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).

\[ (1(0|1) \ast (11|10|01)) \mid (11|10|1) \]

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

\[(\epsilon | 1|11)(01|001|011|0011)^*(\epsilon | 0|00)\]

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

\[ 1(0|1)^*(0|1)(0|1)(0|1)(0|1) \mid 1(0|1)(0|1)1 \mid 1(0|1)1(0|1) \mid 11(0|1)(0|1) \]
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 $\epsilon$-transitions over
the given alphabet $\Sigma$ 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, $\Sigma = \{a, b, c\}$

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

Corespondences (DFA to NFA):
- $q_0$ to $\{q_0\}$
- $q_1$ to $\{q_1\}$
- $q_2$ to $\{q_2\}$
- $q_3$ to $\{q_0, q_3\}$
- $q_4$ to $\{\}$

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

DFA:

Corespondences (DFA to NFA):
- $q_0$ to $\{q_0\}$
- $q_1$ to $\{q_1\}$
- $q_2$ to $\{\}$
4. Transform the following DFA into an NFA using as few states as possible:
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 \((\text{cat eat})^2 \text{dog}\), where \(A^i\) 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.

\[(0010101101010110001100) \text{ or } (1101010001010)110011\]
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:

```flex
%%
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:** Consider the following scanner:

```flex
%%
a { return A; }
aba { return B; }
bab { return C; }
```

and the string “abab”. This can be broken into ‘a’, followed by ‘bab’. However, the largest
possible match strategy will first consume the beginning of the string as ‘aba’, then stop when
it finds that the remainder of the input is just ‘b’, which can’t be matched to any token.