Sample Problems for Lexical Analysis

1. Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$:

   (a) The set of all strings representing a binary number that is not a multiple of 4_{10} (4 in base-10).

   (b) The set of all strings in which the sequences 000 and 111 do not occur.

   (c) The set of all strings representing a binary number that is greater than 8_{10} (8 in base-10).

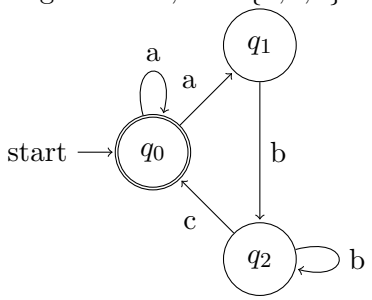
2. Draw DFAs for the languages defined in parts (b) and (c) of question 1.

   (b) The set of all strings in which the sequences 000 and 111 do not occur.

   (c) The set of all strings representing a binary number that is greater than 8_{10} (8 in base-10).

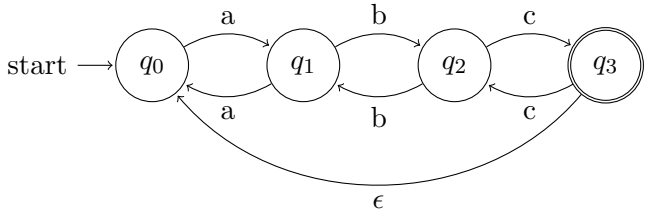
3. Using the techniques covered in class, transform the following NFAs with ϵ-transitions over the given alphabet Σ into DFAs. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account for your transformations. Hint: Is there a subset of states the NFA transitions to when fed a symbol for which the set of current states has no explicit transition?
Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state s of the DFA maps to the set of states Q of the NFA such that an input string stops at s in the DFA if and only if it stops at one of the states in Q in the NFA.

(a) Original NFA, $\Sigma = \{a, b, c\}$



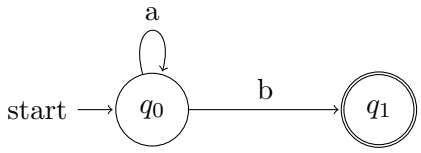
DFA:

(b) Original NFA, $\Sigma = \{a, b, c\}$



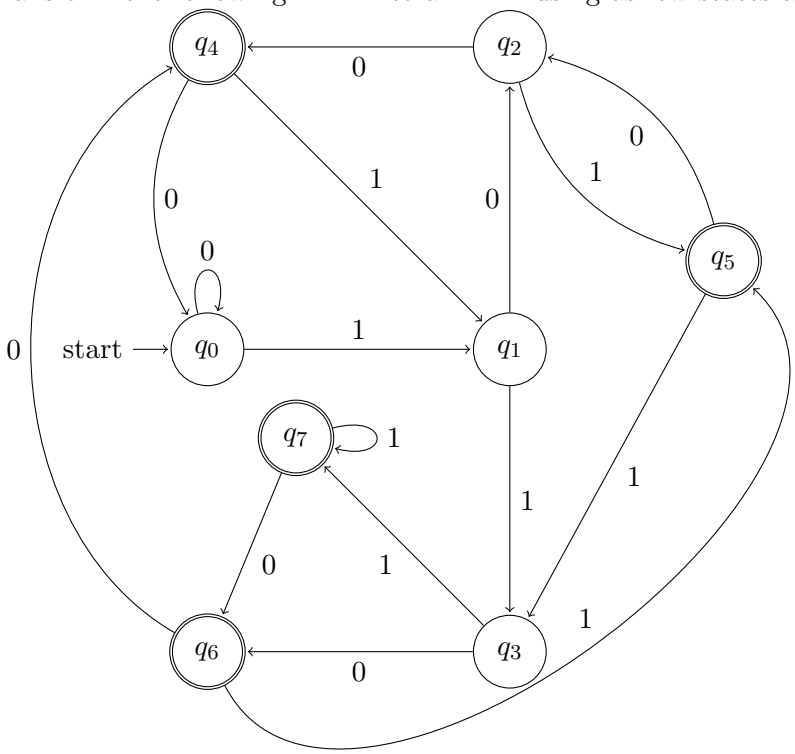
DFA:

(c) Original NFA, $\Sigma = \{a, b\}$



DFA:

4. Transform the following DFA into an NFA using as few states as possible:



NFA:

5. Consider the following tokens and their associated regular expressions, given as a **flex** scanner specification:

```
%%
(001|110)                 printf("cat")
(0100*|1011*)             printf("eat")
(00*1100*|11*0011*)       printf("dog")
```

Give an input to this scanner such that the output string is $(\texttt{cat eat})^{12}\texttt{dog}$, where $\texttt{A}^{\texttt{i}}$ denotes `A` repeated `i` times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer.

6. Recall from the lecture that, when using regular expressions to scan an input, we resolve conflicts by taking the largest possible match at any point. That is, if we have the following **flex** scanner specification:

```
%%
do                          { return T_Do; }
[A-Za-z_][A-Za-z0-9_]*      { return T_Identifier; }
```

and we see the input string "dot", we will match the second rule and emit T_Identifier for the whole string, not T_Do.

However, it is possible to have a set of regular expressions for which we can tokenize a particular string, but for which taking the largest possible match will fail to break the input into tokens. Give an example of a set of regular expressions and an input string such that: a) the string can be broken into substrings, where each substring matches one of the regular expressions, b) our usual lexer algorithm, taking the largest match at every step, will fail to break the string in a way in which each piece matches one of the regular expressions. Explain how the string can be tokenized and why taking the largest match won't work in this case.

Answer: