# CS143 Spring 2024 – Written Assignment 1

Due Thursday, April 18, 2024 11:59 PM PDT

This assignment covers regular languages, finite automata, and lexical analysis. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by 11:59 PM PDT on the due date. Please review the course policies for more information: `http://web.stanford.edu/class/cs143/policies/index.html`. A LaTeX template for writing your solutions is available on the course website. To create finite automata diagrams, you can either use the TikZ package directly by following the examples in the template, or a tool like `https://madebyevan.com/fsm/`.

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

   (a) The set of strings beginning with 0 and having an odd numbered length.
   **Solution:**
   $$0((0|1)(0|1))^*$$

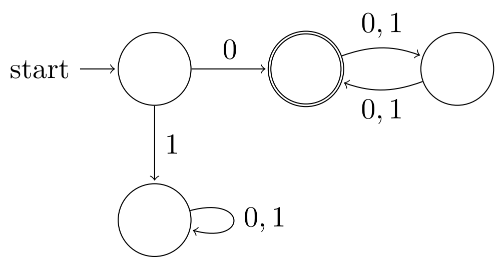
   (b) The set of all strings that contain less than three 1's.
   **Solution:**
   $$0^*\,1?\,0^*\,1?\,0^*$$

   (c) The set of strings where 0's and 1's appear in alternating groups of odd numbered length. Examples of strings in the language: $\varepsilon$, $\underline{0}$, $\underline{0}\,\underline{111}$, $\underline{000}\,\underline{1}\,\underline{000}$, and $\underline{1}\,\underline{0}\,\underline{1}\,\underline{000}$. Examples of strings not in the language: 11000, 111011, and 00).
   **Solution:**
   $$(0(00)^*)?\,(1(11)^*\,0(00)^*)^*\,(1(11)^*)?$$

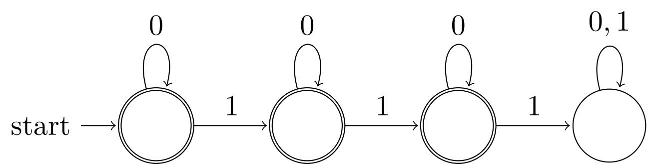
2. Draw DFAs for each of the languages from question 1. Note that a DFA must have a transition defined for every state and symbol pair. You must take this fact into account for your transformations. Your DFAs should not have more than 10 states. Submissions with unnecessarily complex DFAs may not receive full credit.

   Notice that a short regular expression does not automatically imply a DFA with few states, nor vice versa.

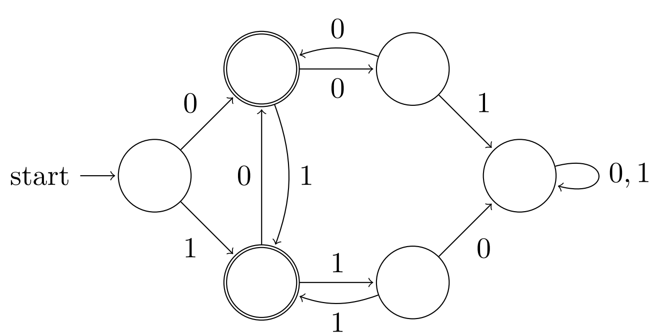
   (a) The set of strings beginning with 0 and having an odd numbered length.
   **Solution:**




   (b) The set of all strings that contain less than three 1's.
   **Solution:**




   (c) The set of strings where 0's and 1's appear in alternating groups of odd numbered length.
   **Solution:**

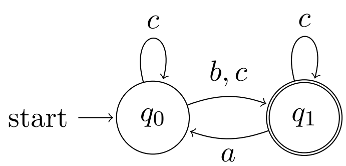


3. Using the techniques covered in class, transform the following NFAs over the alphabet $\{a, b, c\}$ into DFAs. Your DFAs should not have more than 10 states. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account 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?

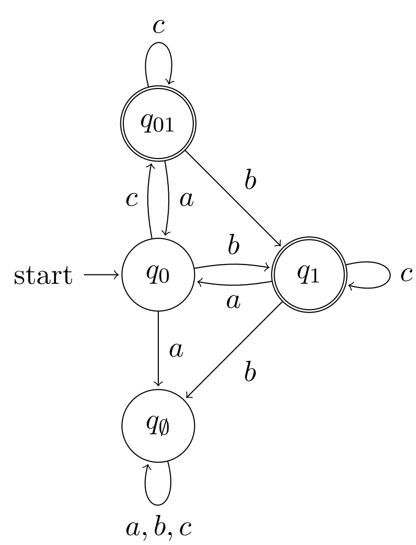
   Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state $q$ of your DFA maps to the set of states $Q$ of the NFA such that an input string stops at $q$ in the DFA if and only if it stops at one of the states in $Q$ in the NFA.

   Tip: for readability, states in the DFA may be labeled according to the set of states they represent in the NFA. For example, state $q_{012}$ in the DFA would correspond to the set of states $\{q_0, q_1, q_2\}$ in the NFA, whereas state $q_{13}$ would correspond to set of states $\{q_1, q_3\}$ in the NFA.

   (a)



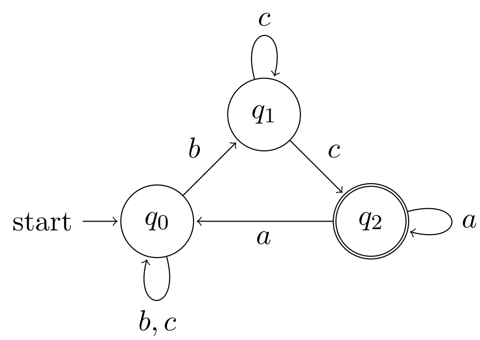

**Solution:**



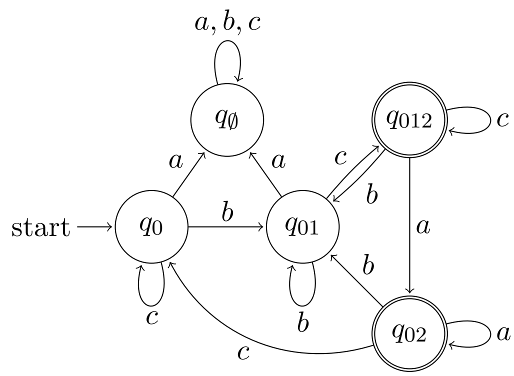

Corresponding states (DFA to NFA):

- $q_0 : \{q_0\}$
- $q_1 : \{q_1\}$
- $q_{01} : \{q_0, q_1\}$
- $q_\emptyset : \{\}$

(b)



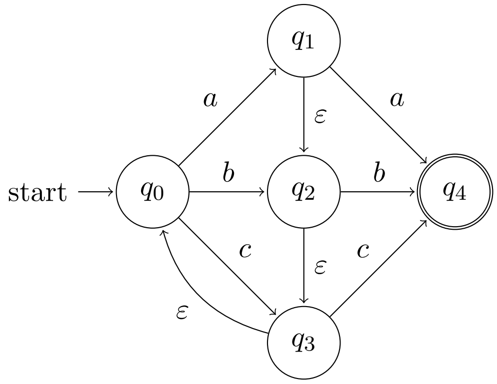

**Solution:**



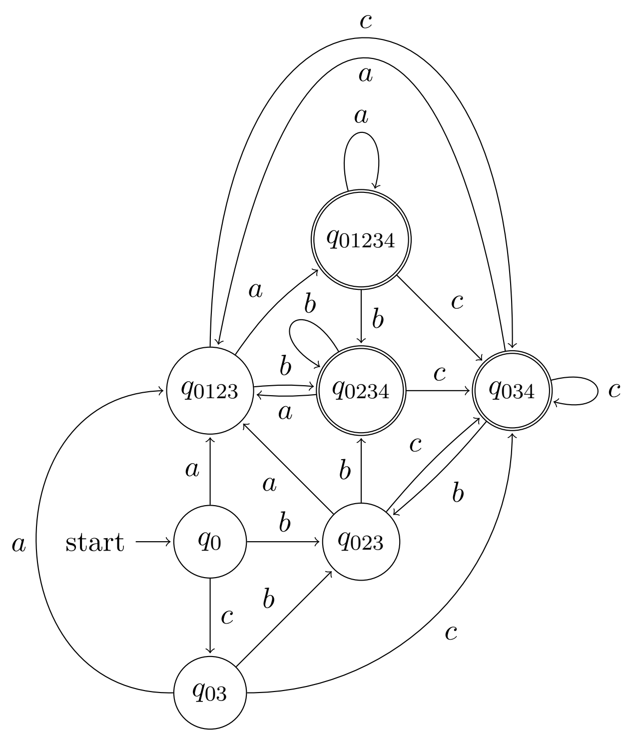

Corresponding states (DFA to NFA):

- $q_0 : \{q_0\}$
- $q_{01} : \{q_0, q_1\}$
- $q_{02} : \{q_0, q_2\}$
- $q_{012} : \{q_0, q_1, q_2\}$
- $q_\emptyset : \{\}$

(c)




**Solution:**




Corresponding states (DFA to NFA):

- $q_0 : \{q_0\}$
- $q_{0123} : \{q_0, q_1, q_2, q_3\}$
- $q_{023} : \{q_0, q_2, q_3\}$
- $q_{03} : \{q_0, q_3\}$
- $q_{01234} : \{q_0, q_1, q_2, q_3, q_4\}$
- $q_{0234} : \{q_0, q_2, q_3, q_4\}$
- $q_{034} : \{q_0, q_3, q_4\}$

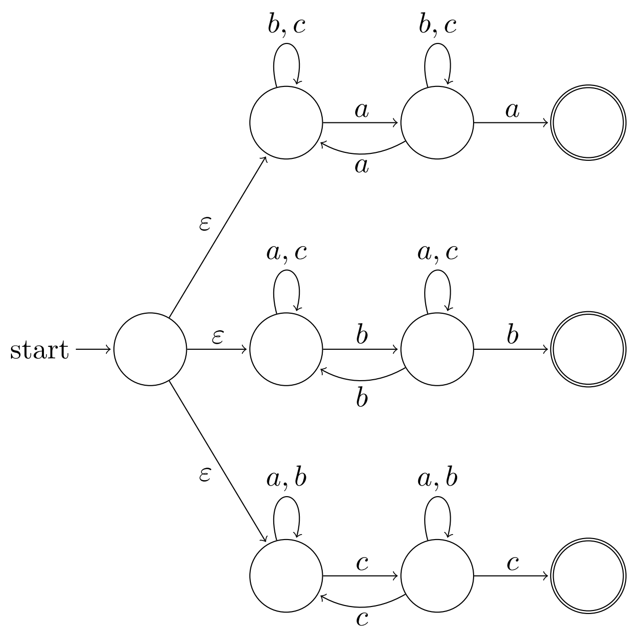
4. Let $L$ be a language over $\Sigma = \{a, b, c\}$, such that string $w$ is in $L$ if and only if $w \neq \varepsilon$ and the last character in $w$ appears an even number of times in $w$

   Examples of strings in $L$: $aa$, $abacaba$, $bab$.

   Examples of strings **not** in $L$: $\varepsilon$, $a$, $abcabca$, $bbb$.

   Draw an NFA for $L$. Your solution should have no more than 15 states.

   **Solution:**



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

```
%%
(01|10)                          printf("apple");
(10)?0?                          printf("banana");
(0110+|1001*1)                   printf("coconut");
```

Give an input to this scanner such that the output string is $((\texttt{apple})^3(\texttt{banana})^4)^2((\texttt{apple})^3\texttt{coconut})^2$, where $\texttt{A}^i$ denotes $\texttt{A}$ repeated $i$ times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer.

**Solution:**

$$((01)^3(0)^4)^2\,(01)^3\,0110\,(10)^3 1001$$

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 no more than two 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.

As a challenge (not necessary for credit), try to find a solution that only uses one regular expression.

**Solution**: Consider the following scanner for a JavaScript-like language that has both `==` and `===` operators:

```
%%
==        { return T_Equal; }
===       { return T_StrictEqual; }
```

and the string "`====`". This can be broken into '`==`', followed by '`==`'. However, the largest possible match strategy will first consume the begining of the string as '`===`', then stop when it finds that the remainder of the input is just '`=`', which can't be matched to any token.