Module 6 Review: Questions on Regex and DFA

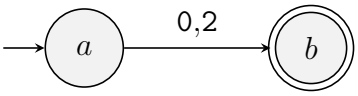
Module 6 is our first view on a computational perspective of language. Namely, we are examining the idea of input being recognized or accepted by a language. If we take programming languages as an example, for those of you who have some experience in programming, you know that when you want some command to be executed in a program, it has to be written correctly. If not, it is immediately 'rejected', and your code doesn't compile! Before a computer understands what you want it to do, it has to be able to recognize what you are telling it.

Questions:

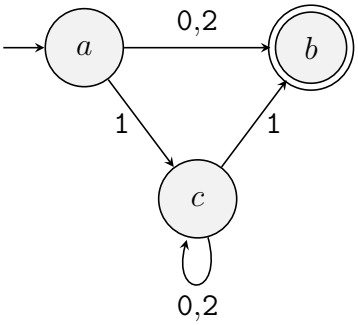
1. Considering the regular expression `([02]|1[02]*1)+` over the alphabet $\Sigma = \{0, 1, 2\}$:

   - Construct a DFA that accepts the same language as the regular expression.
   - (Slightly tricky) In one English sentence, describe the language accepted by both representations.

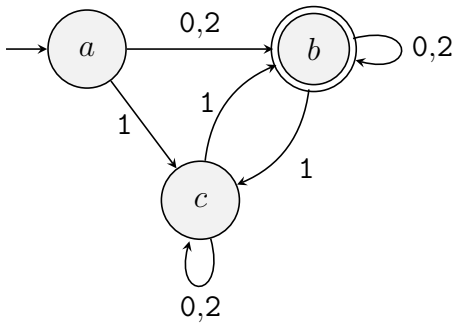
   A great place to start in creating our DFA to is notice that a single input of 0 or 2 is accepted. That means we can can draw an arrow from a start state to an accepting state and label it with 0,2.

   Now, we may notice that taking that path is the same as taking the path indicated by "1[01]*1" How would we represent that? We need two 1s, and in between, we can have as many 0s or 2s as we like, even none!

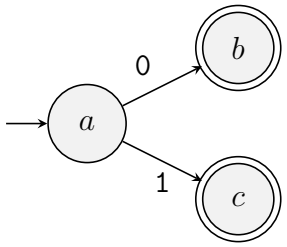
   Almost there! Now, how do we take care of the +? Well, if we have a single 0 or 2, we still accept the string, and if we have 1, we need a second one to move back to the accepting state. Thus,



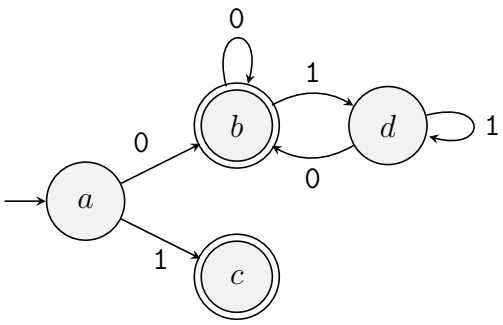
That's our DFA! If you're wondering what the accepted language is, take a look at the regex; we can have 0 and 2 wherever we like, but we need to always have 1s in groups of two. Precisely, the language is all strings for which the sum of all digits in the string is even.

2. Given the alphabet $\Sigma = \{0, 1\}$, construct a regular expression and equivalent DFA that accept the language of all strings that begin and end with the same character.

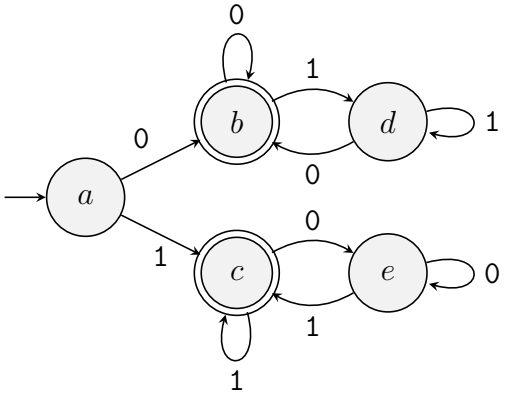
   The regex for this one follows from what the language specifies. As long as we start and end with 1 or start or end with 0, we can have whatever we want in the middle. So, we might have something like `1[01]*1|0[01]*0`. However, this misses two sneaky cases: when we have exactly one character. So, we add them to get `1[01]*1|0[01]*0|0|1`. For the DFA, we know that we have to deal with 1 and 0 separately, but if we have a single input of either one of them, we reach an accepting state. Thus,



   Now, if we start with a 0 and continue to input 0s, we will still be in an accepting state! However, if we input any number 1 after we start the string with 0, we need to add another 0 to the end of the string for it to be accepted. So,

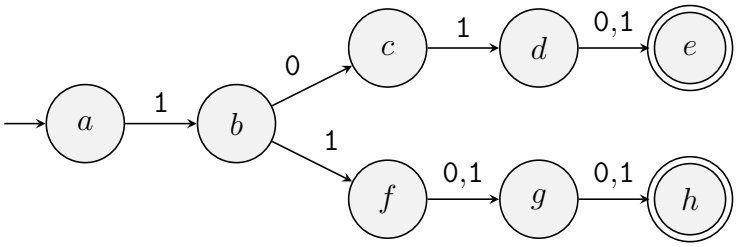


We're almost done! Now, for the bottom branch, we are almost doing the same thing, except that we switch the position of the 0s and 1s. Therefore,

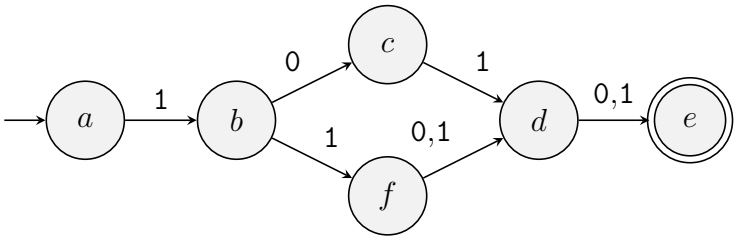


3. Considering the regex `(1(01|1[01])[01])+` over the alphabet $\Sigma = \{0, 1\}$, construct a DFA that accepts the same language. Also, try to determine what language the DFA accepts (it's not at all obvious initially!).

   A thing that I like to do when I get a regex and need to create the equivalent DFA is to first 'wire up' the main accepting string. I see that my string must start with a 1. Then, it must be followed by 0 then 1 or 1 then whatever I want. Finally, I can have single 0 or 1. Now, my string is accepting! Let's draw that out:

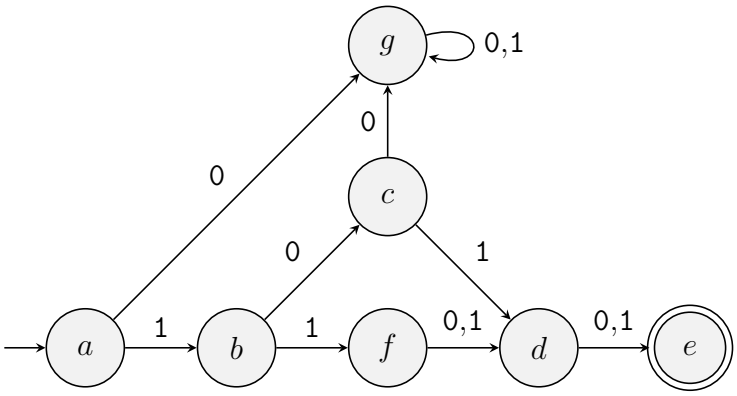


   Since the transitions from d to e and g to h are basically the same thing, I will condense those to avoid clutter (I will connect f to d instead):

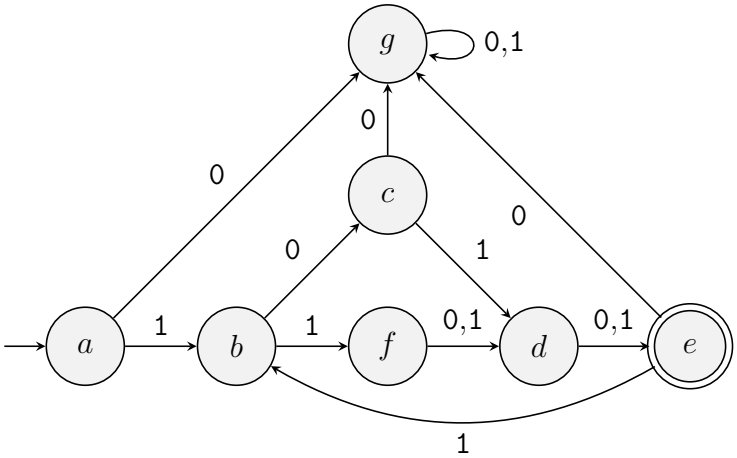


   Since the regex only accepts that type of input, it means that every other deviating input that is not what we already have in our DFA should be rejected (we will take

care of the accepting state later). Where does this happen? Well, at a, we don't have 0, and at state c, we don't have 0. Let's draw a garbage state and send everything else there:



Almost there! Now, for the accepting state: What should happen there? Well, by the regex, we accept any string such that we take the string types like the ones we just passed through the DFA, and repeat only those types as many times as we want. We know that if we started our string with 0, it would immediately be rejected. So, if we are in a accepting state, and we add a 0, that would now also be rejected. If we have a 1, that is fine; we would just go back to the state where we had just inputted a single 1. Thus,



Now, the language. This is a bit tricky, but if you notice closely, we only accept strings in their length is a multiple of 4. Given that observation and the fact our alphabet consisted of 0s and 1s, what could that mean (hint - module 3!)? Perhaps, we're working with strings of 4-bit binary numbers? If so, what kind of numbers are allowed? Well, the leftmost bit must be 1, so that means numbers 8 to 15. To the right of that, we accept strings where the numbers follow the pattern of 101X or 11XX.

What do those numbers have in common? They are all greater than 9! That's one way to view the language - strings of multiple 4-bit binary numbers that are all greater

than 9. If you are keen and can see a second pattern, perhaps you might be thinking in a different number system (hexadecimal, maybe?). Since we don't use numerals in hexadecimal to represent numbers greater than 9, these strings are also special. We could also say that this language is strings of 4-bit binary numbers whose hexadecimal representations are solely made up of letters.

4. (On your own time) Considering the regex `(01[01]|1[01]1)+` over the alphabet $\Sigma = \{0, 1\}$, construct a DFA that accepts the same language. Also, try to determine what language the DFA accepts (it's not at all obvious initially! Try binary!).

5. (On your own time) Given the alphabet $\Sigma = \{0, 1, 2\}$, construct a regular expression and equivalent DFA that accepts the language of all strings that contain "210" as a substring, where the position of the 0 in at least one occurrence of "210" is a multiple of 3 (the first character in the string is at position 1).

    - For example, strings like "21$\underline{0}$0000", "110201210", "221210221" are accepted, whereas strings like "11210112", "2022210" are rejected.