4.1 Definition of regular expressions. We can build complex languages from simple languages using operations on languages. Consider an alphabet \( \Sigma = \{a_1, \ldots, a_n\} \). The simple languages over \( \Sigma \) are:

- The empty language \( \emptyset \), which contains no word.
- For every symbol \( a \in \Sigma \), the language \( \{a\} \), which contains only the one-letter word “a”.

The boolean operations on languages are \( \cap \) (intersection), \( \cup \) (union), and \( \overline{\cdot} \) (complementation with respect to \( \Sigma^* \)). The regular operations on languages are \( \cup \) (union), \( \circ \) (concatenation), and \( \ast \) (iteration):

\[
L_1 \circ L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}.
\]

\[
L^* = \bigcup_{i \geq 0} L^i = \{\varepsilon\} \cup L \cup (L \circ L) \cup (L \circ L \circ L) \cup \ldots
\]

An expression that applies regular operations to simple languages is called a regular expression, and the resulting language is a regular language. To distinguish between expressions and languages, we write \( L(E) \) for the regular language defined by the regular expression \( E \). For instance, the expression \( E = (\{0\} \circ \{1\}) \circ \emptyset \) defines the language \( L(E) = \{00, 10\} \).

4.2 Notation. When writing regular expressions, we use the following conventions:

- For simple languages of the form \( \{a\} \), we write \( a \) (omitting braces).
- Parentheses are omitted according to the rule that iteration binds stronger than concatenation, which binds stronger than union.
- The concatenation symbol \( \circ \) is often omitted.
- We write \( \Sigma \) for any language.
- We write \( \varepsilon \) for \( \emptyset^* \) (which is the language that contains only the empty word).

For example, \( 01^* \cup \varepsilon \) stands for the expression \( (\{0\} \circ (\{1\}^*)) \cup (\emptyset^*) \).

4.3 Examples.

\[
\Sigma^*000\Sigma^* \ldots \text{ the language of all words that contain the substring 000} \\
(\Sigma\Sigma)^* \ldots \text{ the language of all words with an even number of letters} \\
(0^*10^*1)^*0^* \ldots \text{ the language of all words that contain an even number of 1’s}
\]

4.4 From regular expressions to finite automata. For every simple language \( S \), there is a finite automaton \( M \) such that \( L(M) = S \). In particular, the language \( \emptyset \) is defined by the automaton on the left (without accept state), and the language \( \{a\} \) is defined by the automaton on the right:

Furthermore, the three regular operations can be performed on finite automata:

**Union** Recall Section 3.5: given two NFAs \( M_1 \) and \( M_2 \), we can construct \( M_1^\cup \) such that \( L(M_1^\cup) = L(M_1) \cup L(M_2) \). If \( M_1 \) has \( n_1 \) and \( M_2 \) has \( n_2 \) states, then \( M_1^\cup \) has \( n_1 + n_2 + 1 \) states.

**Concatenation** Given two NFAs \( M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \) and \( M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \), define:

\[
M_1^\circ = (Q_1 \cup Q_2, \Sigma, \delta_1 \cup \delta_2 \cup \{(p, \varepsilon, q_{02}) \mid p \in F_1\}, q_{01}, F_2).
\]
Then $L(M_\epsilon^n) = L(M_1) \circ L(M_2)$. If $M_1$ has $n_1$ and $M_2$ has $n_2$ states, then $M_\epsilon^n$ has $n_1 + n_2$ states.

**Iteration** Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$, define:

$$M_\epsilon^* = (Q \cup \{q_{\text{new}}\}, \Sigma, \delta \cup \{(q_{\text{new}}, \epsilon, q_0)\} \cup \{(p, \epsilon, q_{\text{new}}) \mid p \in F\}, q_{\text{new}}, \{q_{\text{new}}\}).$$

Then $L(M_\epsilon^*) = L(M)^*$. If $M$ has $n$ states, then $M_\epsilon^*$ has $n + 1$ states.

In this way, every regular expression $E$ can be converted into an NFA $M$ (with $\epsilon$-transitions) that defines the same language; that is, $L(M) = L(E)$. The size of $M$ is proportional to the number of regular operations in $E$, and the conversion requires linear time. Removing $\epsilon$-transitions (in linear time) and determinizing (in exponential time) can be performed as usual:

**Theorem 4A.** For every regular language $R$, there is a DFA such that $L(M) = R$.

### 4.5 From finite automata to regular expressions.

Not only is every regular language definable by a finite automaton, also every finite automaton defines a regular language; that is, the regular expressions and the finite automata (deterministic or nondeterministic) represent exactly the same class of languages:

**Theorem 4B.** For every NFA $M$, the language $L(M)$ is regular.

This theorem is proved by converting a given NFA $M$ into a regular expression $E$ such that $L(E) = L(M)$. The algorithm\(^1\) removes one state of $M$ at a time. It follows an approach called *dynamic programming*. Suppose that state $q_k$ is removed. Then for every pair of states $q_i$, $q_j$ (with $k \notin \{i, j\}$) we perform the following update of the transition labels:

\(^1\)detailed together with an example in Section 1.3 of M. Sipser’s book
If $E_{ij}$ is a regular expression that defines the inputs on which the automaton can move directly from $q_i$ to $q_j$ before the removal of $q_k$, then $E_{ij} \cup E_{ik} E_{kk}^* E_{kj}$ defines the inputs on which the automaton can move directly from $q_i$ to $q_j$ after the removal of $q_k$. Note that the update involves all three regular operations ($\cup$, $\circ$, and $^*$). With each removal of a state, the regular expressions can grow by a factor of 4. Consequently, if $M$ has $n$ states, the regular expression for $L(M)$ may have size $O(4^n)$.

### 4.6 Boolean operations on regular expressions

The regular expressions contain only one of the three boolean operations, namely, union. The equivalence between regular expressions and finite automata shows that the other two operations, intersection and complement, are unnecessary: given two regular expressions $E_1$ and $E_2$, we can obtain a regular expression for $L(E_1) \cap L(E_2)$ by

1. converting $E_1$ and $E_2$ into equivalent finite automata $M_1$ and $M_2$ (Section 4.3),
2. intersecting $M_1$ and $M_2$, so obtaining the finite automaton $M_{\cap}$ (Section 2.3),
3. converting $M_{\cap}$ to a regular expression $E_{\cap}$ (Section 4.4).

The conversion (3) has exponential cost. A similar procedure can be applied for complementing a regular expression: convert to an automaton, complement, and convert back. So, while intersection and complement do not add expressiveness to regular expressions, they add an exponential improvement in succinctness.