Exercise 8.1.

(a) Give a recursive algorithm for the following operation:

**Input:** States \( p \) and \( q \) of the master automaton.

**Output:** State \( r \) of the master automaton such that \( L(r) = L(p) \cdot L(q) \).

Observe that the languages \( L(p) \) and \( L(q) \) can have different lengths. Try to reduce the problem for \( p, q \) to the problem for \( p^a, q \).

(b) Give a recursive algorithm for the following operation:

**Input:** A state \( q \) of the master automaton.

**Output:** State \( r \) of the master automaton such that \( L(r) = L(q)^R \) where \( R \) is the reverse operator.

(c) A coding over an alphabet \( \Sigma \) is a function \( h: \Sigma \mapsto \Sigma \). A coding \( h \) can naturally be extended to a function over words, i.e., \( h(\varepsilon) = \varepsilon \) and \( h(w) = h(w_1)h(w_2)\cdots h(w_n) \) for every \( w \in \Sigma^n \). Give an algorithm for the following operation:

**Input:** A state \( q \) of the master automaton and a coding \( h \).

**Output:** State \( r \) of the master automaton such that \( L(r) = \{h(w) : w \in L(q)\} \).

Can you make your algorithm more efficient when \( h \) is a permutation?

**Solution.**

(a) Let \( L \) and \( L' \) be fixed-length languages. The following holds:

\[
L \cdot L' = \begin{cases} 
\emptyset & \text{if } L = \emptyset, \\
L' & \text{if } L = \{\varepsilon\}, \\
\bigcup_{a \in \Sigma} \{a\} \cdot L^a \cdot L' & \text{otherwise.}
\end{cases}
\]

These identities give rise to the algorithm on the next page, where we added an extra case for \( q = q_\varepsilon \) and \( q = q_\emptyset \) because the operation \textit{make} is not defined for \( q_\emptyset \):

(b) Let \( L \) be a fixed-length language. The following holds:

\[
L^R = \begin{cases} 
\emptyset & \text{if } L = \emptyset, \\
\{\varepsilon\} & \text{if } L = \{\varepsilon\}, \\
\bigcup_{a \in \Sigma} (L^a)^R \cdot \{a\} & \text{otherwise.}
\end{cases}
\]

These identities give rise to the following algorithm:

[hard] Note that Lines 11 and 12 are introduced in order to represent the language \( \{a_i\} \) in Line 13 as a state \texttt{make}(s_1, s_2, ..., s_n) of the master automaton. This can be avoided by using the algorithm from Exercise 8.1, namely the state that represents \( \{a_i\} \) is \texttt{add-lang}(\{a_i\}). Thus, Lines 11-13 can be replaced just by \( r \leftarrow \text{concat(reverse}(q^a), \text{add-lang}(\{a_i\})) \).
Input: States \( p \) and \( q \) of the master automaton.

Output: State \( r \) of the master automaton such that \( L(r) = L(p) \cdot L(q) \).

\[
\text{concat}(p, q) :
1. \text{if } G(p, q) \text{ is not empty then}
   2. \text{return } G(p, q)
3. \text{else if } p = q_0 \text{ then}
   4. \text{return } q_0
5. \text{else if } p = q_\epsilon \text{ then}
   6. \text{return } q
7. \text{else if } q = q_0 \text{ then}
   8. \text{return } q_0
9. \text{else if } q = q_\epsilon \text{ then}
   10. \text{return } q
11. \text{else}
12. \text{for } a_i \in \Sigma \text{ do}
13. \quad s_i \leftarrow \text{concat}(p^{a_i}, q)
14. \text{G}(p, q) \leftarrow \text{make}(s_1, s_2, ..., s_n)
15. \text{return } G(p, q)
\]

Input: A state \( q \) of the master automaton.

Output: State \( r \) of the master automaton such that \( L(r) = L(q)^R \).

\[
\text{reverse}(q) :
1. \text{if } G(q) \text{ is not empty then}
   2. \text{return } G(q)
3. \text{else if } q = q_0 \text{ then}
   4. \text{return } q_0
5. \text{else if } q = q_\epsilon \text{ then}
   6. \text{return } q_\epsilon
7. \text{else}
8. \quad p \leftarrow q_0
9. \text{for } a_i \in \Sigma \text{ do}
10. \quad s_i \leftarrow q_\epsilon
11. \quad s_j \leftarrow q_0 \text{ for every } i \neq j
12. \quad r \leftarrow \text{concat}(\text{reverse}(q^{a_i}), \text{make}(s_1, s_2, ..., s_n))
13. \quad p \leftarrow \text{union}(p, r)
14. \quad G(q) \leftarrow p
15. \text{return } G(q)
\]

(c) Let \( L \) be a fixed-length language and let \( h \) be a coding. The following holds:

\[
h(L) = \begin{cases}
0 & \text{if } L = \emptyset, \\
\{\varepsilon\} & \text{if } L = \{\varepsilon\}, \\
\bigcup_{a \in \Sigma} h(a) \cdot h(L^a) & \text{otherwise.}
\end{cases}
\]

These identities give rise to the following algorithm:
Input: A state $q$ of the master automaton and a coding $h$.
Output: State $r$ of the master automaton such that $L(r) = \{h(w) : w \in L(q)\}$.

1. \texttt{coding}(q,h) :
2. \hspace{1em} if $G(q)$ is not empty then
3. \hspace{2em} return $G(q)$
4. \hspace{1em} else if $q = q_0$ then
5. \hspace{2em} return $q_0$
6. \hspace{1em} else if $q = q_\varepsilon$ then
7. \hspace{2em} return $q_\varepsilon$
8. \hspace{1em} else
9. \hspace{2em} $p \leftarrow q_0$
10. \hspace{2em} for $a \in \Sigma$ do
11. \hspace{3em} $r \leftarrow \texttt{coding}(q^a,h)$
12. \hspace{3em} $s_{h(a)} \leftarrow r$
13. \hspace{3em} $s_b \leftarrow q_\emptyset$ for every $b \neq h(a)$
14. \hspace{2em} $p \leftarrow \texttt{union}(p, \texttt{make}(s))$
15. \hspace{1em} $G(q) \leftarrow p$
16. \hspace{1em} return $G(q)$

The above algorithm makes use of \texttt{union} because the coding may be the same for distinct letters, i.e. $h(a) = h(b)$ for $a \neq b$ is possible. However, if the coding is a permutation, then this is not possible, and thus each letter maps to a unique residual. Therefore, the algorithm can be adapted as follows:

Input: A state $q$ of the master automaton and a coding $h$ which is a permutation.
Output: State $r$ of the master automaton such that $L(r) = \{h(w) : w \in L(q)\}$.

1. \texttt{coding-permutation}(q,h) :
2. \hspace{1em} if $G(q)$ is not empty then
3. \hspace{2em} return $G(q)$
4. \hspace{1em} else if $q = q_0$ then
5. \hspace{2em} return $q_0$
6. \hspace{1em} else if $q = q_\varepsilon$ then
7. \hspace{2em} return $q_\varepsilon$
8. \hspace{1em} else
9. \hspace{2em} for $a \in \Sigma$ do
10. \hspace{3em} $s_{h(a)} \leftarrow \texttt{coding-permutation}(q^a,h)$
11. \hspace{2em} $G(q) \leftarrow \texttt{make}(s)$
12. \hspace{1em} return $G(q)$

Exercise 8.2.
Let $\Sigma = \{\text{request, answer, working, idle}\}$.

(1) Build a regular expression and an automaton recognizing all words with the property $P_1$: for every occurrence of request there is a later occurrence of answer.
(2) Build an automaton recognizing all words with the property \( P_2 \): there is an occurrence of \( \text{answer} \) before which only \( \text{working} \) and \( \text{request} \) occur.

(3) Using automata theoretic constructions, prove that all words accepted by the automaton \( A \) below satisfy \( P_1 \), and give a regular expression for all words accepted by the automaton \( A \) that violate \( P_2 \).

\[
\begin{array}{c}
\Sigma \\
\rightarrow \\
q_0 \quad \text{answer} \quad q_1
\end{array}
\]

\textit{Solution.} (1) A possible regular expression is \((\Sigma^* \text{answer}^*)(\Sigma \backslash \{\text{request}\})^*)\). (Observe that we must also describe the sequences containing no occurrence of \( \text{request} \).) A minimal DFA for the property is

\[
\begin{array}{c}
\Sigma \backslash \{\text{request}\} \\
\rightarrow \\
s_0 \quad \text{request} \quad s_1 \\
\text{answer}
\end{array}
\]

(3) A minimal NFA for \( P_2 \) is

\[
\begin{array}{c}
\{\text{working, request}\} \\
\rightarrow \\
r_0 \quad \text{answer} \quad r_1 \\
\Sigma
\end{array}
\]

(4) Complementing the automaton for \( P_1 \) we get

\[
\begin{array}{c}
\Sigma \backslash \{\text{request}\} \\
\rightarrow \\
s_0 \quad \text{request} \quad s_1 \\
\text{answer}
\end{array}
\]

The intersection of \( A \) and the automaton for \( P_1 \) is empty: indeed, we can only reach a final state of \( A \) by executing \( \text{request} \), while we can only reach a final state of the automaton for \( P_1 \) by executing \( \text{answer} \). So we cannot simultaneously reach final states in both.

For the second half, since the automaton for \( P_2 \) is deterministic, we can complement it by exchanging final and non-final states (and not forgetting that the trap state now becomes an accepting state). We get:
The intersection with $A$ yields

\[
\{ \text{working, request} \} \Sigma
\]

and the regular expression is $(\text{working} + \text{request})^* \text{idle} \Sigma^* \text{answer}$.

**Exercise 8.3.**

Suppose there are $n$ processes being executed concurrently. Each process has a critical section and a non critical section. At any time, at most one process should be in its critical section. In order to respect this mutual exclusion property, the processes communicate through a channel $c$. Channel $c$ is a queue that can store up to $m$ messages. A process can send a message $x$ to the channel with the instruction $c!x$. A process can also consume the first message of the channel with the instruction $c?x$. If the channel is full when executing $c!x$, then the process blocks and waits until it can send $x$. When a process executes $c?x$, it blocks and waits until the first message of the channel becomes $x$.

Consider the following algorithm. Process $i$ declares its intention of entering its critical section by sending $i$ to the channel, and then enters it when the first message of the channel becomes $i$:

```plaintext
1 process(i):
2   while true do
3       c!i
4       c?i
5       /* critical section */
6       /* non critical section */
```

(a) Sketch an automaton that models a channel of size $m > 0$ where messages are drawn from some finite alphabet $\Sigma$.

(b) Model the above algorithm, with $n = 2$ and $m = 1$, as a network of automata. There should be three automata: one for the channel, one for `process(0)` and one for `process(1)`.

(c) Construct the asynchronous product of the network obtained in (b).

(d) Use the automaton obtained in (c) to show that the above algorithm violates mutual exclusion, i.e. the two processes can be in their critical sections at the same time.
(e) Design an algorithm that makes use of a channel to achieve mutual exclusion for two processes \((n = 2)\). You may choose \(m\) as you wish.

(f) Model your algorithm from (e) as a network of automata.

(g) Construct the asynchronous product of the network obtained in (f).

(h) Use the automaton obtained in (g) to show that your algorithm achieves mutual exclusion.

**Solution.**

(a) We construct an automaton \(A_{\Sigma,m}\) that stores the content of the channel within its states. For example, the automaton for \(\Sigma = \{0, 1\}\) and \(m = 2\) is as follows:

![Diagram of automaton](attachment:image.png)

More formally, \(A_{\Sigma,m} = (Q, \Gamma, \delta, q_0, F)\) is defined as:

\[
Q = \{w \in (\Sigma \cup \square)^m : (w_i = \square) \implies (w_{i+1} = \square) \text{ for every } 1 \leq i < m\},
\]

\[
\Gamma = \{c! \sigma : \sigma \in \Sigma\} \cup \{c? \sigma : \sigma \in \Sigma\},
\]

\[
q_0 = \square^m,
\]

\[
F = Q.
\]

Let \(\ell : Q \to \{1, 2, \ldots, m\}\) be the function that associates to each state \(q\) the position of the last letter of \(q\) which is not \(\square\). For example, \(\ell(abb\square) = 3\). The transitions are formally defined as follows:

\[
\delta(q, c! \sigma) = \begin{cases} q_1q_2\cdots q_{\ell(q)}\sigma\square^{m-\ell(q)-1} & \text{if } \ell(q) < m, \\ \text{none} & \text{otherwise}, \end{cases}
\]

\[
\delta(q, c? \sigma) = \begin{cases} q_2q_3\cdots q_m\square & \text{if } q_1 = \sigma, \\ \text{none} & \text{otherwise}. \end{cases}
\]
Note that \( A_{\Sigma,m} \) grows exponentially since \( |Q| = \sum_{i=0}^{m} |\Sigma|^i = (|\Sigma|^{m+1} - 1)/(|\Sigma| - 1) \).

(b) The automata for the channel, process(0) and process(1) are respectively:

(c)
(d) The algorithm violates mutual exclusion since state \((c_0, c_1, \square)\) is reachable in the above automaton.

(e) We initialize a channel \(c\) of size one with message 1. When a process wants to enter its critical section, it simply consumes 1 from the channel and sends it back once it is done:

```python
1 process():
2     while true do
3         /* non critical section */
4         c ? 1
5         /* critical section */
6         c ! 1
```

(f) The automata modeling the channel and the two processes are respectively:
[hard] Note that we have introduced the new letters $c!1$ and $c?1$. We could have simply used letters $c!1$ and $c?1$. However, these new letters will be important when considering the asynchronous product of the network. If the two automata modeling the processes both used $c!1$ and $c?1$, then the asynchronous product would force them to synchronize on these letters.

[hard] In class, we have seen an alternative solution: to simply swap line 4 and 5 of the processes described in #6.2. This also works. You can verify this solution either manually or with Spin.

(g)
(h) None of the state of the above automaton is of the form $(c_0, c_1, \sigma)$ where $\sigma \in \{\square, 1\}$. This implies that both processes cannot be in their critical sections at the same time.