

CMSC 330
Fall 2006
Homework 1 - Answers

1. Regular expressions

(a) Any sequence of symbols that has a 01 at the end. Any sequence of symbols is just $(0|1)^*$ so answer is:

$$(0|1)^*01$$

(b) A number is divisible by 5 if it ends in a 0 or 5. First digit cannot be 0 (except for the integer 0) so answer is:

$$(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*(0|5)$$

First (...) ensures string has a non-zero first digit, (...)* gives any number of intervening digits, and (0|5) says string ends in 0 or 5.

However, answer forces all strings to be at least 2 digits, thus we miss the single digit answers of 0 and 5. So full answer is:

$$((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*(0|5)) | (0|5)$$

(c) From <http://www.ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/syntax.html>: “alphabets, decimal digits, and the underscore character, and begin with a alphabets(including underscore). There are no restrictions on the lengths of Ruby identifiers. So:

$$(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_)(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_0|1|2|3|4|5|6|7|8|9)^*$$

[Should also write $a|A|b|B|c|C \dots$ to include upper case as well. Note that this grammar also includes certain reserved words like BEGIN, END, for, if, else, The above expression accepts those to also be “identifiers,” but are not really allowed to be so. Eliminating those from the expression still makes the language a regular expression, but makes it more complex. How would you do that?]

(d) Want an odd number of 0s or an odd number of 1s. Here is a case where the ambiguity of English makes the problem partially ambiguous. By saying an odd number of 0s or an odd number of 1s, we also include the case where both are true (odd number of 0s and 1s).

In class it was discussed that the set of DFA and the set of regular expressions were the same, and that there exist algorithms to convert from one to the other. This problem is an example where the

DFA is trivial to write (see answer to 2(d)), but the regular expression is more complex. The string is either repeated pairs of 0s interspersed with as many 1s as desired following by a single 0 with additional 1s, or the opposite, a repeated pair of 1s interspersed with as many 0s as desired followed by a single 1 with additional 0s:

$$(1^*01^*01^*)^*01^* | (0^*10^*10^*)^*10^*$$

(e) In order to contain the embedded string 101, the string must contain anything following by 101 followed by anything or:

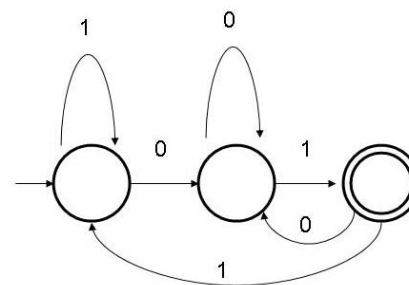
$$(0|1)^*101(0|1)^*$$

(f) Like 1(d), the DFA is easier to write than the regular expression (See 2(f)). In order to not contain 101, following a 1 it must be followed by a 00 or else it will have 101. So answer is:

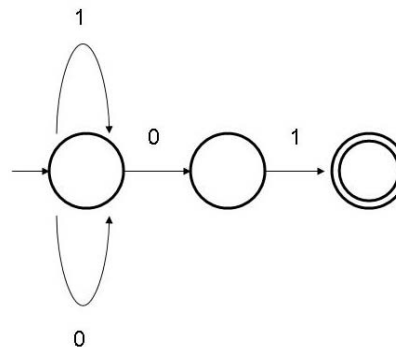
$$(0 | 11^*00)^* 1^*0^*$$

2 DFA

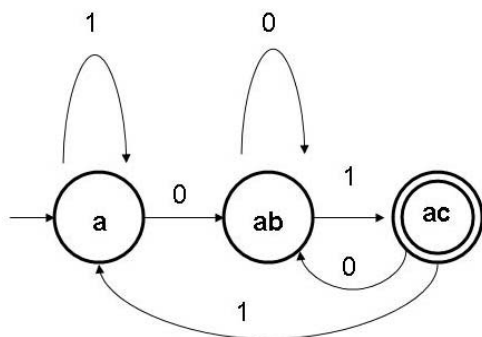
(a)



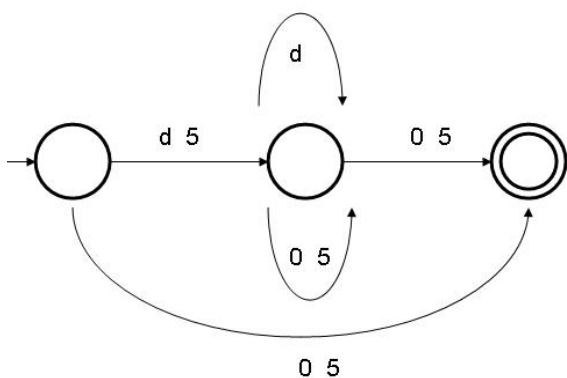
You can also take the answer to 1(a) and convert it to a NFA:



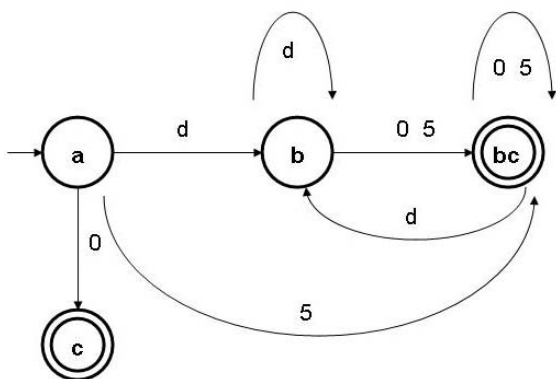
Labeling the 3 states a, b, and c from left to right and converting this to an equivalent DFA (which is just the first one given above):



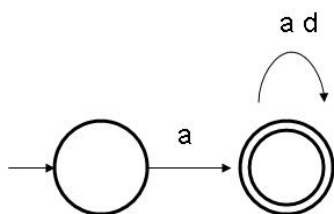
(b) Let d stand for any digit 1 through 9 except 5, we can give NFA directly from answer to 1(b):



This NFA, converted to an equivalent DFA is just:

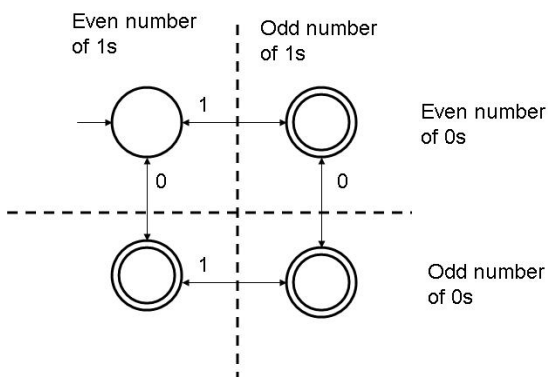


(c) Let a stand for any letter or underscore ($_$) and d stand for any digit 0 through 9. The NFA is just:

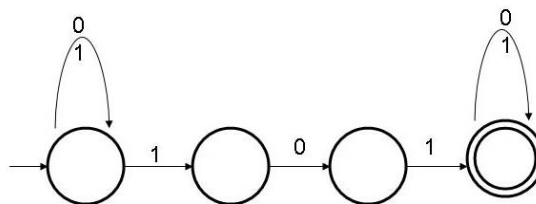


Converting this simple NFA to DFA is left to you.

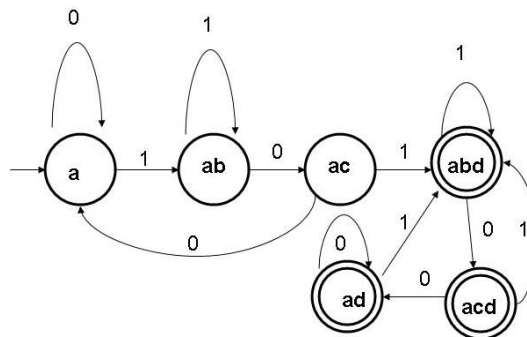
(d) As stated in the answer to 1(d), this is a trivial DFA. You need to keep track of 2 binary decisions – have you seen an odd or even number of 0s and have you seen an odd or even number of 1s? Keep track of the 0s by the “vertical” states below and the number of 1s by the “horizontal” states. The DFA is just:



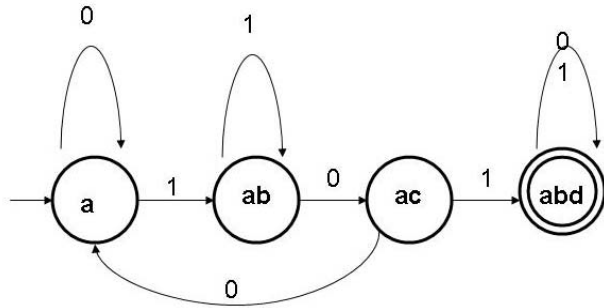
(e) NFA is just:



which is just the DFA:



It should be clear that once you are in the final state abd , you can never leave, so a simpler solution is:

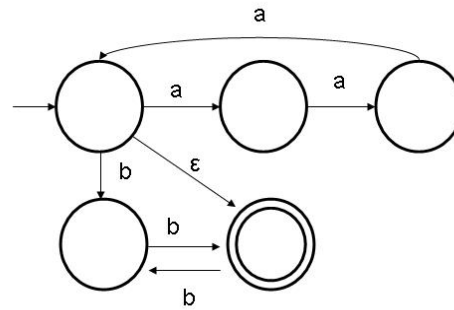


3. There are 2 paths to the final state on the right and 1 path to final state on row 2 of NFA. So regular expression is:

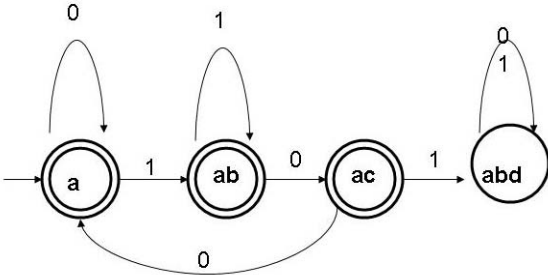
$$(0^*11^*01^*)|(0^*11^*100^*11^*)|(0^*11^*1)$$

4. (a) $(aaa)^*(bb)^*$

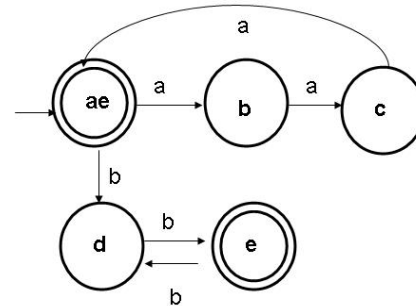
(b) NFA is then:



(f) We will accept all strings except the ones accepted by answer 2(e). So the answer is just the same DFA as the previous one, with the accepting and non-accepting states reversed.

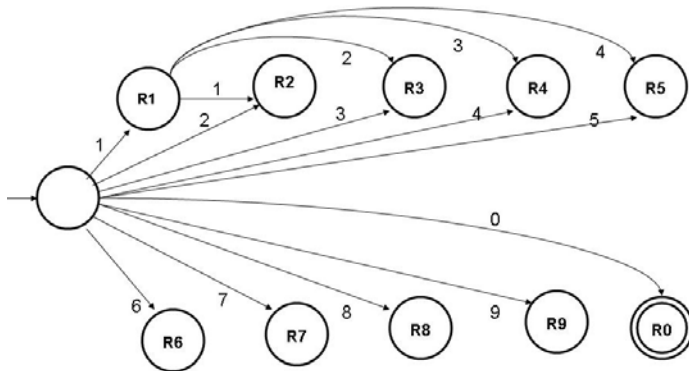


DFA is then:



(g) A number is divisible by 9 if the sum of the digits equals 9. So as we read digits, all we have to do is keep track of the remainders mod 9 and the final state is the one with a 0 remainder (0 mod 9).

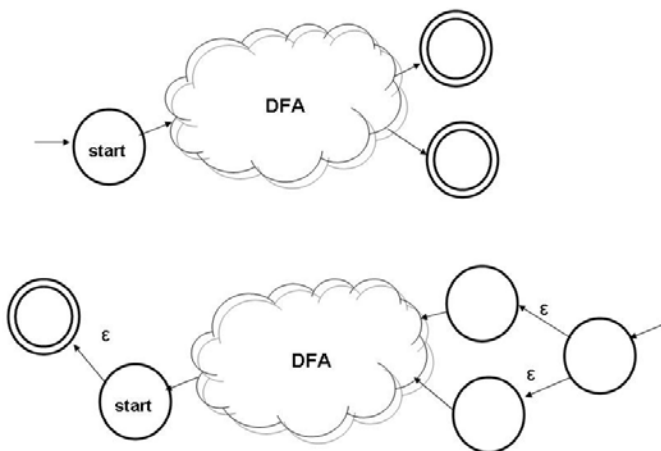
The DFA is easy, but a bit messy. The start of the DFA for start state and R1 is given as:



The rest of the DFA is given as: If in state R_n and input is m , arc goes to state R_x where $x = (m+n) \bmod 9$.

(c) Although every string in L is generated by a^*b^* , when we say that a regular expression generates a set L (or a regular expression accepts a set L), we mean that the regular expression accepts those strings in L and no other. In this case, a^*b^* accepts L , but also accepts many other strings (e.g., ab , aab , $aabbb$) which are not in L .

5. Assume S is recognized by the first DFA pictured below. S^R is S "run backwards", as pictured in the second DFA below. Construct S^R as follows:



$S^{1/2}$ works as follows: The first member of the pair simulates S . The second member of the pair says there is some transformation that goes to a final state. Since there are an equal number of transformations in the first pair as in the second, if the state (x, Y) where x is in Y is reached, that says that there is some string of length equal to x which ends up in a final state of S .

Since $S^{1/2}$ is an NFA, we can construct the DFA that accepts the same set.

- (1) Reverse all the arcs in S , so each transformation goes to previous state.
- (2) Make all the final states in S to be non-final.
- (3) Add a new start state with epsilon moves to all the previous final states of S . (That is the backwards machine can start in any of the final states of S .)
- (4) Add an epsilon arc in the former start state to a new state, which is now the final state of the backwards machine.

This machine will now accept S^R . It starts in the former final states of S , and any time it passes through the former start state, it can use the epsilon transfer to accept the string, if that is the end of the string.

It should be clear that S^R is a NFA. So use our algorithm to convert the NFA to a DFA and you get a DFA that accepts S^R .

6. Construct NFA for $R^{1/2}$ as follows:

- (1) Let S be a DFA.
- (2) Let S^R be the NFA constructed for S -reverse as given in answer to problem 5.
- (3) Construct a new NFA $S^{1/2}$ as follows. The states of the NFA will be pairs (X, Y) where X is a state of S and Y is a subset of states of S^R .
- (4) The start state of $S^{1/2}$ is the pair (a, b) where a is the start state of S and b is the start state of S^R .
- (5) The transformations for S^R are as follows:
Arc d takes state (x, y) to state (p, q) if
 - There is an arc labeled d from x to p in S
 - There is some arc from y to q in S^R
- (6) The final states are the states (x, Y) where x is in Y