1. Let $\Sigma=\{s,S,t,o\}$. Write an FSA equivalent to the Strictly 2-Piecewise language that forbids the following subsequences of length two $\{Ss,Ss\}$ but permits all others. Test your fsa with the `recognizes` function defined in `lesson10.ml` on the following words with the following code (assuming you have bound your fsa to the name `fsaSP`):

   ```ocaml
   recognizes "ss" fsaSP ;; (* should be true *)
   recognizes "tsttttttttttOst" fsaSP ;; (* should be true *)
   recognizes "tsttttttttttSt" fsaSP ;; (* should be false *)
   recognizes "tSttttttttttSt" fsaSP ;; (* should be true *)
   recognizes "" fsaSP ;; (* should be true *)
   recognizes "oooooooooooooop" fsaSP ;; (* should be false *)
   ``

2. Let $\Sigma=\{s,S,t,o\}$. Build an FSA which accepts only words whose number of $\{s\}$ and $\{S\}$ sum to an odd number. Sounds $\{s,S\}$ are sibilant sounds so this means the number of sibilants must be odd. Test your fsa with the `recognizes` function defined in `lesson10.ml` on the following words with the following code (assuming you have bound your fsa to the name `fsaSP`):

   ```ocaml
   recognizes "ss" fsaODD ;; (* should be false *)
   recognizes "tsttttttttttttOst" fsaODD ;; (* should be true *)
   recognizes "tstttttttttttSt" fsaODD ;; (* should be true *)
   recognizes "tStttttttttttSt" fsaODD ;; (* should be true *)
   recognizes "" fsaODD ;; (* should be false *)
   recognizes "oooooooooooooop" fsaODD ;; (* should be false *)
   ``

3. Prefix Trees. A prefix tree is a finite-state acceptor that accepts a finite set of words. For example, Figure 1 shows a prefix tree which accepts the words $\{be, beak, beer, best, cat, catch\}$. Define a function `make_pt` which takes a finite set of strings and returns a prefix tree. In other words this function should have the following signature.
val make_pt : StringSet.t \rightarrow fsa = <fun>

Make a prefix tree with the set \{be, beak, beer, best, cat, catch\} and test it as follows.

\begin{verbatim}
  recognizes "be" fsaPT ;; (* should be true *)
  recognizes "beak" fsaPT ;; (* should be true *)
  recognizes "bear" fsaPT ;; (* should be true *)
  recognizes "beer" fsaPT ;; (* should be true *)
  recognizes "best" fsaPT ;; (* should be true *)
  recognizes "cat" fsaPT ;; (* should be true *)
  recognizes "catch" fsaPT ;; (* should be true *)
  recognizes "bet" fsaPT ;; (* should be false *)
  recognizes "cow" fsaPT ;; (* should be false *)
  recognizes "quickly" fsaPT ;; (* should be false *)
\end{verbatim}

4. Sequential Transducers. Acceptors can be thought of mapping strings to the values \{true, false\}. Probabilistic acceptors can be thought of mapping strings to real numbers between zero and one. Transducers can be thought of as mapping strings to other strings. The sequential transducer in the figure converts all strings with a substring “mt” to a substring “mp” (nasal assimilation). The label [x:y] on the transition means “read character x and write string y.” Let’s see how this transducer works. What happens when you give it the string atamta. In the derivation below [x:y] is written \( \frac{y}{x} \).

\[
\begin{array}{c}
1 \xrightarrow{a} 1 \\
1 \xrightarrow{t} 1 \\
1 \xrightarrow{m} 1 \\
1 \xrightarrow{m} 1 \\
2 \xrightarrow{t} 1 \\
1 \xrightarrow{a} 1 \\
\end{array}
\]

Note that sequential transducers have no final states (equivalently, every state is considered final). Also note that while reading happens one character at a time, what may be written can be any string. It just so happens that in our example, those strings are of length 1.

Using the code for acceptors and probabilistic acceptors as guides, implement sequential transducers in Ocaml. This includes defining a type as well as defining a recognize_st function.

Hint: In lesson11.ml, transform_prob computes the running probability as it goes along the path. You will want to compute the running string instead.

Implement the transducer above and verify that it works as intended.

BONUS Let \( \Sigma=\{p,a,b\} \). Write a sequential transducer that converts all [p]s to [b]s that are between two [a]s. For example, the string appapa maps to appaba. Hint: to get apa to write as aba, try writing the empty string after reading p. In our notation (with arbitrary state names):

\[
\ldots A \xrightarrow{a} B \xrightarrow{\lambda}_p C \ldots
\]

What happens next is up to you! Once you have written the transducer to convert substrings apa to aba, answer this question: what does this transducer map appap to?