These lecture notes are provided as a supplement to the textbook. In the exercises/problems section of Chapter 1, the textbook defines Finite State Transducers (FST) as deterministic automata that at each step read one input symbol $a \in \Sigma$ and write one output symbol $b \in \Gamma$. In these notes we define a more general form of FST that can output arbitrary strings at each step, a feature often useful in applications.

1 Defining FST

Definition 1 A Finite State Transducer (FST) is a 5-tuple $M = (Q, \Sigma, \Gamma, \delta, s)$ where

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of input symbols,
- $\Gamma$ is a finite set of output symbols,
- $\delta: Q \times \Sigma \rightarrow Q \times \Gamma^*$ is the transition function,
- $s \in Q$ is the start state.

As all automata, an FST keeps track of an internal state $q \in Q$, initially set to $q = s$, which may change at each step of the computation. During the computation, the FST reads a string $w \in \Sigma^*$ over the input alphabet $\Sigma$, and outputs a string $u \in \Gamma^*$ over a possibly different alphabet $\Gamma$. The input symbols are read one at a time. When the machine is in state $q \in Q$ and reads the symbol $a \in \Sigma$, it moves to a new state $p \in Q$ and produces a string $w \in \Gamma^*$ obtained by evaluating the transition function $(p, w) = \delta(q, a)$. The output of the computation is obtained by concatenating the strings produced at every step. So, the behavior of an FST is described by a function $f_M: \Sigma^* \rightarrow \Gamma^*$ mapping the input string $w \in \Sigma^*$ to an output string $f_M(w) \in \Gamma^*$.

In order to formally define the output of an FST, we first extend the transition function $\delta$ to a function $\delta^*_M: Q \times \Sigma^* \rightarrow Q \times \Gamma^*$ that takes arbitrary strings (over the alphabet $\Sigma$) as input, rather than single symbols.

Definition 2 Let $M = (Q, \Sigma, \Gamma, \delta, s)$ be an FST. The extended transition function $\delta^*(q, w)$ is defined by induction on the length of $w$ as follows:

- Base case ($|w| = 0$): for every $q \in Q$, let $\delta^*(q, \epsilon) = (q, \epsilon)$

- Inductive case ($|w| > 0$): for every $a \in \Sigma$ and $w' \in \Sigma^*$, let $\delta^*(q, aw') = (q'', u''w'')$ where $(q', u') = \delta(q, a)$ and $(q'', u'') = \delta^*(q', w')$. 

The output of $M$ on input $w$ is defined as $f_M(w) = u$ where $(q, u) = \delta^*(s, w)$ for some $q \in Q$.

We say that a function $f: \Sigma^* \rightarrow \Delta^*$ is FST-computable, if it can be computer by an FST, i.e., there is an FST $M$ such that $f(w) = f_M(w)$ for all $w \in \Sigma^*$.

## 2 Composing FSTs

Any two functions $f: \Sigma^* \rightarrow \Gamma^*$ and $g: \Gamma^* \rightarrow \Delta^*$ can be combined using the standard function composition operation $(g \circ f): \Sigma^* \rightarrow \Delta^*$ defined as $(g \circ f)(w) = g(f(w))$. At this point, it is natural to ask: if two functions are FST-computable, is their composition also FST-computable? Notice that this is not a completely trivial question: given two FST-computable functions $f_{M_1}$ and $f_{M_2}$ (say, computed by FSTs $M_1$ and $M_2$), evaluating $f_{M_2}(f_{M_1}(w))$ requires the computation of an intermediate result $f_{M_1}(w)$ which may be just too long to be stored by an FST. So, we cannot simply apply the two functions in sequence as you would do using a general purpose programming language. In order for $f_{M_2} \circ f_{M_1}$ to be FST-computable, we need to be able to run $M_1$ and $M_2$ at the same time, and process the input string $w$ in a streaming fashion. The following theorem shows how to do that.

**Theorem 1** For any FST $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, s_1)$ and $M_2 = (Q_2, \Gamma, \Delta, \delta_2, s_2)$, there is an FST $M = M_2 \circ M_1$ such that $f_M = f_{M_2} \circ f_{M_1}$.

**Proof** Let $M = (Q, \Sigma, \Delta, \delta, s)$ where $Q = Q_1 \times Q_2$, $s = (s_1, s_2) \in Q$ and $\delta: Q \times \Sigma^1 \rightarrow Q \times \Gamma^*$ is the function defined as $\delta((q_1, q_2), a) = ((q_1', q_2'), v)$ with $(q_1', u) = \delta_1(q_1, a)$ and $(q_2', v) = \delta_2(q_2, u)$. It can be easily verified by induction that $f_M = f_{M_2} \circ f_{M_1}$. \qed

We remark that in order to compose two FSTs $M_2 \circ M_1$, the output alphabet of $M_1$ must match the input alphabet of $M_2$. The intuition behind the above construction is the following. The FST $M_2 \circ M_1$ works by running $M_1$ on the input string $w \in \Sigma^*$ to obtain some intermediate result $u \in \Gamma^*$. As $M_1$ outputs $u$, the composed automaton $M_2 \circ M_1$ runs the second FST on $u$ to obtain the final output string $v$. Since finite automata (and FST in particular) do not have enough memory to store the intermediate result of the computation $u$, the two component automata $M_1$, $M_2$ are run at the same time, and the output of $M_1$ is fed to $M_2$ while it is being produced. In order to run the two automata at the same time, we use the cartesian product $Q_1 \times Q_2$ as the set of states of the composite automaton. Each state $(q_1, q_2) \in Q$ records the current state of $M_1$ and the current state of $M_2$. When a symbol $a \in \Sigma_1$ is read from the input, we first invoke the transition function of $M_1$ to obtain $(q_1', u) = \delta_1(q_1, a)$. This is the step performed by $M_1$ on input $a$, and it means “move to state $q_1'$ and output $u$”. In the composed automaton, the string $u$ represents not a final output, but an intermediate value to be passed to the next automaton. Accordingly, the composite automaton runs $M_2$ on input $u$, starting from the current state $q_2$, to obtain an updated state and output string $(q_2', v) = \delta_2(q_2, u)$. Notice that since the intermediate output $u \in \Gamma_1^*$ is not just a symbol, but a string, we need to use the extended transition function $\delta_2^*$. 
3 FST-reductions

The following theorem shows that the preimage of a regular language under an FST-computable function is also regular.

**Theorem 2** For any FST-computable function \( f: \Sigma^* \rightarrow \Gamma^* \) and any regular language \( B \subseteq \Gamma^* \), the language \( A = f_T^{-1}(B) = \{ w \in \Sigma^* | f(w) \in B \} \) is also regular.

**Proof** Let \( M = (Q, \Gamma, \delta, s, F) \) be a DFA such that \( L(M) = B \), and let \( T = (Q_T, \Sigma, \Gamma, \delta_T, s_T) \) be an FST. We combine \( M \) and \( T \) into a DFA \( M' = (Q \times Q_T, \Sigma, \delta', (s, s_T), F \times Q_T) \) where \( \delta'((q, q_T), a) = (\delta^*(q, w), q_T') \) for \( (q_T', w) = \delta_T(q_T, a) \). The language \( f_T^{-1}(B) \) is regular because it is the language of the DFA \( M' \). □

Notice that, by definition, the requirement \( A = f^{-1}(B) \) is equivalent to the condition “\( w \in A \iff f(w) \in B \)”. (As usual, you can rewrite this double implication as two separate properties “\( w \in A \implies f(w) \in B \)”, and “\( w \notin A \implies f(w) \notin B \).”) A function \( f: \Sigma^* \rightarrow \Gamma^* \) satisfying these properties is called a reduction from \( A \) to \( B \). The name reduction comes from the fact that you can think of \( f \) as a method to translate (membership) questions about \( A \) to (membership) questions about \( B \). So, the task of determining if \( w \in A \) is reduces to the task of determining if \( f(w) \in B \). For simplicity of exposition, below, we assume all languages are over some fixed alphabet \( \Sigma \).

**Definition 3** For any two languages \( A, B \subseteq \Sigma^* \), a reduction from \( A \) to \( B \) is a function \( f: \Sigma^* \rightarrow \Sigma^* \) such that for all \( w \in \Sigma^* \),

- if \( w \in A \) then \( f(w) \in B \), and
- if \( w \notin A \) then \( f(w) \notin B \).

A reduction \( f \) is FST-computable if it is computed by a Finite State Transducer \( T \). We say that \( A \) is FST-reducible to \( B \) (in symbols \( A \leq_{FST} B \) if there is an FST-computable reduction \( f \) from \( A \) to \( B \).

Reductions are one of the most important concepts in the study of the theory of computation, and we will encounter many other flavors of reductions later on. But for now, we will keep the discussion focused on regular languages. The notation \( A \leq B \) (read “\( A \) reduces to \( B \)”)
 can be interpreted as a comparison between the “hardness” of the two problems. Think of problems that can be solved by a DFA (i.e., regular languages) as being “easy”, and problems that cannot be solved by a DFA (i.e., nonregular languages) as being hard. The closure property proved in Theorem 2 can be reformulated as follows.

**Corollary 1** If \( A \leq_{FST} B \) and \( B \) is regular, then \( A \) is also regular.

Informally, if \( A \) is not harder than \( B \) and \( B \) is “easy”, then you can conclude that \( A \) is also easy. By taking the contrapositive, we also get that if \( A \) is not harder than \( B \) and \( A \) is hard, then you can conclude that also \( B \) must be hard.
Corollary 2 If $A \leq_{FST} B$ and $A$ is not regular, then $B$ is also not regular.

Reductions are a powerful tool to study the complexity of computational problems, and can be used both to prove that certain problems are computationally easy, and others are computationally hard. But be careful: you need to use reductions in the correct direction. For example, if you show that $A \leq_{FST} B$ and also prove (or know) that $A$ is regular, you cannot draw any conclusion about $B$: the problem $B$ may be regular or nonregular. Similarly, if $A \leq_{FST} B$ and $B$ is nonregular, you cannot conclude anything about the regularity of $A$. 