CS143 Compilers - Written Assignment 1

Due Thursday, April 19, 2018 at 11:59 PM

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 should be submitted electronically through Gradescope as a PDF by 11:59 PM PDT. 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.

Note 1: Make sure to sign up for a Gradescope account and add CS143 (entry code: M5VE22) if you have not done so already. Some students in the past have waited until the last minute to create an account only to run into issues and then been unable to submit the assignment in time. Don't let this happen to you!

Note 2: Although the written assignments are similar to those of previous offerings of CS143, the problems themselves change from year to year.

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

   (a) The set of all strings which start and end with the same digit.

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

   (c) The set of all strings that contain the substring 10100.

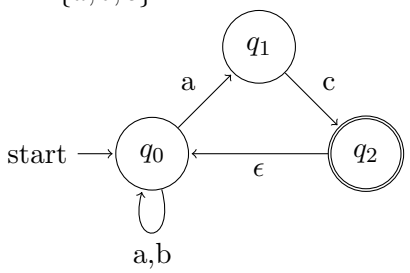
2. Draw DFAs for each of the languages from question 1.

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?

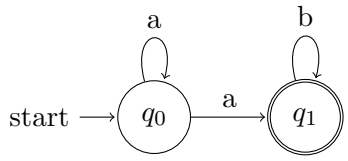
   Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state s of your DFA maps to the set of states Q of the NFA such that an input string stops at s in the DFA if and only if it stops at one of the states in Q in the NFA.

   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.

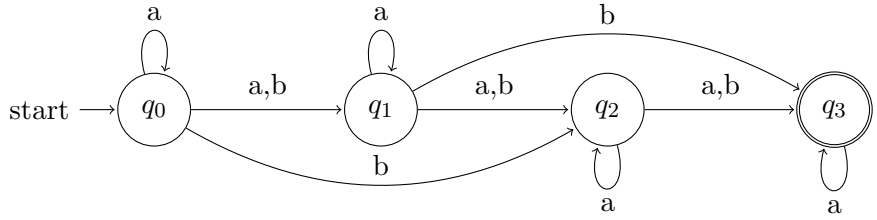
   (a) $\Sigma = \{a, b, c\}$



(b) $\Sigma = \{a, b\}$



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



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

   String w is in L iff w is of the form $w = sx^n$ where $|x| = 1$, $n \geq 1$, 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 tail consisting of a character repeated at least once and said character appears nowhere else in w.

   Examples of strings in L: abab<u>c</u>, <u>bbb</u>, aaac<u>bb</u>

   Examples of strings **not** in L: ϵ, <u>b</u>aaa<u>b</u>, ab<u>c</u>ab<u>c</u>

   Draw an NFA for L.

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 $(\texttt{apple}^3\texttt{banana})^5\ \texttt{coconut}^2$, 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.