CS143 Spring 2025 – Written Assignment 1

Due Thursday, April 17, 2025 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/`.

Questions on assignments or exams that ask for regular expressions should be written using the five operators introduced in lecture 3 (in particular, even though the set of regular languages is closed under negation, there is no notation for negation available). Other notation is fine so long as you concisely define it first (e.g. $A? := A + \varepsilon$ is ok).

1. Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$. **Note:** the next question will ask you to write DFAs for each of these languages. Starting by writing down a DFA/NFA may or may not be easier than the regular expression.

   (a) The set of strings in which the substring 111 does *not* appear. Example strings in the language: ε, 1101, 0101100011.

   Solution:

   (b) The set of strings in which the first, fourth, seventh, ... (continuing for every third) character, if present, is a 1. Example strings in the language: 1001, 110100, ε.

   Solution:

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) **Solution:**

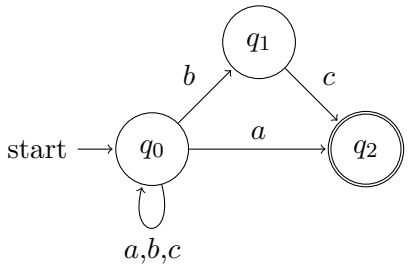
   (b) **Solution:**

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 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.

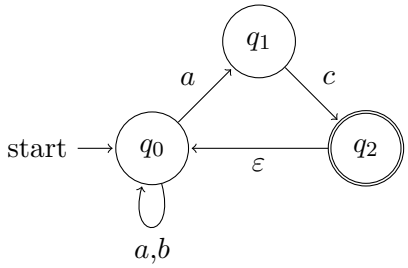
   Tip: for readability, states in the DFA may be labeled according to the set of states they represent in the NFA. For example, state q_{012} in the DFA would correspond to the set of states $\{q_0, q_1, q_2\}$ in the NFA, whereas state q_{13} would correspond to set of states $\{q_1, q_3\}$ in the NFA.

   (a)



   Solution:

   (b)



   Solution:

4. Starting next week, we will discuss parsing and context free languages. The language over $\Sigma = \{0, 1\}$ consisting of "all strings with an unequal number of 0s and 1s" turns out to be an example of a language that is context free but not regular (there is no DFA that recognizes it exactly).

   (a) Give a regular expression for an **infinite subset** of this language (you can pick any subset you like, so long as it is regular and contains an infinite number of different strings).

   (b) Give a regular expression that matches a **superset** of this language but does not match *every* string in Σ^* (that is, $(0 + 1)^*$ is not a valid answer).

(*both parts have short answers that do not depend on any knowledge of context-free grammars*)

5. Consider the following program written in a hypothetical C-like language:

```
while get() == 0:
    pass
halt
```

This language only has a few features:

values Every value is either 0 or 1.

expressions

- Literals for 0 and 1.
- Each call to `get()` asks the environment (e.g. a human user) to input 0 or 1 and returns the response.
- The `==` operator evaluates to 1 if its two arguments are the same and 0 otherwise.

statements

- The `pass` statement does nothing but proceed to the next statement.
- The `halt` statement ends the execution.
- To evaluate the `while` construct, we evaluate its condition expression and check if it is 1; if it is, we execute the body and return to the condition (unless a halt instruction was reached in the body); if the condition is 0, we continue past the body to the next statement. We re-evaluate the condition for each iteration. Syntactically, loop bodies are indented once.

(this spec might not be fully precise, but we aren't trying to trick you; it's just a simple language with loops, input, and a 'halt' instruction that behave how you would expect)

When running this example program, there are three types of execution traces that might arise:

- The user *stops responding*; in this case, the program gets stuck and does not reach a halt instruction (example *stuck input*: 000).
- The user responds in such a way that the program continues to transition between states but never reaches a halt instruction (example *non-terminating input*: 0000...).
- The user gives an input which reaches a halt instruction; these are called *terminating inputs* (examples: 1, 01, 001). Note that after reaching a halt instruction, no further input is possible, even if there are more statements following it.

The set of terminating inputs for this particular program form a regular language, specifically 0^*1. For each of the following three programs, give a regular expression that matches the set of terminating inputs for it (and no other strings):

(a)
```
while get():
    halt
while get():
    pass
halt
```

(b)
```
while (get() == get()) == 0:
    pass
halt
```

(c)
```
while get() == 0:
    while get() == 1:
        pass
halt
```

Completely optional food for thought: Is it possible to write a program in this language whose terminating inputs are *not* a regular language? If not, could you prove it by writing a program that *compiles* an input program into its regex? What if the language also had variable expressions and assignment statements that could store a boolean value?

6. 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 $((\texttt{apple})^3\texttt{banana})^3((\texttt{apple})\texttt{coconut})^2$, where $\texttt{A}^i$ 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.

Solution:

7. Recall from the lecture that, when using regular expressions to scan an input, we resolve conflicts greedily 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 an optional challenge (no extra credit), try to find a solution that only uses one regular expression.

Solution: