CS143 Spring 2021 - Written Assignment 1

Thursday, April 15, 2021 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. Please review the the course policies for more information https://web.stanford.edu/class/cs143/policies/. A LaTeX template for writing your solutions is available on the course website. There is a post on Piazza describing how to create the finite automata diagrams.

1. Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$. Hint some of these languages may include ϵ.

   (a) The set of all strings which do not end with 11.

   (b) The set of all strings representing a binary number where the sum of its bits is odd.

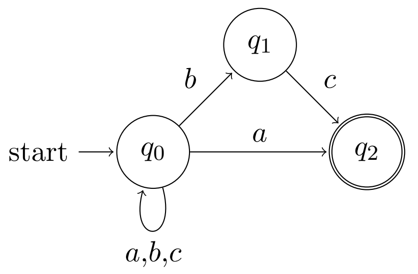
   (c) The set of all strings where all characters are repeated at least once where ever they appear in the string (e.g. 00 or 000111100 but not 00100).

2. Draw DFA's 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.

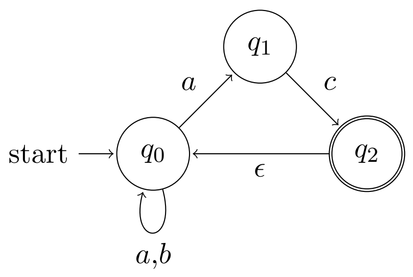
3. Using the techniques covered in class, transform the following NFAs over the alphabet $\{a, b, c\}$ into DFAs. Your DFAs should not have more than 10 states. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account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

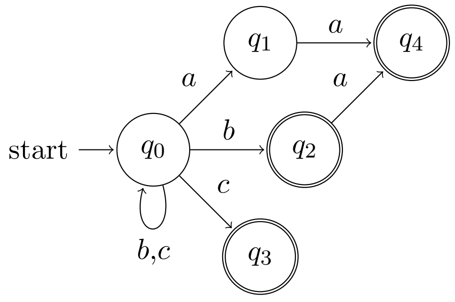
   Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state s of your DFA maps to the set of states Q of the NFA such that an input string stops at s in the DFA if and only if it stops at one of the states in Q in the NFA.

   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)

(b) 

(c) 

4. Let L be a language over $\Sigma = \{a, b, c\}$ where the following holds:

   String w is in L if and only if w is of the form $w = x^n s$ where $|x| = 1$, $n \geq 0$, and s is a string that does not contain x as a substring. Here, x^n denotes x being repeated n times. You can imagine w as a string with a head consisting of a character repeated 0 or more times and said character does not appears in the tail of w.

   Examples of strings in L: $\underline{\text{c}}$abab, $\underline{\text{bbb}}$, $\underline{\text{bb}}$aaac, baab

   Examples of strings **not** in L: $\underline{\text{b}}$aaa$\underline{\text{b}}$c, $\underline{\text{c}}$ab$\underline{\text{c}}$ab

   Draw an NFA for L. Your solution should not have more than 10 states.

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

```
%%
(01|10)                    printf("apple");
1(01)*0                    printf("banana");
(1011*0|0100*1)            printf("coconut");
```

   Give an input to this scanner such that the output string is ($\texttt{apple}^3\texttt{banana}$)^3($\texttt{apple coconut}$)^2, 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.