University of Virginia - cs3102: Theory of Computation Spring 2010

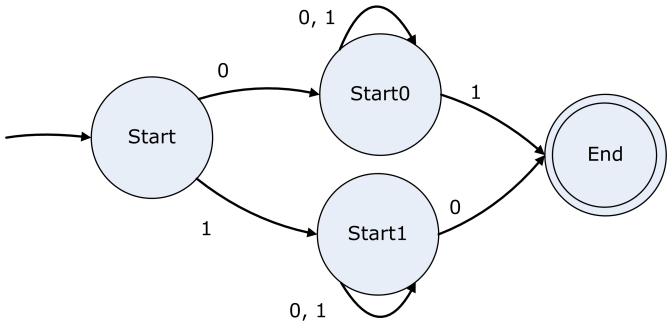
PS2 - Comments

Average: 77.4 (full credit for each question is 100 points)

Distribution (of 54 submissions): $\geq$ 90, 12; 80 – 89, 11; 70-79, 15; below 70, 16.

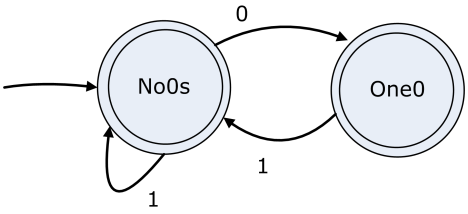
Problem 1: Constructing NFAs. For each of the following languages, draw an NFA that recognizes the language using *fewest possible number of states*. For all languages, assume $\Sigma = \{0,1\}$.

a. (Average: 2.8, full credit: 3) $\{w|w$ starts and ends with different symbols. $\}$

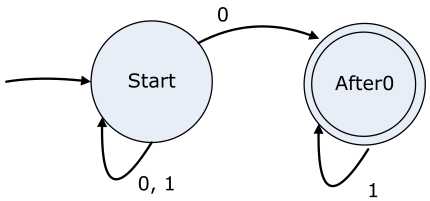


Note that the permanent rejecting state is not shown (for this and all the other diagrams). For any state where there is no transition on a given symbol, there is a transition from that state on that symbol to a permanent rejecting state.

b. (2.96 / 3) $\{w|w$ does not contain two consecutive 0s. $\}$



c. (3.51 / 4) The language described by the regular expression $\{0,1\}^* 01^*$.



Note that this language is the same language as $\{0,1\}^* 0 \{0,1\}^*$ (all bitstrings containing at least one 0).

Problem 2: Nondeterminstic Finite Automata. Answer these questions for the NFA N_1 in Sipser's Example 1.38. The extended transition function, δ^*, is as defined in the regular languages notes.

a. (1.19 / 2) What is the set of possible states of N_1 after processing input `1`?

Answer: $\{q1, q2, q3\}$
We start in state q1. On input 1 there are two possible transitions: the self-edge that goes back to state q_1 and the edge that goes to state q_2. We also need to follow the ϵ-transitions. There is one from q_2 to q_3, but no ϵ-transitions out of state q_3.

b. (1.81 / 2) What is the set of possible states of N_1 after processing input `010`?

Answer: $\{q_1, q_3\}$

c. (2.70 / 3) Is there a string w such that $\delta^*(q_1, w)$ does not include q_1? (If so, give the string w. If not, argue why no such w exists.)

Answer: No such string exists. Since q_1 has a self-edge for inputs 0 and 1, for any string in $\{0,1\}^*$ one possible state of the NFA is q_1. This is because the NFA can always follow the self-edges and stay in q_1.

d. (2.66 / 3) Is there a string w such that $\delta^*(q_1, w) = \{q_1, q_2, q_3, q_4\}$? (If so, give the string w. If not, argue why no such w exists.)

Answer: The shortest such string is 11. We know there is no shorter string since to reach state q_4, we need to process at least two 1s (for the transitions between $q_1 \to q_2$ and $q_3 \to q_4$. On input 11, we can reach all four states: $q_1 \to_1 q_1 \to_1 q_1$; $q_1 \to_1 q_1 \to_1 q_2$; $q_1 \to_1 q_1 \to_1 q_2 \to_\epsilon q_3$; $q_1 \to_1 q_2 \to_\epsilon q_3 \to_1 q_4$. (We use the notation $q \to_x t$ to mean we are consuming input $x \in \Sigma_\epsilon$ and transitioning from state q to state t.)

Problem 3: Eliminating Epsilon. (9.32 / 10) Prove that for every NFA N, there is an NFA N' that recognizes the same language as N but does not use any ϵ-transitions. (A very short, very convincing proof is possible, but you will receive some credit for a longer proof.)

Answer: The easy way to prove this is to use the equivalence of NFAs and DFAs (as proven in class and in Theorem 1.39). Since we can use this construction to produce a DFA N_D that recognizes the same language as N, it is trivial to construct N' from N_D (just modify the transition rules by replacing the output state with a singleton set containing that state).

Problem 4: Priming the Pump. (8.39 / 10) Use the pumping lemma to prove the language, $PRIMES$, is non-regular:

$$PRIMES = \{1^n | n \text{ is a prime number}\}$$

Answer: Assume $PRIMES$ is regular with some pumping length p. We use the pumping lemma to obtain a contradiction.

Choose $s = 1^q$ where q is some prime number greater than p. (Note that we cannot choose $s = 1^p$ since p might be composite. But, since there is no maximum prime, we know that for any p such a q must exist.)

The pumping lemma requires that s can be broken into xyz with $|y| \geq 1$. Hence, y must contain at least one 1: $y = 1^k$, $k \geq 1$, and xz contains the remaining $q - k$ 1s.

Thus, the strings that result from pumping are $1^{ki}1^{q-k} = 1^{k(i-1)+q}$. Choose $i = q + 1$. The string xy^iz is $1^{kq+q} = 1^{(k+1)q}$. This cannot be in $PRIMES$ since it has $(k+1)q$ 1s which is divisible by q.

We have a contradiction, so $PRIMES$ is non-regular.

Problem 5: Regularity. For each language below, answer whether or not the language is regular. Include a convincing proof supporting your answer. You may use any proof technique you want, including construction, the pumping lemma, and the closure properties we have established for regular languages in the book, class, and other problems.

a. (1.6 / 2) $\{a^n b^m | n \geq 0, 0 \geq m \leq 27\}$

Answer: There was a typo in the question: $0 \geq m \leq 27$ is only satisfied by $m = 0$. This doesn't effect the answer though — in either case the language is regular. Here is a regular grammar that recognizes the language (for $m = 0$):

$S \rightarrow \text{aS}$
$S \rightarrow \epsilon$

For $0 \leq m \leq 27$:

$S \rightarrow \text{aS}$
$S \rightarrow \epsilon$
$S \rightarrow \text{b}B_{26}$
$B_{26} \rightarrow \text{b}B_{25}$
$B_{26} \rightarrow \epsilon$
...
$B_1 \rightarrow \text{b}B_0$
$B_1 \rightarrow \epsilon$
$B_0 \rightarrow \epsilon$

Another way to see this is that the language is the concatenation of a regular language and a finite language, so it must be regular.

b. (1.8 / 2) $\{a^n b^m | m = 2n\}$

Answer: The language is **not** regular. Recognizing it requires counting the number of as to match the number of bs, but the number is unbounded so cannot be kept by any finite state machine.

This argument is enough to be convincing. You could also answer this by using the pumping lemma to get a contradiction. Choose $s = a^p b^{2p}$ which is in A. From the pumping lemma, $s = xyz$ where $|xy| \leq p$. The length constraint on xy requires that y must be within the first p symbols of s which is a^p. Hence, pumping y increases the number of as but not the number of bs. This produces a string $a^{p+yi}b^{2p}$, which is not in A for any $i \neq 1$. Thus, we have a contradiction and know the language is not regular.

c. (1.8 / 2) $\{w | w$ is a valid email address$\}$ (for a summary of the specification of a valid email address, see http://en.wikipedia.org/wiki/E-mail_address#RFC_specification)

Answer: The language is finite, so must be regular. We know the language is finite because the specification places a limit on the maximum length of an email address: "The local-part of an e-mail address may be up to 64 characters long and the domain name may have a maximum of 255 characters.".

d. (1.6 / 2) $\{w | w$ describes a chess position where White has a winning strategy$\}$

We describe a chess position with a sequence of 64 symbols from the alphabet $\Sigma = \{-, p, n, b, r, q, k, P, N, B, R, Q, K\}$ where the lower-case letters represent a white pieces and the upper-case letters represent the corresponding black pieces. White has a winning strategy for a given chess position if there is a way for white to win the game no matter what the other player does. (Note: you do not need to know anything more about chess to answer this.)

Answer: The language is finite, so must be regular. The position is a string of 64 symbols, so the number of possible positions is finite (13^{64}). We don't know enough to figure out which of those positions are winning positions, but since the number of positions is finite, it must be a regular language.

e. (1.4 / 2) $\{w |$ there is no string x such that $w = xx\}$

Answer: The language is **not** regular. Another way of defining this language is as the complement of $\{ww | w \in \Sigma^*\}$. This is not regular since recognizing it requires keeping trace of the first w to match it with the second w. Since the regular languages are closed under complement (proof by swapping the accepting and non-accepting states), this proves the language is not regular. We could prove the language is non-regular using the reverse pumping lemma:

assume the language is regular with pumping length p. Pick $s = a^p b a^p b$ which is not in the language. Because of the constraint $|xy| \leq p$, we know y must be in the first a^p. Pumping this increase the number of as before the first b, but not the number of as in the second part, producing the string $a^{p+k} b a^p b$ which *is* in the language. This contradicts the (reverse) pumping lemma, proving the language is non-regular.

f. (up to 5 bonus points were awarded, average +1) *Bonus Challenge:* $\{w | w$ is a valid day in the Gregorian calendar in the form *month day, year*$\}$

The Gregorian leap year rule is: every year that is divisible by four is a leap year, except for years that are exactly divisible by 100 but not divisible by 400. For example, "February 7, 2010", "February 29, 2012", and "February 29, 2400" are in the language, but "February 29, 2010" and "February 29, 2100" are not.

Answer: The number of days is finite, so it is easy to recognize the 364 always valid Month Day strings, followed by any year: *Valid Month Day* $[0-9]^*$. The problem is dealing with Feburary 29, which is sometimes valid, but sometimes not, depending on the year. We would need to design a DFA that can recognize whether a year (represented as a sequence of decimal digits) satisfies the requirements for having a leap day. Note that only the last four digits matter since 10000 is divisible by 400. Hence, there are a finite number of last four digits to deal with, and we can figure out which ones satisfy the leap day requirements.

Problem 6: Language Splitting. (5.8 / 10) (Based on Sipser's 1.63a) Prove that any infinite regular language (that is, a regular language with an infinite number of strings) can be split into three infinite disjoint regular subsets.

Answer: Remember that a language is a set of strings. We need to prove that any infinite regular language, A, can be divided into A_1, A_2, and A_3 where:

- $A_1 \cup A_2 \cup A_3 = A$ (the union of the shares is equal to the original language)
- A_1, A_2, and A_3 each contain an *infinite* number of strings (for *any* choice of A, an infinite regular language)
- A_1, A_2, and A_3 are disjoint (that is, $A_1 \cap A_2 = A_1 \cap A_3 = A_2 \cap A_3 = \emptyset$)
- A_1, A_2, and A_3 are regular sets (they can be recognized by some DFA)

Since A is regular we know it can be recognized by some DFA. Using the same reasoning that led to the pumping lemma, there is some pumping length p (limited by the number of states in A), after which we must have gone through a cycle, returning to the same state twice. By the pumping lemma, any string $w \in A$ with length greater than or equal to p can be subdivided into $w = xyz$ such that $|y| > 0$

and all strings xy^iz for $i \geq 0$ are in A. We can split A into three languages satisfying the needed properties by subdividing the integers into 3 disjoint sets, and making the three language shares correspond to xy^iz for each of the sets. One way to divide the integers into infinite sets is using divisibility:

- A_1 contains the strings xy^{3i}z for $i \geq 0$.
- A_2 contains the strings xy^{3i+1}z for $i \geq 0$.
- A_3 contains the strings xy^{3i+2}z for $i \geq 0$.

Note that the pumping lemma says that we can do this for any string in A with $|w| \geq p$, so this means we have three, disjoint, infinite sets. We do not yet know if we cover the full language A, since there may be other strings in A that use different loops or have length less than p. All of those strings can be put in one of the subsets, say A_1 (which makes most sense since it includes $i = 0$.

So, we have satisfied the first three properties. Finally, we need to argue that A_1, A_2 and A_3 are regular sets. We can ignore the finite strings that were added to A_1, since they are finite and must be regular, so only need to consider the strings $xy^{3i}z$ in A_1. We can construct a DFA that recognizes A_1 by taking the states corresponding to y in the DFA that recognizes A, and replacing the cycle with a new set of states (all are non-accepting) that correspond to going around the cycle three times. For example, if y is $q_{y1} \rightarrow_0 q_{y2} \rightarrow_1 q_{y3} \rightarrow_1 q_{y1}$ we would replace the transition from the state before q_{y1} with a transition to a new state, q_{y1a} which transitions to q_{y2a} on 0, which transitions to q_{y3a} on 1, which transitions to q_{yy1a} on 1 (instead of returning to q_{y1}, and so on, repeating the states, until the end of the third repetition where state q_{yyy3a} will transition back to state q_{y1}. To construct the machine to recognize A_2, we use the same idea, but go through a sequence of states that consume y before entering the loop. Similarly, A_3 can be recognizes by a machine that has a sequence of states that consume yy before entering the loop.

Problem 7: Language Operators. (4.17 / 10) Define a new operation on languages, $\mathcal{D}$, as:

$$\mathcal{D}(L) = \left\{ w | w \in \Sigma^* \text{ and } ww^R \in L \right\}$$

(where w^R denotes the reverse of w). Does $\mathcal{D}$ preserve regularity?

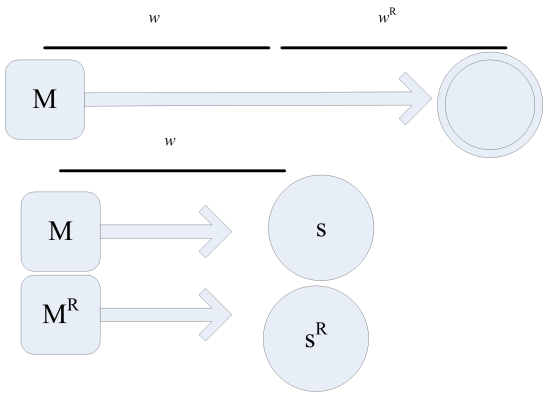
Answer:

Yes. This is a tricky question, since our intuition might mislead us into thinking $\mathcal{D}(L)$ is irregular since it seems similar to the language ww^R, which we know is irregular. In fact, however, we can construct a DFA M_D that recognizes $\mathcal{D}(L)$.

If we could manipulate the input, we could construct the M_D machine by changing the input from w to ww^R, and running M on the new input (as shown in the top part of the figure). If it accepts, then the string w is in $\mathcal{D}(L)$. But, we can't do that! A machine that could splice a copy of the input in reverse to the end of

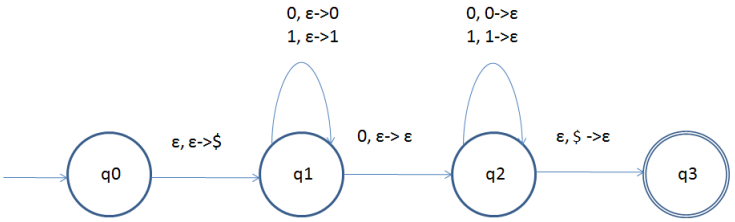
the input is much beyond the capabilities of a DFA, which can only consume the input one symbol at a time.

Instead, we can simulate processing w and w^R simultaneously. The basic idea is to combine M, the DFA that recognizes L, with M^R, the DFA that recognizes L^R (which we know exists since the class of regular languages is closed under reversal), into a single machine that simulates both simultaneously on the same input. Recall that the M^R machine will start in a state that is labeled with F, the set of accepting states of M. On the first input symbol, it goes to the state representing the set of states in M which would transition to an accepting state on that input symbol. At each point in processing the input substring z, it is in a set of states such that running M on z starting from any of those states would end in an accepting state. Thus, at the end of processing the input w, the machine M is in the state of M after procssing w, and M^R is in the state representing the set of states of M that would reach an accepting state if they processed w^R. If the state of the M machine matches any of the states represented by the state of M^R, then there is a path to an accepting state that goes forward the w, and then backward through w (that is, through w^R), so the string ww^R is in L. This means the combined machine recognizes the language $\mathcal{D}(L)$, showing it preserves regularity.



Problem 8: Nondeterminstic Pushdown Automata. (9.36 / 10) Draw a nondeterministic pushdown automaton that recognizes the language $\{w0w^R | w \in \{0,1\}^*\}$. The fewer states you use, the better.

Answer:



The NFA stays in state q_0 when reading w, pushing each input symbol on the stack. Every time we "guess" that we have reached the end of w and are beginning w^R by going to q_1 on any 0. Since its a nondeterministic machine, this transition is only actually followed on the path that leads to the accepting state. In state q_1 we pop off each 0 or 1.

Problem 9: Regular Grammars. (7.32 / 10) A *regular grammar* is a replacement grammar in which all rules have the form $A \to aB$ or $A \to a$ where A and B represent any variable and a represents a terminal. Prove that all regular languages can be recognized by a regular grammar.

Answer: The question is badly broken, but no one noticed! As defined in the question, a regular grammar cannot recognize the empty language! Thus, if you proved what the question asks for you proved a falsehood so there must be something broken in your proof.

The correct definition of a regular grammar also allows rules of the form $A \to \epsilon$. This is very necessary — without it, we could not produce all regular languages, including any language that contains ϵ.

With this correction, we can prove that all regular languages can be recognized by a regular grammar by showing how to construct such a grammar. If L is regular, there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes L. Our grammar will have a variable that represents each DFA state, and will map each transition rule to a grammar rule. The trick is figuring out how to deal with accepting states. Recall that a grammar produces a string only if all variables are replaced with terminals. Hence, for each variable that corresponds to a final state, we add a rule $V \to \epsilon$.

More formally, we construct a regular grammar $G = (V_G, \Sigma, R, S)$ that recognizes L:

$$V_G = Q$$
$$S = q_0$$
$$R_G = \{A \to aB | \forall A \in Q, a \in \Sigma : \delta(A, a) = B\} \cup \{A \to \epsilon | A \in F\}$$

Problem 10: Context-Free Grammars. Consider the grammar G below (S is the start symbol, and 0 and 1 are terminals):

$$S \quad \to \quad \epsilon$$

$S \rightarrow S\mathbf{00}$
$S \rightarrow \mathbf{11}S$
$S \rightarrow \mathbf{0}S\mathbf{1}$
$S \rightarrow \mathbf{10}S$

a. (1.92 / 2) Show that the string 111000 can be produced by G by showing a derivation that produces it.

Answer: $S \rightarrow 11S \rightarrow 1110S \rightarrow 1110S00 \rightarrow 111000$

b. (2.38 / 3) How many different derivations are there in G to produce 111000? (Support your answer with a clear argument.)

Answer: There are three possible derivations of 111000. The other two are:

$S \rightarrow S00 \rightarrow 11S00 \rightarrow 1110S00 \rightarrow 111000$
$S \rightarrow 11S \rightarrow 11S00 \rightarrow 1110S00 \rightarrow 111000$

We know there are no more possible derivations since the string starts with 11 (which can only be produced using the $S \rightarrow 11S$ rule). So, this rule must be used when there is an S at the left end of the string. We have two choices: using $S \rightarrow S00$ first, and then using $S \rightarrow 11S$; or, using $S \rightarrow 11S$ first. Any other productions put a 0 to the left of the leftmost S, so cannot produce a string that starts with 11. For the next step after starting with $S \rightarrow S00$, there is only one choice that works: using $S \rightarrow 11S$ to get $11S00$. The other three options produce strings that either cannot end in 1000 ($S \rightarrow S00$ produces $S0000$ which ends in four 0s) or that cannot start with 111 ($S \rightarrow 0S1$ and $S \rightarrow 10S$) put a 0 in the first or second symbol, with no variables before it. So, the only choice is $S \rightarrow 11S$, and from $11S00$ there is only one possible choice that works. We use a similar line of reasoning for the other option, starting with $S \rightarrow 11S$. For there we can eliminate the $S \rightarrow 11S$ option which would produce $1111S$ since it starts with four 1s, and the $S \rightarrow 0S1$ option, since it produces $110S1$ which cannot produce a string that starts with 111. The other two options work, as shown in the derivations. After the second step, there is only one choice for each following step.

c. (1.85 / 5) What is the fewest number of rules that can be added to G to produce a grammar that describes the language of all even-length strings in $\{0,1\}^*$? (An excellent answer would include the rules to add, a proof why it is not possible it use fewer rules, and a proof that your modified grammar produces all even-length strings.)

Answer: It is easy to see that adding these two rules can make the grammar cover all even-length strings:

$S \rightarrow 00S$
$S \rightarrow 01S$

With these two rules, and the given rules, we have covered all four possible two-symbol sequences before an S.

But, it is also possible to cover all even-length strings by only adding just one rule:

$S \to 0S0$

Proving that the resulting six rules can produce all even-length strings can be done using induction on the length of the string. The six rules (ordered by where S appears on the right side) are:

(1) $S \to \epsilon$
(2) $S \to \mathbf{S00}$
(3) $S \to \mathbf{0S1}$
(4) $S \to \mathbf{0S0}$
(5) $S \to \mathbf{10}S$
(6) $S \to \mathbf{11}S$

Basis. The rules can produce all strings of length 2. There are four possible strings of length two: 00, 01, 10, 11. Here is how to produce them: $S \to_4 S00 \to_1 00$; $S \to_3 0S1 \to_1 01$; $S \to_5 10S \to_1 10$; $S \to_6 11S \to_1 11$.

Induction. We need to prove that if we the grammar can generate all even-length strings of length k, it can generate all even-length strings of length $k+2$. We can prove this by showing all even-length strings of length $k+2$ can be generated using the rules, starting from all even-length strings of length k. Suppose s is some string of length $k+2$. There are two cases to consider:

(a) s starts with a 0. Then either $s = 0t0$ or $s = 0t1$ where t is some even-length string of length k. By the inductive hypothesis, we are assuming $S \to^* t$ since we assumed the grammar can produce all even-length strings of length k. So, there is some derivation $S \to_i \ldots \to t$ that produces t. We can add $S \to_4 0S0$ to the beginning of that derivation to produce $0t0$. We can add $S \to_3 0S1$ to the beginning of that derivation to produce $0t1$. This covers all even-length strings that start with a 0.

(b) s starts with a 1. Then either $s = 10t$ or $s = 11t$ where t is some even-length string of length k. Using the same reasoning as the previous case, we can produce both of these. The first, by adding $S \to_5 10S$ to the beginning of the derivation that produces t, the by adding $S \to_6 11S$ to the beginning of the derivation that produces t.

Note that no part of our proof used rule 2, so that rule is unnecessary. We have proved that rules 1, 3, 4, 5, and 6 by themselves can generate all possible even-length strings.