These problems are due Friday at the start of class, on paper. Feel free to ask questions. You may want to do more problems than these. For example, you may want to practice the product construction with an explicit example, or read the book's proof of the union property, which uses the product construction. I leave that extra practice up to you. In other words, the assigned problems here should be regarded as a *minimal* amount of work, to understand the material.

A. Do problem 1.5c from our textbook.

B. Do problem 1.6j from our textbook.

This next problem is about checking the validity of an HTML file. You don't need to know HTML. (It might be better if you didn't.) To keep the problem manageable, we use a highly simplified version of HTML, and we ignore all of the text in the file other than the tags. The HTML file is regarded as a string over the 14-symbol alphabet

$$\Sigma = \{\texttt{html}, \texttt{/html}, \texttt{head}, \texttt{/head}, \texttt{title}, \texttt{/title}, \texttt{body}, \texttt{/body}, \texttt{ul}, \texttt{/ul}, \texttt{li}, \texttt{/li}, \texttt{a}, \texttt{/a}\}.$$

We consider the file to be valid if and only if it meets the following criteria. (Later in this course, we will develop a more precise method of stating such criteria.)

- The file must begin with `html` and end with `/html`. These symbols cannot occur anywhere else in the file.
- Immediately after `html` there must be a *header*, which begins with `head` and ends with `/head`. These two tags cannot occur elsewhere.
- The header must either contain no tags at all, or must contain a single *title*. A title begins with `title`, ends with `/title`, and contains no other tags. These two tags cannot occur elsewhere.
- Immediately after the header there must be a *body*, which begins with `body` and ends with `/body`. These two tags cannot occur elsewhere.
- The body may contain any number (0 or more) of *unordered lists*. An unordered list begins with `ul` and ends with `/ul`.
- An unordered list may contain any number of *list items*. A list item begins with `li` and ends with `/li`.

- Within any list item there can occur any number of `a` tags. Within the body, but outside any unordered lists, there can occur any number of `a` tags. Whenever an `a` tag occurs, the next tag must be a `/a` tag.

C. Draw a DFA that accepts valid HTML and rejects invalid HTML, as just defined. To keep your drawing clear, please adopt the following convention. If a state q has no outgoing transition arrow for a symbol a, then it is understood that the machine rejects its input when it is in state q and sees symbol a.

For our final problem we need to agree on a uniform way of describing DFAs in Python. Let us adopt the convention that the states of a DFA are numbered $q_0, q_1, \ldots, q_{n-1}$, where n is the number of states and q_0 is not necessarily the start state. Similarly, let's agree that the symbols are numbered $a_0, a_1, \ldots, a_{m-1}$. Then one can specify a DFA in Python by the following data.

1. A list `delta` of lists, such that each list in delta has the same length. Let n be the length of `delta` and m the length of each list in `delta`; these are the number of states and the number of symbols, respectively. This `delta` is the table for the DFA's transition function δ.

2. A number `s` belonging to the set $\{0, 1, \ldots, n-1\}$ to indicate the start state.

3. A list `F` of numbers from $\{0, 1, \ldots, n-1\}$, with no repeats, to indicate the final states.

We have not specified the set Q of states or the alphabet Σ, but we know how big each is from the structure of `delta`, and we know how to compute with them because the states and symbols are uniquely identified as numbers in $\{0, 1, \ldots, n-1\}$ and $\{0, 1, \ldots, m-1\}$, respectively. Thus the data `delta`, `s`, and `F` encode everything essential to the DFA.

The input string to a DFA on m symbols shall be represented in Python as a list of numbers from the set $\{0, 1, \ldots, m-1\}$. For example, the string $a_3a_3a_1a_0a_5$ shall be represented as [`3`, `3`, `1`, `0`, `5`]. Finally, our Python DFA will output `True` or `False` rather than `Accept` or `Reject`, because the former are more Pythonic.

D. Write a Python function `dfa` that simulates a given DFA on a given input. That is, `dfa` takes in four arguments — `delta`, `s`, `F`, and `input` — and outputs either `Accept` or `Reject`, according to whether the DFA described by `delta`, `s`, `F` would accept or reject when given the input `input`.