

## Zero-th lesson: Computer programs

### 1. The Halting Problem

Turing's paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it.  
— Minsky (1967), p. 104 [from Wikipedia]

>>A **Turing machine** is a [mathematical model of computation](#) that defines an [abstract machine](#)<sup>[1]</sup> which manipulates symbols on a strip of tape according to a table of rules.<sup>[2]</sup>

>>The machine operates on an infinite<sup>[4]</sup> memory tape divided into [discrete cells](#).<sup>[5]</sup> The machine positions its *head* over a cell and "reads" (scans<sup>[6]</sup>) the symbol there. Then, [given the symbol, and the present place of the machine] in a *finite table*<sup>[7]</sup> of user-specified instructions, the machine (i) writes a symbol (e.g. a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure<sup>[8]</sup> or no writing), then (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head),<sup>[9]</sup> then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts<sup>[10]</sup> the computation.

The Turing machine was invented in 1936 by [Alan Turing](#)<sup>[11][12]</sup> (c.f. fig. 1). With this model, Turing was able to provide a (negative) answer to the question: (1) Does a machine exist that can determine whether any arbitrary machine on its tape is "circular" (e.g. freezes, or fails to continue its computational task).<sup>[14]</sup> Thus by providing a mathematical description of a very simple device capable of arbitrary computations, he was able to prove properties of computation in general—and in particular, the [uncomputability](#) of the [Entscheidungsproblem](#) ("decision problem").<sup>[15]</sup>

>> Thus, Turing machines prove fundamental limitations on the power of mechanical computation (Sipser 2006:137 observes that "A Turing machine can do everything that a real computer can do. Nevertheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.").

>> While [Turing machines] can express arbitrary computations, their minimalistic design makes them unsuitable for computation in practice: real-world [computers](#) are based on different designs that, unlike Turing machines, use [random-access memory](#).

>> In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates. This result was obtained in 1966 by F. C. Hennie and R. E. Stearns. (Arora and Barak, 2009, theorem 1.9)

### Relationship with Goedel's incompleteness theorems

>> The concepts raised by [Gödel's incompleteness theorems](#) are very similar to those raised

by the halting problem, and the proofs are quite similar. In fact, a weaker form of the First Incompleteness Theorem is an easy consequence of the undecidability of the halting problem. This weaker form differs from the standard statement of the incompleteness theorem by asserting that a complete, [consistent](#) and [sound axiomatization](#) of all statements about natural numbers is unachievable.

>> The first incompleteness theorem states that no [consistent system](#) of axioms whose theorems can be listed by an [effective procedure](#) (i.e., an [algorithm](#)) is capable of proving all truths about the arithmetic of the [natural numbers](#). For any such formal system, there will always be statements about the natural numbers that are true, but that are unprovable within the system. The second incompleteness theorem, an extension of the first, shows that the system cannot demonstrate its own consistency.

>>For example, one [...] consequence of the halting problem's undecidability is that there cannot be a general [algorithm](#) that decides whether a given statement about [natural numbers](#) is true or not. The reason for this is that the [proposition](#) stating that a certain program will halt given a certain input can be converted into an equivalent statement about natural numbers. If we had an algorithm that could find the truth value of every statement about natural numbers, it could certainly find the truth value of this one; but that would determine whether the original program halts, which is impossible, since the halting problem is undecidable.

## 2. Computer programs

>> A **computer program** is a collection of [instructions](#)<sup>[1]</sup> that performs a specific task when [executed](#) by a [computer](#). A computer requires programs to function and typically executes the program's instructions in a [central processing unit](#).<sup>[2]</sup>

>> A computer program in the form of a [human-readable](#), computer programming language is called [source code](#). Source code may be (1) converted into an [executable image](#) by a [compiler](#) or (2) [executed](#) immediately with the aid of an [interpreter](#). Compilers are used to translate source code from a programming language into [...] [machine code](#).<sup>[19]</sup> [...] Machine code consists of the [central processing unit's](#) native instructions, ready for execution. Compiled computer programs are commonly referred to as executables, binary images, or simply as [binaries](#) – a reference to the [binary file format](#) used to store the executable code.