Heuristically, regular languages correspond to problems that can be solved with finite memory, **i.e.** we only need to remember one of finitely many things. By contrast, nonregular languages correspond to problems that cannot be solved with finite memory.

Theorem: The collection of regular languages L is also closed under the following two operations:

1. Intersection, **i.e.** if L', L" are regular languages (**i.e.** $L' \in C$ and $L'' \in C$), then their intersection $L' \cap L$" is a regular language.
2. Complement, **i.e.** if L is a regular language (**i.e.** $L \in C$), then $A^* \backslash L$ is a regular language ($A^* \backslash L \in C$).

Remark: These two properties did not come into the definition of a regular language, but they are true and often quite useful.

8.3 Finite State Acceptors and Automata Theory

Definition: An automation is a mathematical model of a computing device. Plural of automation is automata.

Basic idea: Reason about computability without having to worry about the complexity of actual implementation.
It is most reasonable to consider at the beginning just finite states automata, **i.e.** machines with a finite number of internal states. The data is entered discretely, and each datum causes the machine to either remain in the same internal state or else make the transition to some other state determined solely by 2 pieces of information:

1. The current state
2. The input datum

In other words, if S is the finite set of all possible states of our finite state machine, then the transition mapping t that tells us how the internal state of the machine changes on inputting a datum will depend on the current state $s \in S$ and the imput datum a, **i.e.** the machine will enter a (potentially) new state $s' = t(s, a)$.

Want to use finite state machines to recognise languages over some alphabet A. Let L be our language.
Since our finite state machine accepts (**i.e.** returns yes to) w if $w \in L$,

$$\begin{array}{cc} \underline{\text{Input}} & \underline{\text{Output}} \\ \text{Word } w = a_1 ... a_n, a_i \in A \, \forall i & \begin{array}{c} \text{Yes if } w \in L \\ \text{No if } w \notin L \end{array} \end{array}$$

we call our machine a finite state acceptor. We want to give a rigorous definition of a finite state acceptor. To check $w = a_1 ... a_n$, we input each a_i starting with a_1 and trace how the internal state of the machine changes. S is our set of states of the machine (a finite set). The transition mapping t takes the pair(s, a) and returns the new state $s' = t(s, a)$ (where $s \in S$ and $a \in A$) that the machine has reached so $t : S \times A \to S$.
Some elements and subsets of S are important to understand:

1. The initial state $i \in S$ where the machine starts

2. The subset $F \subseteq S$ of finishing states

It turns out that knowing S, F, i, t, A specifies a finite state acceptor completely.

Definition: A finite state acceptor (S, A, i, t, F) consists of a finite set S of states, a finite set A that is the input alphabet, a starting state $i \in S$, a transition mapping $t : S \times A \to S$, and a set F of finishing states, where $F \subseteq S$.

Definition: Let (S, A, i, t, F) be a finite state acceptor, and let A^* denote the set of words over the input alphabet A. A word $a_1, a_2...a_n$ of length n over the alphabet A is said to be recognised or accepted by the finite state acceptor if $\exists s_0, s_1, ..., s_n \in S$ states s.t. $s_0 = i$ (the initial state), $s_n \in F$, and $s_i = t(s_{i-1}, a_i)\ \forall i \quad 1 \leq i \leq n$.

Definition: Let (S, A, i, t, F) be a finite state acceptor. A language L over the alphabet A is said to be recognised or accepted by the finite state acceptor if L is the set consisting of all words recognized by the finite state acceptor.

In the definition of a finite state acceptor, t is the transition mapping, which may or may not be a function (hence the careful terminology). This is because finite state acceptors come in 2 flavours:

1. Deterministic: every state has exactly one transition for each possible input, **i.e.** $\forall (s, a) \in S \times A\ \exists!\ t(s, a) \in S$. In other words, the transition mapping is a function.
2. Non-deterministic: an input can lead to one, more than one or no transition for a given state. Some $(s, a) \in S \times A$ might be assigned to more than one element of S, **i.e.** the transition mapping is not a function.