Problem Set 3: Finite Automata, Regular Expressions, Non-regular Languages
CS254, Fall 2019, Anna Rafferty

You are welcome to talk to others about the problems on this problem set, but you should **read and attempt each problem prior do discussing it with others.** The write up must be your own work. That means that **you should write your answers by yourself, without looking at notes from sessions with collaborators, and you must be able to explain any answer you write down.** You should list anyone you collaborated with in your collaboration form (linked on Moodle).

Due dates: Any 4 problems due on Monday, 30 September. The remaining 3 problems are due on Wednesday, 2 October. As for all homeworks, if a problem is due on a specific day it must be turned in by 9PM on that day. See homework handout (a copy is posted on Moodle) for more specific homework policies.

1. Sipser 1.41.

Solution: We want to show that if languages A and B are regular, then the prefect shuffle of A and B (shuffle(A, B)) must be regular. We define shuffle(A, B) as follows:

$$\text{shuffle}(A, B) = \{w \mid w = a_1 b_1 \dots a_k b_k \text{ where } a_1 \dots a_k \in A \text{ and } b_1 \dots b_k \in B\}$$

The big idea of this proof will be to make a new DFA out of the DFAs for languages A and B where we go "back and forth" between these original DFAs. Let's describe that more formally.

Since A, B are regular, we know there exist DFAs $M_A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$ such that $L(M_A) = A$ and $L(M_B) = B$. We construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows:

- $Q := (Q_A \times Q_B) \cup (Q_B \times Q_A) \cup \{(\cdot, q_{0A})\}$
- $\Sigma := \Sigma$
- $\delta((r,s),a) := \begin{cases} (\delta_A(q_{0A}, a), q_{0B}) & \text{if } r = \cdot \text{ and } s = q_{0A}, \\ (\delta_A(s,a), r) & \text{if } s \in Q_A, r \in Q_B, \\ (\delta_B(s,a), r) & \text{if } r \in Q_A, s \in Q_B. \end{cases}$
- $q_0 := (\cdot, q_{0A})$
- $F := F_B \times F_A = \{(q_{FB}, q_{FA}) \mid q_{FA} \in F_A, q_{FB} \in F_B\}$

What is this machine doing? It's states are pairs of states in M_A and M_B that track where it is in each machine, and which state is first versus second in the pair tracks whether the next character it reads is at an odd of even index - that is, whether it should be in the string for language A or language B. It starts off in the state $(\cdot, q_{0A})$, with the fact that a state from M_A is the second item meaning that we're about to read a character at an even index (think of this like we're starting in the start state for machine A) and the first item being $\cdot$ meaning that we haven't yet read any characters and thus haven't yet been in M_B. At each transition, we switch whether the state from M_A or M_B is in the second spot (meaning the "current" state), and we use the first spot in the pair to remember where we should go next in the other machine. We should accept a string if at the end of reading it, the second item in the pair is a state from M_A (meaning we'd be about to read another even index character, so the length of what we've read so far is even), and both the states in M_A and M_B that we're in are accept states.

Now, let's turn the intuition in the previous paragraph into a proof that this machine works. We'll break into two cases: odd length strings and even length strings. Consider a string w where the length of w is odd. Then $w \notin$ shuffle(A,B) because every $w = a_1 b_1 \dots a_k b k$ for some k, meaning $|w| = 2k$. Our definition of δ means that a state (r, s) with $s \in Q_A$ is reachable only with an even number of transitions from the start state (since whether $s \in Q_A$ or $s \in Q_B$ switches with every transition and the start state has $s \in Q_A$). Thus, upon reading string w in M, we'll end in a state (r, s) where $s \in Q_B$, and such states (r, s) are not in $F_B \times F_A$. Thus, $w \notin L(M)$.

Next, consider a string w where the length of w is even. That means we can write $w = a_1 b_1 \dots a_k b_k$ for some k. Consider the sequence of states $(r_1, s_1), \dots, (r_m, s_m)$ that will be entered in M when reading string w. Note that $m = 2k + 1$ here is odd, since we enter one more state than the number of characters in the string we read.

By the definition of δ, for each consecutive pair of states $(r_i, s_i), (r_{i+1}, s_{i+1})$, $r_i = s_{i+1}$, with the exception of $i = 1$ (where $s_2 = q_{0B}$).

Consider first only the first items of even indexed states in the sequence: $r_2, r_4, \ldots, r_{m-1}$. By our definition of δ, $r_2 = \delta_A(q_{0A}, a_1)$, and for each r_i where $i > 2$ and i is even, $r_i = \delta_A(r_{i-2}, a_{i/2}) = r_i$. Additionally, $s_m = r_{m-1}$. Thus, by definition of when a machine accepts a string, $a_1 \ldots a_k \in L(M_A)$ iff $s_m \in F_A$.

Now, we'll consider the first items of odd indexed states in the sequence: $r_1, r_3, \ldots, r_m$. By our definition of δ, $r_1 = \cdot$, $s_2 = q_{0B}$, and thus $r_3 = \delta_B(q_{0B}, b_1)$. Then for r_i where $i > 3$ and i is odd, $r_i = \delta_B(r_{i-2}, b_{\lfloor i/2 \rfloor}) = r_i$. hus, by definition of when a machine accepts a string, $b_1 \ldots b_k \in L(M_B)$ iff $r_m \in F_B$.

Thus, for a string $w = a_1 b_1 \ldots a_k b_k$, $\hat{\delta}((\cdot, q_{0A}), w) \in F_A \times F_B$ iff $a_1 \ldots a_k \in L(M_A)$ and $b_1 \ldots b_k \in L(M_B)$. That means $w \in L(M)$ iff $a_1 \ldots a_k \in L(M_A)$ and $b_1 \ldots b_k \in L(M_B)$, which by our definition of the shuffle operations, means that $L(M) = \text{shuffle}(A, B)$.

Note: The reason for having a separate start state rather than (q_{0B}, q_{0A}) is to disallow ϵ, since the definition of shuffle seems to imply that the strings in A and B must be of length at least 1 (since, e.g., $a_1 \ldots a_k \in A$). A solution that assumed that $\epsilon \in \text{shuffle}(A, B)$ iff $\epsilon \in A$ and $\epsilon \in B$ could omit the separate $(\cdot, q_{0A})$ state.

2. Consider the "prefix" operation over languages A and B:

$$A \text{ prefix } B = \{w \mid w \in A \text{ and some prefix of } w \in B\}$$

A string x is a prefix of a string w if there exists a string y such that $xy = w$. Show that the regular languages are closed under the prefix operation.

Solution: We want to show that if A, B are regular languages, then A prefix $B = \{w \mid w \in A \text{ and some prefix of } w \in B\}$ is regular. As above, we'll construct a machine that recognizes this prefix language. We'll do something similar to our proof of intersection, but change the second item in the pair to just always be ACCEPT if we've entered an accept state in the machine that recognizes language B.

More formally, assume that A, B are regular languages. Then there exist DFAs $M_A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$ such that $L(M_A) = A$ and $L(M_B) = B$. We'll construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ for A prefix B as follows:

- $Q := Q_A \times (Q_B \cup \{\text{ACCEPT}\})$
- $\Sigma := \Sigma$
- $\delta((q_a, q_b), c) := \begin{cases} (\delta_A(q_a, c), \text{ACCEPT}) & \text{if } \delta_B(q_b, c) \in F_B \text{ or } q_b = \text{ACCEPT}, \\ (\delta_A(q_a, c), \delta_B(q_b, c)) & \text{otherwise.} \end{cases}$
- $q_0 := (q_{0A}, q_{0B})$ if $q_{0B} \notin F_B$. Otherwise, $q_0 := (q_{0A}, \text{ACCEPT})$.
- $F := (r, s) \mid r \in F_A$ and $s = \text{ACCEPT}$

Now consider a string w, and the resulting state from reading w through this machine: $\hat{\delta}(q_0, w) = (r, s)$. Since we have defined δ such that the first item in the pair always follows transitions from δ_A, $r \in F_A$ if and only if $\hat{\delta}_A(q_{0A}, w) \in F_A$, meaning $w \in A$. The second item in the states follows δ_B except for transitions into an accept state, which are sent to ACCEPT, and there are no outgoing transitions from ACCEPT for the second item in the pair. If there exists some string x such that $xy = w$ for a string y and $x \in B$, then $\hat{\delta}_B(q_{0B}, x) \in F_B$. After reading x, the second item in the pair will be ACCEPT. Thus, if such an x exists, $\hat{\delta}(q_0, w) \in F$ if and only if $w \in A$. Alternatively, if no such string $x \in B$ exists, then $\hat{\delta}_B(q_{0B}, x) \notin F_B$ for any prefix x of w. Thus, the second item in the final state pair cannot be ACCEPT, and thus $\hat{\delta}(q_0, w) \notin F$. Thus, $\hat{\delta}(q_0, w) \in F$ if and only if $w \in A$ prefix B, meaning that the language of machine M is A prefix B. We have shown that A prefix B is regular if A, B are regular, which means that the regular languages are closed under prefix.

3. We talked a bit about Python regular expressions in class and in the reading handout. In this problem, you'll practice using these regular expressions to both find information in a string and change a string. All of your answers to this problem should go in a single file named `ps03_03.py`. (Yes, that's an underscore when we usually use dashes. The reason is that importing Python modules with dashes is somewhat more annoying.)

(a) First, you'll write a regular expression that can find email addresses in a string. Email addresses have a local-part and a domain. The local-art and the domain are separated by an symbol. For instance, in my email address, `arafferty` is the local part and `carleton.edu` is the domain. According to Wikipedia, the local-part can consist of any upper or lowercase characters, any digits, and any of the following characters: !#$%&'*+-/=?^_`{}|~;. It can also have a period (.) in it as long as it doesn't start with the period, and we'll limit the local-part to be 54 characters or fewer. The domain can consist of upper or lower case letters, any digits, hyphens (-), and periods. Like the local-part, the domain cannot start with a period. Both the local-part and the domain must contain at least one character.[1] Write a function `get_all_emails` that takes a single string parameter and returns a list where each item is an email address in the string. Here's an example of me using mine:

```
>>> import ps03_03.py
>>> ps03_03.get_all_emails('eris@sleepycat,all.done+YAY@compl3t3 oh and lawn#mower@g.a.r.d.e.n')
['eris@sleepycat', 'all.done+YAY@compl3t3', 'lawn#mower@g.a.r.d.e.n']
>>> ps03_03.get_all_emails('.turtles@SEA-Turtle')
['turtles@SEA-Turtle']
>>> ps03_03.get_all_emails('.turtles@.SEA-Turtle')
[]
```

(b) Now, write a function that makes emails a little harder to find. Specifically, it rewrites the email into the domain name, followed by the words " preceded by @ and then preceded by ", and then followed by the local-part. (The example below makes this more clear!)

You'll write this in a separate function `obfuscate` that takes a single string parameter and returns the string with any emails having this modification. It's okay if this code has some overlap with your previous code. You'll need to use groups and substitution. Take a look at this example from the Python documentation for additional examples of substitution: https://docs.python.org/3.7/library/re.html#text-munging.

Here's an example of me using mine:

```
>>> ps03_03.obfuscate('Here is my email: turtles@SEA-Turtle')
'Here is my email: SEA-Turtle preceded by @ and then preceded by turtles'
>>> ps03_03.obfuscate('eris email is eris@sleepycat, and YAY has email too: all.done+YAY@compl3t3')
'eris email is sleepycat preceded by @ and then preceded by eris, and YAY has email too: compl3t3
    preceded by @ and then preceded by all.done+YAY'
```

4. $\bar{A}$ is the complement of a set A: $\bar{A} = \Sigma^* - A$. Many regular expression implementations in various programming languages have a built in complement operator. As we showed in class, regular languages are closed under complementation, so including this operator doesn't provide any extra power: these implementations of regular expressions can match the same languages as the version that Sipser describes. Write an algorithm for implementing the complement operator using the other regular expression operators. That is, write an algorithm that starts with a regular expression R and produces a new regular expression S such that $L(S) = \Sigma^* - L(R)$. Illustrate your algorithm's behavior using the example of $R = b \cup c$ over the alphabet $\Sigma = \{a, b, c\}$. Your algorithm may use any algorithms provided in Sipser - e.g., converting among different machines.

Solution: You can solve this by using other algorithms in Sipser. Let $R' = \bar{R}$ (i.e., we want a regular expression for the complement of R). First, convert R into an NFA N as discussed in class and in the book. Then, convert N into a DFA D, again following Sipser. We now transform this DFA into a DFA D' that recognizes the complement of its current language by making the new final states F' equal to all states $Q - F$, where F is the set of final states for machine D. We now have a DFA D' such that $L(D') = R'$. Next, we convert this DFA into a GNFA, and then convert the GNFA back into a regular expression following Sipser.

Implementing this algorithm to recognize $R' = \overline{(b \cup c)}$, we convert first to a five state NFA that recognizes R. Then we convert this into a DFA with for $b \cup c$; following Sipser we can do the conversion and simplify if we want to make things smaller. This gives us a three state DFA with one final state. We now swap the final and non-final states to make the other two states final. Then we convert this new machine into a GNFA and then into a regular expression. The final output is $\epsilon \cup (a \cup (b \cup c)(a \cup b \cup c)(a \cup b \cup c)^*)$.

5. Let $\Sigma = \{a, b, c\}$, and consider a language L over Σ where $L = \{w \mid |w| = 3 \cdot \#a(w)\}$. That is, L contains all strings w where the length of w is 3 times the number of a's in w. Decide if L is regular, and prove your answer correct.

[1] There are actually additional restrictions on what can be an email address, which you can find more about if you read the linked Wikipedia article, and the 54 characters or fewer is not a restriction in real email addresses but this definition is what we'll use for this assignment.

Solution: L is not regular. We show this via a proof by contradiction. Assume L is regular. Then let p be the pumping length for L. We choose a string $s = a^p b^{2p}$. $|s| = 3p \geq p$, so by the pumping lemma, s can be pumped. Consider how we might write $s = xyz$. We know that $|xy| \leq p$, which means x and y can contain only as. Because $|y| > 0$, y must contain at least one a. Let $m \geq 0$ be the number of as in x, and $n > 0$ be the number of as in y. Then $s = a^m a^n a^{p-m-n} b^{2p}$. We choose $i = 0$, and examine $xy^0z = a^m a^{p-m-n} b^{2p}$. We see that this string has $p - n$ as, and total length $3p - n$. This string cannot be in the language because $3 * (p - n) \neq 3p - n$ unless $n = 0$, and we know $n > 0$ to make $|y| > 0$. Thus, we have a contradiction since $xy^0z \notin L$. Our assumption is incorrect, and L is not regular.

6. Sipser 1.35.

Solution: We seek to show that $E = \{w \in \Sigma_2^* \mid$ the bottom row of w is the reverse of the top row of w is not regular. We'll use proof by contradiction and the pumping lemma. Assume that E is regular. Then there must exist a pumping length p. We choose string $s = \begin{bmatrix}0\\1\end{bmatrix}^p \begin{bmatrix}1\\0\end{bmatrix}^p$. Then the first row of s is 0^p1^p and the bottom row is 1^p0^p, so the bottom row is the reverse of the top (we showed in class that $(xy)^R = y^R x^R$). We have that $|s| = 2p > p$, so there must be a way to write $s = xyz$ such that $|xy| \leq p$ and $|y| > 0$ such that for all i, xy^iz is also in A_2. Since s beings with p of the character $\begin{bmatrix}0\\1\end{bmatrix}$, all characters in xy are $\begin{bmatrix}0\\1\end{bmatrix}$. Let y contain k of these characters, where $k > 0$ (by condition 2 of the pumping lemma), and $k \leq p$. We choose $i = 0$. Then $xy^0z = \begin{bmatrix}0\\1\end{bmatrix}^{p-k} \begin{bmatrix}1\\0\end{bmatrix}^p$. Then the top row is equal to $0^{p-k}1^p$. Letting $x = 0^{p-k}$ and $y = 1^p$, we have by the theorem above that the reverse of this row is 1^p0^{p-k}. However, the bottom row is $1^{p-k}0^p$. These are not equal since for $k \neq 0$, $p - k \neq p$. Thus, $xy^0z \notin E$. We thus have a contradiction, so our assumption was incorrect and E is not regular.

Note: If after doing this problem you don't feel confident about being able to prove languages non-regular using the pumping lemma, try a couple more of the "prove language L is non-regular" in Sipser. The solutions for 1.29a and 1.29c are in your book.

7. Sipser 1.54. (This problem should help you understand what the pumping lemma says and crucially what it doesn't say!)

Solution: The language $F = \{a^i b^j c^k \mid i, j, k \geq 0$ and if $i = 1$ then $j = k\}$. We can prove it is non-regular using the closure properties of regular languages. Assume F is regular. Consider $L = \{w|w$ contains exactly one a$\}$. This language is regular: we can draw an NFA with two states. The start state loops on b, c, there is a transition from the start to the final state on a, and the final state loops on b, c. This machine accepts language L, so L must be regular.

Then $L \cap F = \{ab^k c^k \mid k \geq 0\}$. We can prove this language is non-regular using the pumping lemma, with the proof looking very similar to the proof that $b^k c^k$ is not a regular language, except that in rewriting our string $w = ab^p c^p$, where p is the pumping length, we have to consider all possibilities for x and y:

- $x = \epsilon$, $y = ab^j$ for some $0 \leq j \leq p - 1$: Then xy^0z will not have any as, and thus will not be in the language.
- $x = ab^n$, $y = b^j$ for some $n \geq 0$, $j \geq 1$, and $n + j \leq p - 1$: Then xy^0z will have $p - j$ bs and p cs. Thus, this string is not in the language.

Since there's no way to write $w = xyz$ such that all three conditions of the pumping lemma hold, we have a contradiction: $L \cap F$ is non-regular. Since the regular languages are closed under intersection and we know L is regular, F must be non-regular.

The second part of the problem asks us to show that F acts like a regular language in the pumping lemma. Let $p = 4$. For any string s ($|s| \geq p$), it could have one of three forms: it begins with a b or a c ($i = 0$), it begins with an a followed by a b (i=1), or it begins with more than one a (i > 1). If s is of the form given in the first case, we can write it as $b^m c^n$ for $m, n \geq 0$. Then we let $x = \epsilon$ and $y = b$. Then $|xy| = 1 \leq 4$ and $|y| > 0$. For any i, $xy^iz = b^i b^m c^n = b^{i+m} c^n$. This fits the form given for F, so $xy^iz \in F$. We could also have form two, which we can write as $ab^m c^m$, $m > 0$ (since $|s| \geq 4$). Let $x = \epsilon$ and $y = a$. Then we again have that $xy^iz = a^i b^m c^m \in F$. In the final case, $x = a^\ell b^j c^k$ where $\ell \neq 1$. If $\ell > 2$, we let $x = a^2$, $y = a$. Then $xy^iz = a^2 a^i a^{\ell-3} b^j c^k = a^{\ell-1+i} b^j c^k$. $\ell - 1 + i$ is

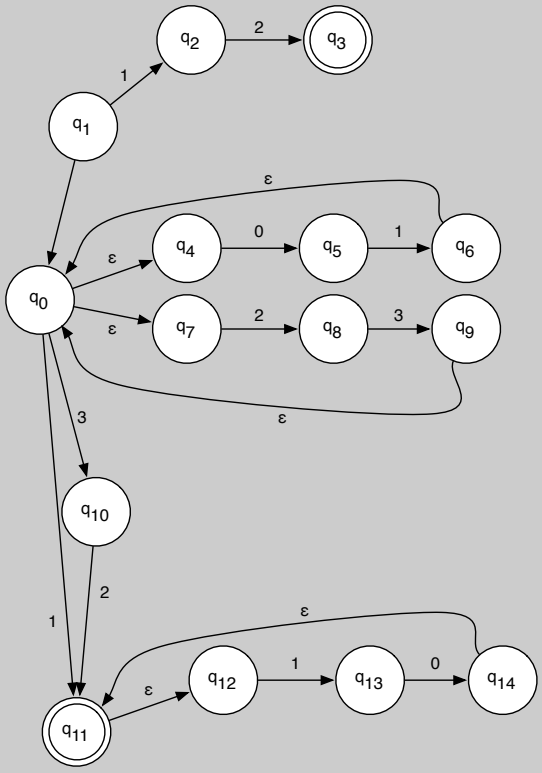
at least 2 since $\ell > 2$, so any such string is in F. Finally, if $\ell = 2$ (we know it is not 0 or 1 since in this case we know the string starts with at least two a's), then we make $x = aa$, $y = b$. Then $xy^iz = aab^ib^jc^k = a^2b^{i+j}c^k \in F$. Thus, F satisfies the conditions of the pumping lemma.

This is not a contradiction because the pumping lemma says that if a language is regular, then the conditions of the pumping lemma hold. It doesn't say anything about whether the conditions hold if a language is not regular.

8. **Optional questions**: Note: You do not need to turn these problems in, and they will not be graded. I encourage you to try them anyway so you ensure you're comfortable converting among DFAs, NFAs, and regular expressions.

   (a) Let $\Sigma = \{0, 1, 2, 3\}$. Draw an NFA that recognizes the same language as the regular expression $(01 \cup 23)^*(32 \cup 1)(10)^* \cup 12$. Here, we've omitted writing concatenation using $\circ$ and just written it as two regular expressions following one another (e.g., if R_1, R_2 are regular expressions, $R_1 \circ R_2 = R_1R_2$). After you draw the NFA, convert the NFA into a DFA. *Note: You do not need to turn this problem in, and it will not be graded. I encourage you to do it anyway so that you can go in both directions for converting between NFAs and regular expressions, and so you make sure you can convert from an NFA to a DFA.*

   Solution: Here's an NFA. It's not minimal, but hopefully makes clear where the different parts of the regular expression come into the machine:



   (b) Choose one of the DFAs or NFAs you drew for last week's problem set, and convert it to a regular expression.