one on another has 10 characters, counting necessary parentheses, and the one on the last has only one character on it.

10. A farmer with a dog, a chicken, and a bag of grain have to cross a river on a row-boat that is only large enough to accommodate the farmer and one of these three possessions. His dilemma is that if he leaves the dog and the chicken unattended on one shore, the dog will eat the chicken, and if he leaves the chicken and the grain unattended, the chicken will eat the grain.

    Luckily, the farmer is a graduate of CSU and has taken CS301. He realizes that sequences of moves, if any, that solve the problem, constitute a language. The alphabet is $\{d, c, g, f\}$, for "dog," "chicken," "grain," and "farmer only." A string such as $cfdc$ denotes that the farmer crosses with the chicken, returns with an empty cargo, then crosses with the dog and returns with the chicken.

    The accept state is the one where the farmer, dog, chicken, and grain are safely on the right side of the river. The trap state is one where the farmer screws up and the chicken or grain are eaten.

    The problem has a solution if and only if the language accepted by the automaton is nonempty.

   (a) Draw the DFA for this language.

   Solution: *Rather than giving the DFA, I'll describe a strategy for getting it.*

   We can represent the possible "states" by who's on the right shore. Since there are four actors, there are $2^4 = 16$ *possible states:* $\{\}, \{f\}, \{d\}, \{c\}, \{g\}, \{f,d\}, \{f,c\}, \{f,g\}, \{d,c\}$ $\{d,g\}, \{c,g\}, \{f,d,c\}, \{f,d,g\}, \{f,c,g\}, \{c,d,g\}, \{f,d,c,g\}$.

   Some of these represent an illegal configuration. These are trap states. According to the DFA state minimization algorithm, the trap states can all be merged to a single state. Let's cross out the trap states:

   $\{\}, \{d\}, \{c\}, \{g\}, \{f,c\}, \{d,g\}, \{f,d,c\}, \{f,d,g\}, \{f,c,g\}, \{f,d,c,g\}$.

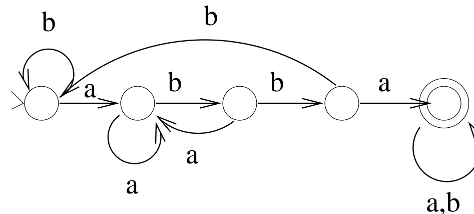
The start state is the empty state. The only one reachable from it is $\{f, c\}$, on letter c. From $\{f, c\}$, $\{c\}$ is reachable on f. However, it is also the case that the empty state is reachable on c. Many people omitted this transition because they confused "legal" moves with "stupid" moves. We are looking for a DFA for the language of legal solutions, not the language of "smart" solutions. The language of legal solutions is infinite.

From $\{c\}$, $\{f, d, c\}$ is reachable on d and $\{f, c, g\}$ is reachable on g. From $\{f, d, c\}$, $\{c\}$ is reachable on d and $\{d\}$ is reachable on c. From $\{f, c, g\}$, c is reachable on g and $\{g\}$ is reachable on c, etc. Just dumbly connecting two states if one is obtainable from the other with a crossing, we get the DFA. We've also gotten something else: by eliminating intuition, second-guessing, and common-sense from the procedure, we've gotten what amounts to an algorithm for this whole class of problems. It includes many practical problems, such as preventing a reactor-core meltdown in a nuclear plant, as well as other frivolous ones, such as the following:

> *Three men with their wives come to a river. The boat they find can only take two people at a time. Because the men are jealous, no woman can be left with another man unless her own husband is also present. So how do they manage to cross the river?*

There are many other more complicated problems of this type that I've seen as problems in programming competitions.

11. We created the following DFA for all strings that contain the substring *abba*:



See if you can come up with a general rule that can be turned into an algorithm for generating such a DFA for any string on $\{a, b\}$, not just *abba*.

Note that, in the picture, we can divide the transitions into *on-track* transitions, which point to the right and spell out the word, and *off-track* transitions that are either loops or point backward to an earlier state. Each state except the last has an on-track transition out of it and an off-track transition out of it.

I am looking only for a rule that could be turned into an algorithm for determining which state the off-track transition should point to.

Here is some helpful terminology for describing your rule. If w is a word and you can rewrite it as the concatenation xy, then x is a *prefix* of w and y is a *suffix* of w. Notice that the shortest prefix and suffix of w is e and the longest is w itself. Notice also that in our automaton for *abba*, that there is one state for each prefix of *abba*. Your general rule for any string w, not just *abba*, should describe which prefix the off-track edge out of a node should point to.

Motivation: *If you solve this problem, you are part of the way to rediscovering the fastest known algorithm that, given strings w and t, searches for all occurrences of w in*

t. This operation is of vital importance in text editing, databases, and computational biology. Google is very interested in this sort of thing.

Solution: *Create one state for each prefix of w. In the case of $w = abba$, the states were $\{e, a, ab, abb, abba\}$. The start state is e and the accept state is w. Every transition from accept state w leads back to state w. For each other state x and letter $c \in \Sigma$, the transition out of x on letter c goes to the longest suffix y of xc that is a prefix of w. (A special case is when that suffix is xc; this is the edge to the next longer prefix of w.)*

Since it's easy to come up with an algorithm for finding the longest suffix of one string that's a prefix of another, this gives an algorithm for generating such a DFA for any string.

12. **Omitted because of time constraints: Why the minimum-state DFA for a regular language is unique.**

    I went over this proof in class one day. It's not covered by our text, but the result is surprising and useful. We saw some of its uses during the review before the midterm. For instance, we used it to get an algorithm for determining whether two regular expressions express the same language.

    Let L be a regular language. For $x \in \Sigma^*$ let L_x be the language $\{z | xz \in L\}$. Let us define an equivalence relation R on Σ^* where $(x, y) \in R$ if and only if $L_x = L_y$.

    (a) Why is it that if $(x, y) \notin R$, x and y lead to different states in any DFA for L?

    Solution: *Let's suppose x and y lead to the same state, and get a contradiction. Since $(x, y) \notin R$, then there exists z such that exactly one of $\{xz, yz\}$ is in L. Without loss of generality, suppose $xz \in L$. Then $yz \notin L$. If x leads to state q, then the path from q labeled z leads to an accept state t. Since y also leads to q, it follows that yz leads to t, and yz is accepted, contradicting $yz \notin L$.*

    (b) Why is it that if $(x, y) \in R$, x and y lead to the same state in the minimum-state DFA produced by our state-minimization algorithm?

    Solution: *We defined the* language *$L(q)$ of each state q to be the language of strings accepted if you use q as the start state. We merged states if they had the same language. $L_x = \{z | xz \in L\}$ is the language of the state that x leads to and $L_y = \{z | yz \in L\}$ is the language of the state that y leads to. Since $(x, y) \in R$, these two languages are the same. Therefore, x and y lead to the same state in our minimum-state DFA.*

    (c) Why is it that the minimum-state DFA for a regular language is unique?

    Solution: *R is an equivalence relation. We've shown in part (a) that strings in different equivalence classes* can't *lead to the same state. In part (b) we get a DFA where two strings in the same equivalence class* do *lead to the same state. This is therefore the best that we could hope for. Since the equivalence classes are defined on Σ^* by the language, not by any particular representation of the language, they are unique. It is easy to see that the transitions must also be unique. (There is a transition from q_1 to q_2 labeled c if and only if $L(q_1)$ is the concatenation of the languages $\{c\}$ and $L(q_2)$.)*