CS143 Spring 2026 – Written Assignment 1

Due Thursday, April 16, 2026 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 Ti*k*Z package directly by following the examples in the template, or a tool like `https://madebyevan.com/fsm/`.

1. Write regular expressions *and* DFAs that recognize the following languages over the alphabet $\Sigma = \{0, 1\}$. Your DFAs must contain no more than 4 states.

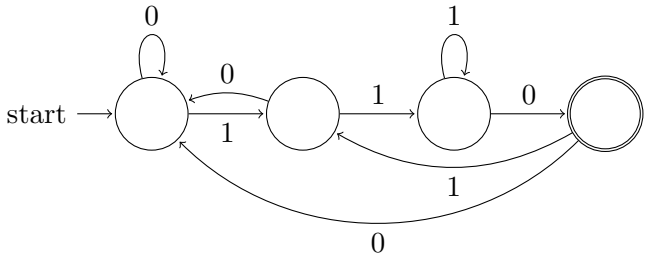
   (a) The set of strings that end with 110.

   Solution:

   Regular expression:

   $$(0|1)^*110$$

   DFA:

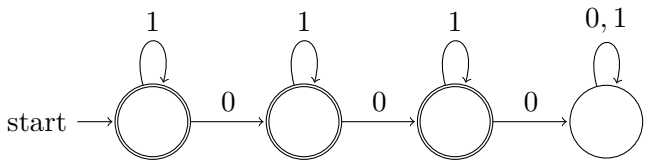
   (b) The set of strings that contain less than three 0's.

   Solution:

   Regular expression:

   $$1^*0?1^*0?1^*$$

   DFA:

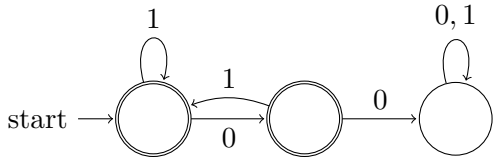
(c) The set of strings that do not contain two or more consequent 0's.

Solution:

Regular expression:

$$1^*(01^+)^*0?$$

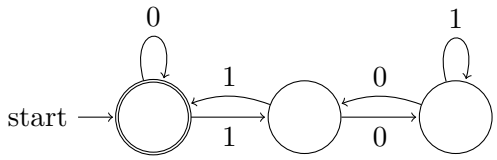
DFA:



(d) The set of strings that, when interpreted as a binary number, is a multiple of 3 (as an edge case, the empty string shall be interpreted as number 0, which is a multiple of 3). **Hint:** For this subproblem, it might be easier to draw the DFA first, and then construct the regular expression by considering what paths are possible in the DFA to reach the accept state.

Solution:

DFA:



Regular expression:

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

The three nodes in the DFA, from left to right, represent that the current binary number has reminder 0,1,2 modulo 3 respectively. The transition edges follow naturally by how the reminder changes if we add one extra digit at the end.

From the DFA, one can see that any valid path can be separated into zero or more round trips from the starting node to itself (thus the final `*` in the regex), where each round trip is:

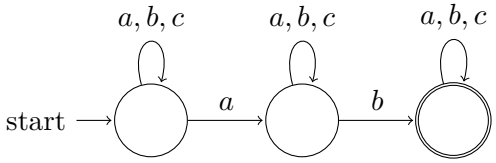
- Either the self-edge via character `0`,
- Or a transition to the middle node via character `1`, followed by zero or more round trips through the right node (where each round trip can be described by 01*0), and finally transition back to the starting node via character `1`. Thus, the overall regex for this case can be described by 1(01*0)1.

This results in our final regular expression (0|1(01*0)*1)*.

2. Write NFAs that recognize the following languages over the alphabet $\Sigma = \{a, b, c\}$.

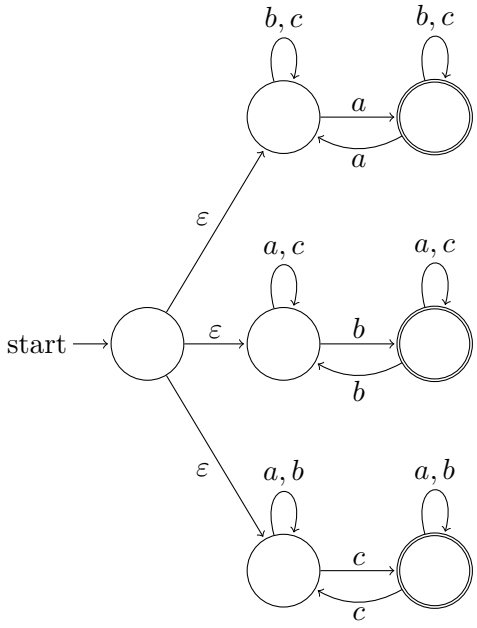
   (a) The language L where a string w is in L iff there exists $i < j$ such that $w[i] =$`'a'` and $w[j] =$`'b'`. For example, `ab`, `bacb` are in L, where `ba`, `ac` are not. Your NFA should have no more than 3 states.

   Solution:



   (b) The language L where a string w is in L iff some character $x \in \{a, b, c\}$ appears an odd number of times in w. For example, `ab` is in L, but `aa` is not. Your NFA should have no more than 8 states.

   Solution:

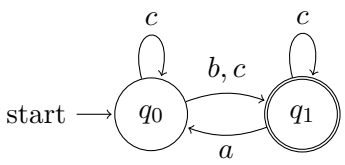


3. Using the techniques covered in class, transform the following NFAs over the alphabet $\{a, b, c\}$ into DFAs. Your DFAs should not have more than 10 states. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account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

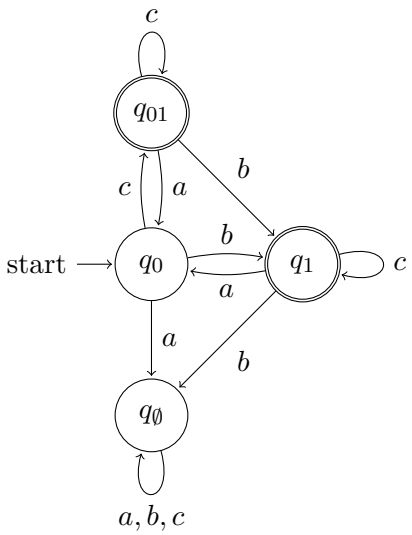
   Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state q of your DFA maps to the set of states Q of the NFA such that an input string stops at q in the DFA if and only if it stops at one of the states in Q in the NFA.

   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)



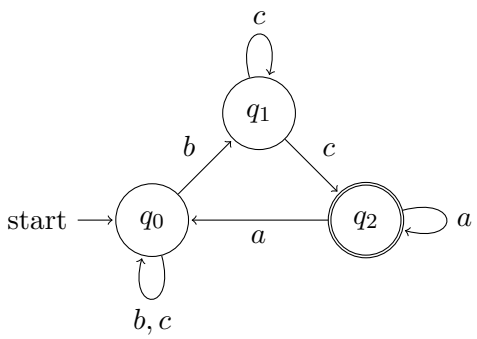
   Solution:



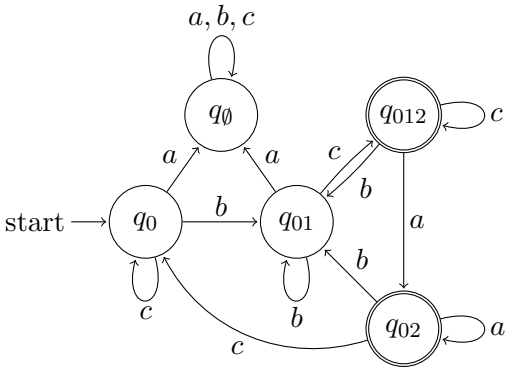
   Corresponding states (DFA to NFA):

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

(b)



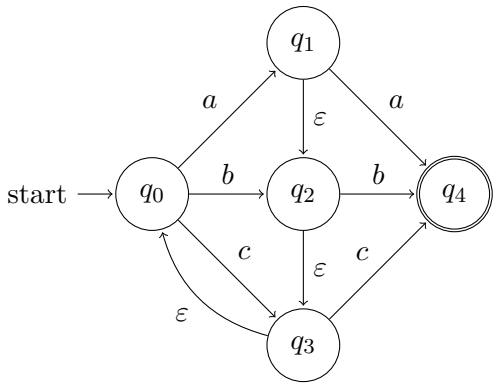
Solution:



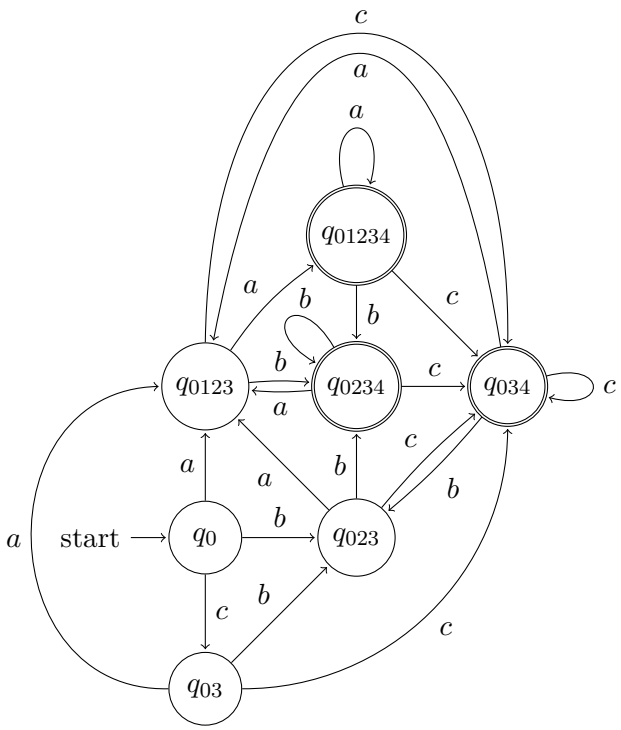
Corresponding states (DFA to NFA):

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

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

```
%%
0                       printf("a");
(10|01)                 printf("b");
(101|011+)              printf("c");
```

Give an input to this scanner such that the output lexeme sequence is `aacbbca`.

For ease of grading, please use underlines to show the parts of your input string that correspond to each lexeme, for example, 10 0 0111.

Solution:

0 0 011 01 01 011 0

Any solution that uses `01` for `b` and `011+` for `c` is correct. Note that `10` for `b` and `101` for `c` cannot be used here, since flex always looks for the longest possible match.

5. 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.