

Theoretical Computer Science

Lecture Notes

Prof. Dr. Titus Dose

OTH Regensburg – Faculty of Computer Science and Mathematics

October 30, 2024

Table of Contents I

1. Mathematical Basics

- 1.1 Foundations from Propositional Logic
- 1.2 Methods of Proof
- 1.3 Basic Notations and Terms

Table of Contents II

2. Computability

- 2.1 History of the Concept of Algorithms
- 2.2 Alphabets, Words, Formal Languages
- 2.3 While Programs
- 2.4 Register Machines
- 2.5 Mini-While Programs
- 2.6 Turing Machines
- 2.7 Fundamental Theorem of the Theory of Algorithms
- 2.8 Runtime of Algorithms
- 2.9 Further Computation Models
- 2.10 Decidability

Table of Contents III

1

Mathematical Basics

1.1

Foundations from Propositional Logic

This lecture is about proving mathematical statements.

A mathematical statement is either true (1) or false (0). So it always has a unique truth value.

Some examples:

- ▶ 5 is a prime
- ▶ there are infinitely many twin primes (a twin prime is a pair $(p, p + 2)$ such that both numbers are prime)
- ▶ there are no even primes

Can the following sentence be considered as a statement?

“This is a sentence, which is wrong”

We will be using the following logical connections that turn statements into more complicated statements:

- ▶ “and”, conjunction, \wedge
- ▶ “or”, disjunction, \vee
- ▶ “not”, negation, \neg
- ▶ implication, \Rightarrow , “if ..., then ...”
- ▶ equivalence, \Leftrightarrow , “if and only if ..., then ...”

In general, the natural language description should be preferred to the formula notation using symbols: e.g., “and” should be preferred to “ \wedge ”.

The following truth value table *defines* the connections:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

Some typical mistakes and misunderstandings:

- ▶ the negation of the statement “Each prime is odd” is **not** “Each prime is even”, but “There is an even prime”.
- ▶ the disjunction is inclusive (contrary to “either-or”), i.e., the following statement is true: “2 is even or 2 is a prime”.
- ▶ Everything is implied by a false statement. E.g., the following statement is true: “If 9 is a prime, then there are infinitely many twin primes” (even if we do not know whether there are infinitely many twin primes).¹

¹This is in accordance with everyday language: the statement “If it rains tomorrow, then the street will get wet” is considered to be true even if it does not rain the next day.

► Note: $A \Leftrightarrow B$ is equivalent with $(A \Rightarrow B) \wedge (B \Rightarrow A)$

A	B	$A \Rightarrow B$	$B \Rightarrow A$	$(A \Rightarrow B) \wedge (B \Rightarrow A)$	$A \Leftrightarrow B$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	1	0	0	0
0	0	1	1	1	1

A typical mistake is to forget one of the two implications, especially when having a longer chain of equivalent statements. As an example, consider the following statements about a real number x

$$x = 2 \Leftrightarrow x + 1 = 3$$

$$\Leftrightarrow (x + 1)^2 = 9 \text{ WRONG!}$$

$$\Leftrightarrow 2^{(x+1)^2} = 512$$

However, the statements " $x = 2$ " and " $2^{(x+1)^2} = 512$ " are not equivalent as $-4 \neq 2$, but $2^{(-4+1)^2} = 512$.

Tip: In the first semester, mainly do without \Leftrightarrow and instead prove $A \Rightarrow B$ and $B \Rightarrow A$ separately.

1.2

Methods of Proof

We will be using 3.5 methods of proof.

1. Direct proof
2. Indirect proof
 - 2.1 Proof by contradiction
 - 2.2 Proof by contraposition
3. Induction

Direct Proof

Whenever we prove a mathematical statement, we start with a statement A , assume it to be true, and show that then another statement B is also true (i.e., we always prove implications).

The condition A is not always stated explicitly. We can e.g. prove the statement “2 is a prime”. But where’s the premise?

When proving the statement, we will have to use some premises, e.g., certain properties of the multiplication of natural numbers, properties of natural numbers, . . .

We cannot squeeze water from a stone. We always need axioms.

Direct proof: we assume that A is true and show by a chain of logical inferences² that B is true.

²It can be proven that a couple of simple inference rules suffice. Thus, it is easy to write a program that can check arbitrary mathematical proofs for their correctness (if they are written down detailed enough).

Direct Proof – Example

Let p be a natural number. Let A be the statement “ p is an even prime” and B the statement “ p is less than 5”. We now prove $A \Rightarrow B$.

Proof.

We assume that A is true. So p is an even prime.

A implies statement C : “ $p = 2$ ” because every other even number is divisible by 2 and thus no prime.

C implies B as 2 is less than 5. □

Contraposition

The statement $A \Rightarrow B$ is equivalent to $\neg B \Rightarrow \neg A$.³

Instead of proving $A \Rightarrow B$ directly, we give a direct proof for the statement $\neg B \Rightarrow \neg A$.

Example: Let p be a natural number. Let A be the statement “ p^2 is not divisible by 4” and B the statement “ p is odd”. We prove $A \Rightarrow B$.

Proof.

We give a proof by contraposition.

So we start with the statement $\neg B$: “ p is even”.

Then $p = 2 \cdot q$ for some natural number q .

Then $p^2 = (2 \cdot q)^2 = 4 \cdot q^2$, so p^2 is divisible by 4, which is the statement $\neg A$. □

³Write down the truth value table if you are not convinced. Cf. page 10.

Proof by Contradiction

The proof by contradiction is similar to the proof by contraposition.

We prove $A \Rightarrow B$ by starting with the statement $A \wedge \neg B$ and inferring a contradiction. Then our assumption $A \wedge \neg B$ must have been wrong (otherwise, it wouldn't imply a contradiction).

Thus, it is impossible that A is true and B is not. In other words, if A is true, then B is also true, i.e., $A \Rightarrow B$.

Proof by Contradiction – Example

Example: Let us prove that every natural number greater than 1 has a divisor that is prime (prime divisor).

Proof.

Let's assume for a contradiction that there exists some number that does not have a prime divisor. Let p be the least such number.

Then p is not prime (p is divisor of p). Thus, $p = r \cdot s$ for natural numbers $r, s > 1$.

As p is the least number > 1 without any prime divisors, $r > 1$, and $r < p$, the number r has a prime divisor t , i.e., $r = t \cdot u$ for a natural number u .

But then $p = r \cdot s = t \cdot u \cdot s$ and since t is prime, p has a prime divisor. This is a contradiction. So our assumption was wrong and every natural number > 1 has a prime divisor. \square

Proof by Contradiction – 2nd Example

Now we can prove with the same technique that there are infinitely many primes.

Proof.

Let's assume for a contradiction that there exist only finitely many primes, i.e., there are some $n \in \mathbb{N}$ and primes p_1, \dots, p_n such that these are the only primes.

Consider $p = p_1 \cdot p_2 \cdot \dots \cdot p_n + 1$.

As p is greater than 1, it has a prime divisor (by the statement proven on the last page). So there is some i so that p_i is a divisor of p , i.e., $p = p_i \cdot s$ for a natural number s .

Thus $s = \frac{p}{p_i} = \frac{p_1 \cdot p_2 \cdot \dots \cdot p_n + 1}{p_i} = \frac{p_1 \cdot p_2 \cdot \dots \cdot p_n}{p_i} + \frac{1}{p_i}$, i.e., $\frac{1}{p_i} = s - \frac{p_1 \cdot p_2 \cdot \dots \cdot p_n}{p_i}$.

So $\frac{1}{p_i}$ is the difference of two natural numbers and hence, an integer itself. However, as p_i is prime, $0 < \frac{1}{p_i} \leq \frac{1}{2}$, a contradiction. \square

Complete Induction

Complete Induction is a method of proof we will be using extensively.

Consider the statement: For all natural numbers n the following statement A_n is true:

$$\sum_{i=1}^n i = \frac{n^2 + n}{2} \quad .^4$$

Proving all statements A_n for $n \in \mathbb{N}$ directly is not straightforward. The idea of complete induction is to prove the statement A_0 (base case, BC) and “for all $k \in \mathbb{N}$, if A_k is true, then so is A_{k+1} ” (induction step, IS). This proves that all A_n are true:

$$A_0 \Rightarrow A_1 \Rightarrow A_2 \Rightarrow A_3 \Rightarrow A_4 \Rightarrow \dots$$

Note: BC and IS can be proven using any of the proof methods.

$$^4 \sum_{i=1}^n i = 1 + 2 + \dots + (n-1) + n$$

Complete Induction – Example

For all natural numbers n the following statement A_n is true:

$$\sum_{i=1}^n i = \frac{n^2 + n}{2}$$

Proof.

BC: We prove A_0 : It holds $\sum_{i=1}^0 i = 0 = \frac{0^2+0}{2}$.

IS: We prove “for all $k \in \mathbb{N}$, if A_k is true, then so is A_{k+1} ”. Let $k \in \mathbb{N}$. We assume that A_k is true (**induction hypothesis**). So $\sum_{i=1}^k i = \frac{k^2+k}{2}$.

In order to complete the proof, we need to prove A_{k+1}

Recall: We have assumed the statement A_k :

$$\sum_{i=1}^k i = \frac{k^2 + k}{2}.$$

We want to prove the statement A_{k+1} (in order to complete IS):

$$\sum_{i=1}^{k+1} i = \frac{(k+1)^2 + k+1}{2}.$$

Then

$$\begin{aligned}\sum_{i=1}^{k+1} i &= k+1 + \underbrace{\sum_{i=1}^k i}_{\stackrel{A_k}{=} \frac{k^2+k}{2}} = k+1 + \frac{k^2+k}{2} = \frac{2(k+1) + k^2+k}{2} \\ &= \frac{k^2 + 2k + 1 + k + 1}{2} = \frac{(k+1)^2 + k+1}{2}\end{aligned}$$

holds⁵. Thus A_{k+1} holds, IS is proven, and the proof is complete. □

⁵Note: the last equation is due to Francesco Binomi (1727 – 1643).

Inductive Definitions

Having introduced the concept of proofs via complete induction, we can now use the same concept for definitions (inductive definition).

We define the summation operation \sum . Let s_1, s_2, \dots be natural numbers. We define:

$$\text{BC: } \sum_{i=1}^0 s_i := 0.$$

$$\text{IS: } \sum_{i=1}^{k+1} s_i := s_{k+1} + \sum_{i=1}^k s_i.$$

Why does this define e.g. the expression $\sum_{i=1}^5 i$?

$$\begin{aligned} \sum_{i=1}^5 i &\stackrel{\text{IS}}{=} 5 + \sum_{i=1}^4 i \stackrel{\text{IS}}{=} 5 + 4 + \sum_{i=1}^3 i \stackrel{\text{IS}}{=} 5 + 4 + 3 + \sum_{i=1}^2 i \\ &\stackrel{\text{IS}}{=} 5 + 4 + 3 + 2 + \sum_{i=1}^1 i \stackrel{\text{IS}}{=} 5 + 4 + 3 + 2 + 1 + \sum_{i=1}^0 i \\ &\stackrel{\text{BC}}{=} 5 + 4 + 3 + 2 + 1 + 0. \end{aligned}$$

Structural Induction

Inductive proofs are not only useful when considering statements that hold for all natural numbers.

Generally, they can be used for recursively/inductively defined structures. E.g., for **polynomials** over the integers with one variable:

BC Let x be a variable. Then

- ▶ x is a polynomial
- ▶ each integer z is a polynomial

IS If p_1 and p_2 are polynomials over the integers with one variable x , then $(p_1 + p_2)$ and $(p_1 \cdot p_2)$ are polynomials.

Example:⁶ $((5 \cdot (((x \cdot x) \cdot x) - (3 \cdot (x \cdot x))) + 12)) \cdot (((3 \cdot x) - 5)))$
5, x , 3, 12 are polynomials by BC. Applying IS iteratively finally leads to the polynomial above.

⁶Note: later on we will omit unnecessary brackets and write $5 \cdot (x^3 - 3x^2 + 12) \cdot (3x - 5)$.

Structural Induction – Example

Let's prove: All polynomials have the same number of opening and closing brackets.

Proof.

BC: Neither x nor an integer contains any bracket.

IS: Let p_1 be polynomial with a opening and a closing brackets. Let p_2 be polynomial with b opening and b closing brackets.

The polynomial $(p_1 + p_2)$ has $a + b + 1$ opening and closing brackets. The same holds for $(p_1 \cdot p_2)$. This completes the proof.

In the following we use the term “induction” for both “complete induction” and “structural induction”. □

1.3

Basic Notations and Terms

We use $()$ and $[]$ to bracket mathematical expressions. Arguments of functions are placed in $()$. Sets are bracketed with $\{\}$.

$\mathbb{N} \stackrel{df}{=} \{0, 1, \dots\} = \text{Set of natural numbers}$

$\mathbb{N}^+ \stackrel{df}{=} \{1, 2, \dots\} = \text{Set of positive natural numbers}$

$\mathbb{P} \stackrel{df}{=} \{2, 3, 5, 7, 11, 13, 17, \dots\} = \text{Set of primes}$

$\mathbb{Z} \stackrel{df}{=} \{\dots, -2, -1, 0, 1, 2, \dots\} = \text{Set of integers}$

$\emptyset \stackrel{df}{=} \text{empty set}$

Important: **0 is a natural number.**

For $x \in \mathbb{N}$, $y \in \mathbb{N}^+$ let $(x \bmod y)$ denote the remainder of the division x/y , i.e., $(x \bmod y) = x - zy$, where $z \in \mathbb{N}$ is the greatest number with $zy \leq x$.

Sets

Notation for the definition of sets:

$$\begin{aligned}\{n : n \in \mathbb{N} \text{ and } n \geq 5\} &= \{n \in \mathbb{N} : n \geq 5\} \\ &= \{n \mid n \in \mathbb{N} \text{ and } n \geq 5\} \\ &= \{n \in \mathbb{N} \mid n \geq 5\} \\ &= \{5, 6, \dots\}\end{aligned}$$

Element relationship and inclusion:

- ▶ $a \in M \xLeftrightarrow{df} a$ is an element of the set M
- ▶ $a \notin M \xLeftrightarrow{df} a$ is no element of the set M
- ▶ $M \subseteq N \xLeftrightarrow{df}$ for all a , if $a \in M$, then $a \in N$ (M is subset of N)
- ▶ $M \not\subseteq N \xLeftrightarrow{df}$ it does not hold $M \subseteq N$ (M is no subset of N)
- ▶ $M \subsetneq N \xLeftrightarrow{df} M \subseteq N$ and $M \neq N$ (M is proper subset of N)

Set Operations

$A \cap B \stackrel{df}{=} \{a \mid a \in A \text{ and } a \in B\}$ (Intersection of A and B)

$A \cup B \stackrel{df}{=} \{a \mid a \in A \text{ or } a \in B\}$ (Union of A and B)

$A \setminus B \stackrel{df}{=} \{a \mid a \in A \text{ and } a \notin B\}$ (Difference of A and B)

$A - B \stackrel{df}{=} A \setminus B$ (Difference of A and B)

$\overline{A} \stackrel{df}{=} M \setminus A$ (Complement of A relative to a fixed base set M)

$\mathcal{P}(A) \stackrel{df}{=} \{B \mid B \subseteq A\}$ (Power set of A)

$\#A = |A| \stackrel{df}{=} \text{number of elements of a finite set } A$

Tuple (Vector) and Cartesian Product

For $n \in \mathbb{N}$ we define the following.

- ▶ $(a_1, a_2, \dots, a_n) \stackrel{df}{=} \text{sequence of elements } a_1, a_2, \dots, a_n \text{ in this order } (n\text{-tuple, } n\text{-dimensional vector})$
- ▶ $A_1 \times A_2 \times \dots \times A_n \stackrel{df}{=} \{(a_1, a_2, \dots, a_n) \mid a_i \in A_i \text{ for all } i\}$
(Cartesian product of sets A_1, A_2, \dots, A_n)
- ▶ $A^n \stackrel{df}{=} \underbrace{A \times A \times \dots \times A}_{n \text{ times}} \text{ (} n\text{-dim. Cartesian product of set } A \text{)}$

The first definition yields the empty tuple $()$ for $n = 0$. Thus $A^0 = \{()\}$ and $|A^0| = 1$.

Quantifiers

Quantifiers are merely abbreviations:

- ▶ \exists = “there exist(s)”
- ▶ \forall = “for all”

Instead of “Every even natural number greater or equal 4 is the sum of two (non-necessarily distinct) primes” we may write as shortcut $\forall_{n \in \{2 \cdot i \mid i \in \mathbb{N}, i \geq 2\}} \exists_{p, q \in \mathbb{P}} n = p + q$.

For better readability, we will go without subscripts when using quantifiers, i.e., $\forall n \in \{2 \cdot i \mid i \in \mathbb{N}, i \geq 2\} \exists p, q \in \mathbb{P} n = p + q$.

Negating such statements can be done mechanically, replace \forall with \exists and vice versa, and negate the remaining statement: $\exists n \in \{2 \cdot i \mid i \in \mathbb{N}, i \geq 2\} \forall p, q \in \mathbb{P} n \neq p + q$.

Functions

A function f from A to B (notation $f : A \rightarrow B$) is determined by the source set (aka domain) A , the target set (aka codomain) B , and the graph $G_f \subseteq A \times B$, where for every $a \in A$ there is at most one $b \in B$ with $(a, b) \in G_f$.

If $(a, b) \in G_f$, then the function f at the position a has the function value b (notation: $f(a) = b$).

If there is no $b \in B$ with $f(a) = b$, then $f(a)$ is not defined (notation: $f(a) = \text{n.d.}$).

Example: The functions g and h are different, although $G_g = G_h$.

- ▶ $g : \mathbb{N} \rightarrow \mathbb{N}$ with $g(x) = 0$ for all $x \in \mathbb{N}$
- ▶ $h : \mathbb{Z} \rightarrow \mathbb{Z}$ with $h(x) = 0$ if $x \geq 0$ and $h(x) = \text{n.d.}$ otherwise

Definition 1.1

Let $f: A \rightarrow B$ and $g: B \rightarrow C$ be functions.

- ▶ $g \circ f$ denotes the function $A \rightarrow C$ with $(g \circ f)(x) \stackrel{df}{=} g(f(x))$ (composition of functions).
- ▶ **Domain of definition** of f : $D_f \stackrel{df}{=} \{a \in A \mid \exists b \in B f(a) = b\}$
- ▶ **Range** or **image** of f : $R_f \stackrel{df}{=} \{b \in B \mid \text{there is } a \in A \text{ with } f(a) = b\}$
- ▶ f is **total** $\stackrel{df}{\iff} D_f = A$
- ▶ f is **surjective** $\stackrel{df}{\iff} R_f = B$
- ▶ f is **injective** $\stackrel{df}{\iff} f(a_1) \neq f(a_2)$ for all distinct $a_1, a_2 \in D_f$
- ▶ f is **bijective** $\stackrel{df}{\iff} f$ is total, surjective, and injective
- ▶ If f injective, there exists the **inverse function** $f^{-1}: B \rightarrow A$ with $f^{-1}(b) \stackrel{df}{=} \text{the } a \in A \text{ with } f(a) = b$.
Note: for an injective function f it holds $f^{-1}(f(a)) = a$ for $a \in D_f$ and $f(f^{-1}(b)) = b$ for $b \in R_f$.

2

Computability

Computability – Outline

- ▶ Everyone has a certain idea of what computers can do. We want to take a closer look at this question and find out which tasks can in principle be solved with computers. This requires meaningful and precise definitions of the terms *algorithm* and *computability*. The creation of these definitions is considered one of the most important contributions to mathematics in the 20th century and was the result of a development that can be traced back to discussions of philosophical questions in antiquity.
- ▶ In this chapter, we will first get to know various computation models: While programs (WHILE), Random access machines (RAM), Turing machines (TM).

Computability – Outline II

- ▶ We show the equivalence of these computational models and thus substantiate the thesis that they capture exactly what computers can do.
- ▶ We have now defined the terms *algorithm* and *computability* in a meaningful and precise way.
- ▶ Advantage: We can now also recognize the limits of computability. We show that many easily formulated tasks cannot be solved by computers.
- ▶ For example, the question of whether two given computer programs compute the same function cannot be solved by computers. This shows that the automatic verification of arbitrary computer programs cannot work.

2.1

History of the Concept of Algorithms

Etymology of the Term “Algorithm”

The word *algorithm* is derived from the name of the mathematician *Muhammad Al-Chwarizmi*, who lived in Baghdad around 780–850. He wrote a book, that was important at the time, on methods for dealing with algebraic equations.

In the Middle Ages, the word *algorismus* was used to describe the art of calculating with Arabic numerals.

Later, the term *algorithm* was generally used to describe a procedure for solving a problem.



Al-Chwarizmi

Historical Development

The first half of the 20th century saw a rapid development in mathematical logic, which led to the precise definition of the terms *algorithm* and *computability* in the 1930s. The reason for this was not the emergence of computing technology, but the question of the axiomatizability of mathematics:

- ▶ Even in ancient times, there were discussions about whether there are fundamental limitations to knowledge and predictability. The liar's paradox has been known since the 4th century BC: "This statement is false". The basic ideas behind today's concept of axiomatization can already be found in Plato and Aristotle (400/350 BC).



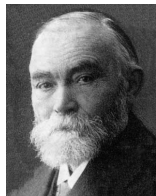
- ▶ Gottfried Leibniz (1646–1716) has a vision of a machine that can manipulate symbols and thus determine the truth value of mathematical statements.



- ▶ 1889: Giuseppe Peano (1858–1932) presents the first formal axioms for natural numbers.



- ▶ 1893: Gottlob Frege (1848–1925) writes his “Grundgesetze der Arithmetik” (Foundations of arithmetic), which are intended to give all of mathematics a purely logical foundation (logicism). All theorems of arithmetic should be traceable to logical truths.



- ▶ 1900: David Hilbert (1862–1943) in his speech at the Mathematics Congress in Paris: Paradoxes (like the liar’s paradox) in mathematics are based on the ambiguity of natural language. They can be avoided by axiomatization.



- ▶ 1902: Bertrand Russell (1872–1970) points out the following paradox to Frege in a letter (*Russel's antinomy*). For $R \stackrel{df}{=} \{x \mid x \notin x\}$ it holds

$$R \in R \Leftrightarrow R \notin R.$$



- ▶ Such contradictions lead to the so-called *foundational crisis of mathematics*.
- ▶ 1920: Hilbert takes up his idea from 1900 again and develops the so-called Hilbert's program: Call for the complete formalization of mathematics. Theorems should be derived mechanically by applying rules from axioms.

- ▶ 1928: Hilbert formulates the decision problem: Is there an algorithm that, given a mathematical statement, determines whether it follows from a given set of axioms?
- ▶ 1931: Kurt Gödel (1906–1978) shows the hopelessness of Hilbert's program. *Gödel's incompleteness theorem*: Non-trivial axiomatic systems (e.g. Peano arithmetic) always leave the truth value of certain sentences open.
“Either mathematics is too big for the human mind or the human mind is more than a machine.”



- ▶ 1936: Alonzo Church (1903-1995) and Alan Turing (1912-1954) independently show that Hilbert's decision problem cannot be solved by any algorithm. This requires the definition of the terms *algorithm* and *computability*.

Church uses the lambda calculus, Turing uses the Turing Machine (TM) named after him. It can be seen later that both models have the same computational power.

Gödel's incompleteness theorem and, above all, the coding of formulas and programs developed by him with natural numbers (Gödelization) had a major influence on the work of Church and Turing.



- ▶ Church-Turing thesis (1936): The functions computable in the intuitive sense are exactly the functions computable by TMs. This means that the terms *algorithm* and *computability* can be defined by TMs.
- ▶ Further computation models are defined in the following years. It turns out that *all these models are equivalent*. This is how the Church-Turing thesis develops into a generally recognized thesis.
- ▶ 1960s: Only now definitions of the term “algorithm” emerge that are characterized by computing technology (e.g. RAMs as a mathematical model for real computers).
- ▶ 1970: Extended Church-Turing thesis: Any meaningful computational model can be efficiently simulated (polynomial run-time difference) on a probabilistic TM.

- ▶ 1990s: Investigation of new computational models such as quantum computers and DNA computers. It is assumed that certain problems can be solved much faster than in classical computation models.

However, all problems that can be solved with these computation models can also be solved with classical computation models.

The mathematical characterization of the terms *algorithm* and *computability* is one of the most important scientific achievements of the 20th century.

2.2

Alphabets, Words, Formal Languages

Learning Objectives

After this section you should

1. understand and be able to use the basic terms introduced.
2. understand the dyadic representation.
3. be able to convert safely between decimal notation, binary notation, and dyadic notation.

Computability – The Computer Science Perspective

We have seen that

- ▶ the question for precise definitions for terms like “algorithms” and “computation” and
- ▶ the question for the limits of these concepts

did not originate from the field of computer technology but from mathematics and the question for axiomatization of mathematical theories.

However, one can also motivate our investigations from a more computer scientific point of view:

What are computers principally capable of and what are their limits?

What are the “things” that computers compute?

They get an input and return an output.

So they compute **functions**.

Input and output are finite sequences of signals which, from a technical point of view, can have a variety of origins: Electricity, electromagnetic waves, quanta, . . .

From a theoretical point of view, we summarize all of these in the concept of **words**.

Definition 2.1 (Alphabets and words)

- ▶ An **alphabet** is a finite, non-empty set.
(e.g. $\{0, 1\}$ or $\{a, b, c\}$)
- ▶ The elements of an alphabet are called **letters** or **symbols**.
- ▶ A **word over an alphabet Σ** is a finite sequence of 0 or more elements from Σ . (e.g. 00110 or *acbb*)
- ▶ We denote the **empty word** (i.e. the word consisting of 0 letters) by ε .

Definition 2.2 (Sets of Words)

Let Σ be an alphabet, $n \geq 0$ and $a_1, a_2, \dots, a_n \in \Sigma$.

- ▶ The **length of a word** $w = a_1 a_2 \cdots a_n$ is $|w| \stackrel{df}{=} n$.
- ▶ $\Sigma^n \stackrel{df}{=} \{w \mid w \text{ is a word over } \Sigma \text{ with } |w| = n\}$
It holds $\Sigma^0 = \{\varepsilon\}$.
- ▶ $\Sigma^* \stackrel{df}{=} \{w \mid w \text{ is a word over } \Sigma\} = \bigcup_{n \geq 0} \Sigma^n$ and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.
- ▶ If $v = b_1 b_2 \cdots b_m$ and $w = c_1 c_2 \cdots c_n$, are words over the alphabet Σ , then the **concatenation** of v and w is defined as $v \cdot w \stackrel{df}{=} b_1 b_2 \cdots b_m c_1 c_2 \cdots c_n$.
- ▶ A **formal language over** Σ is a subset of Σ^* .
- ▶ The **decision problem** of a formal language $L \subseteq \Sigma^*$ is the following task:

Input:	$w \in \Sigma^*$
Output:	1, if $w \in L$
	0, if $w \notin L$

So, our question

What are computers principally capable of and what are their limits?

can be reformulated as

For an alphabet Σ , which functions $\Sigma^* \rightarrow \Sigma^*$ can be computed by computers and which cannot?

Before we start modelling computers, we argue that it does not matter whether we consider functions of the form $\Sigma^* \rightarrow \Sigma^*$, $\mathbb{N} \rightarrow \mathbb{N}$, or $\mathbb{Z} \rightarrow \mathbb{Z}$.

We start with binary alphabets.

Binary Representation of Natural Numbers

Property 2.3

Each natural number $n \geq 1$ can be represented in exactly one way as

$$n = \sum_{i=0}^m a_i \cdot 2^i$$

with $m \in \mathbb{N}$, $a_m = 1$ and $a_0, \dots, a_{m-1} \in \{0, 1\}$.

Proof.

Exercise. □

Definition 2.4 (Binary representation)

- ▶ The **binary representation** of 0 is 0.
- ▶ Let $n \geq 1$ and let $m \in \mathbb{N}$ and $a_0, \dots, a_m \in \{0, 1\}$ with $a_m = 1$ and $n = \sum_{i=0}^m a_i \cdot 2^i$. Then the sequence $a_m a_{m-1} \cdots a_0$ is called **binary representation** of n .
- ▶ Let **bin**: $\mathbb{N} \rightarrow \{0, 1\}^*$ so that $\text{bin}(n)$ denotes the binary representation of a number $n \in \mathbb{N}$.

Note: By Property 2.3, the function bin is well-defined, i.e., for every natural number there exists a binary representation (existence) and there exists only one (uniqueness). In other words: Property 2.3 shows that bin really is a function and is total.

Example 2.5 (Binary representation)

$\text{bin}(0) = 0$, $\text{bin}(1) = 1$, $\text{bin}(2) = 10$, $\text{bin}(3) = 11$, $\text{bin}(4) = 100$,
 $\text{bin}(5) = 101$, ...

The next property shows how to compute binary representations.

Property 2.6

The function bin is injective and $\text{bin}(2n + a) = \text{bin}(n)a$ for $n \geq 1$ and $a \in \{0, 1\}$.

Proof.

Exercise. □

Example 2.7 (Computing the binary representation)

$\text{bin}(54) = \mathbf{110110}$ since

$$54 = 2 \cdot 27 + \mathbf{0}$$

$$27 = 2 \cdot 13 + \mathbf{1}$$

$$13 = 2 \cdot 6 + \mathbf{1}$$

$$6 = 2 \cdot 3 + \mathbf{0}$$

$$3 = 2 \cdot 1 + \mathbf{1}$$

$$1 = 2 \cdot 0 + \mathbf{1}$$

Encoding between Words and Natural Numbers

Recall that by Property 2.6, the binary representation of natural numbers is an injective mapping $\text{bin} : \mathbb{N} \rightarrow \{0, 1\}^*$. Its disadvantage: it is not a bijection.

With the dyadic or k -adic representation, we now learn a *bijection* between \mathbb{N} and Σ^* for arbitrary alphabets Σ .

Property 2.8

Every natural number $n \geq 1$ can be represented in exactly one way as

$$n = \sum_{i=0}^m a_i \cdot 2^i$$

with $m \in \mathbb{N}$ and $a_0, \dots, a_m \in \{1, 2\}$.

Proof.

Exercise. □

Definition 2.9 (Dyadic representation)

$\text{dya} : \mathbb{N} \rightarrow \{1, 2\}^*$ is defined by

$$\text{dya}(0) \stackrel{\text{df}}{=} \varepsilon$$

$$\text{dya}(n) \stackrel{\text{df}}{=} a_m \cdots a_0, \quad \text{if } n \geq 1, \ n = \sum_{i=0}^m a_i \cdot 2^i \text{ and } a_0, \dots, a_m \in \{1, 2\}.$$

The function dya is well-defined by Property 2.8 and total (existence and uniqueness).

Property 2.10

$\text{dya} : \mathbb{N} \rightarrow \{1, 2\}^*$ is bijective.

Proof.

Follows from Definition 2.9.



The dyadic representation thus provides a bijection $\mathbb{N} \rightarrow \Sigma^*$ for all alphabets Σ with $|\Sigma| = 2$, where the letters from Σ are taken as characters for the digits 1 and 2.

We generalize the representation to bijections $\mathbb{N} \rightarrow \Sigma^*$ for all alphabets Σ with $|\Sigma| \geq 2$.

Property 2.11

Let $k \geq 2$. Every natural number $n \geq 1$ can be represented in exactly one way as

$$n = \sum_{i=0}^m a_i \cdot k^i$$

with $m \in \mathbb{N}$ and $a_0, \dots, a_m \in \{1, \dots, k\}$.

Definition 2.12 (k-adic representation)

For $k \geq 2$ is $\text{ad}_k : \mathbb{N} \rightarrow \{1, \dots, k\}^*$ defined by

$$\begin{aligned} \text{ad}_k(0) &\stackrel{\text{df}}{=} \varepsilon \\ \text{ad}_k(n) &\stackrel{\text{df}}{=} a_m \cdots a_0, \quad \text{if } n \geq 1, n = \sum_{i=0}^m a_i \cdot k^i \text{ and } \\ &\quad a_0, \dots, a_m \in \{1, \dots, k\}. \end{aligned}$$

This means that $\text{dya} = \text{ad}_2$.

Property 2.13

For $k \geq 2$ the function $\text{ad}_k : \mathbb{N} \rightarrow \{1, \dots, k\}^*$ is bijective.

The k -adic representation can be determined using the following property.

Property 2.14


Let $k \geq 2$.

1. $dya(2n + a) = dya(n)a$ for $n \geq 0$ and $a \in \{1, 2\}$
2. $dya^{-1}(xa) = 2 \cdot dya^{-1}(x) + a$ for $x \in \{1, 2\}^*$ and $a \in \{1, 2\}$
3. $ad_k(kn + a) = ad_k(n)a$ for $n \geq 0$ and $a \in \{1, \dots, k\}$
4. $ad_k^{-1}(xa) = k \cdot ad_k^{-1}(x) + a$ for $x \in \{1, \dots, k\}^*$, $a \in \{1, \dots, k\}$

Proof.

Exercise. □

Example 2.15 (Determine the k -adic representation)

$$(-3x+4) = 7u$$


If Σ is an alphabet with $k = |\Sigma| \geq 2$, then ad_k yields a bijection $\mathbb{N} \rightarrow \Sigma^*$, where the letters from Σ are taken as characters for the digits $1, \dots, k$.

Therefore, we can now identify natural numbers with words from Σ^* (and thus also words from Σ^* with words from Σ'^* for another alphabet Σ').

In particular, we can use the following encoding $\text{code}_{\Sigma^*} : \Sigma^* \rightarrow \mathbb{N}$ to encode words $w \in \Sigma^*$ by natural numbers.

$$\mathbf{code}_{\Sigma^*}(w) \stackrel{df}{=} \text{ad}_k^{-1}(w)$$

Encodings between Naturals and Integers

We use the bijection $\text{code}_{\mathbb{Z}} : \mathbb{Z} \rightarrow \mathbb{N}$ defined below, to represent integers by natural numbers (and vice versa).

$$\text{code}_{\mathbb{Z}}(x) \stackrel{\text{df}}{=} \begin{cases} 2x & \text{if } x \geq 0 \\ -2x - 1 & \text{otherwise} \end{cases}$$

We denote the inverse function of $\text{code}_{\mathbb{Z}}$ by $\text{code}_{\mathbb{Z}}^{-1}$.

We can now identify \mathbb{Z} and \mathbb{N} , as one can be encoded by the other.

In order to study the limits of computers, we need to investigate which functions $\mathbb{N} \rightarrow \mathbb{N}$ (or alternatively, $\mathbb{Z} \rightarrow \mathbb{Z}$ or $\Sigma^* \rightarrow \Sigma^*$ for any Σ) can be computed by computers and which cannot.

We now argue that also other types of inputs can be encoded as words, natural numbers, and integers.

Encoding of Lists of Natural Numbers

The input or output of an algorithm is often not just a single number (or word), but a list of numbers (or words). However, the latter can be encoded by *one* number. This means that in many places we can restrict ourselves to algorithms whose inputs or outputs are single numbers. This facilitates our investigations.

We now describe how a list of natural numbers can be encoded by a natural number. Let $n \geq 0$ and x_1, \dots, x_n be natural numbers with the following binary representations.

$$\begin{aligned}\text{bin}(x_1) &= a_{1,1}a_{1,2} \cdots a_{1,r_1} \\ \text{bin}(x_2) &= a_{2,1}a_{2,2} \cdots a_{2,r_2} \\ &\vdots \\ \text{bin}(x_n) &= a_{n,1}a_{n,2} \cdots a_{n,r_n}\end{aligned}$$

The encoding of the list (x_1, \dots, x_n) is obtained by writing all bits $a_{i,j}$ twice in the above binary representations and appending the resulting strings one after the other, where the bit sequence 10 serves as the start, separator, and end marker. The encoding of the list is thus the natural number with binary representation

$$\begin{array}{c}
 10 \ a_{1,1} \ a_{1,1} \ a_{1,2} \ a_{1,2} \cdots a_{1,r_1} \ a_{1,r_1} \ 10 \\
 a_{2,1} \ a_{2,1} \ a_{2,2} \ a_{2,2} \cdots a_{2,r_2} \ a_{2,r_2} \ 10 \\
 \vdots \\
 a_{n,1} \ a_{n,1} \ a_{n,2} \ a_{n,2} \cdots a_{n,r_n} \ a_{n,r_n} \ 10.
 \end{array}$$

The encoding of the list (x_1, \dots, x_n) is denoted by $\langle x_1, \dots, x_n \rangle$.

Example: $\langle \rangle = \text{bin}^{-1}(10) = 2$

$$\langle 2 \rangle = \text{bin}^{-1}(10110010) = 178$$

$$\langle 5, 3, 2 \rangle = \text{bin}^{-1}(1011001110111110110010) = 2944946$$

Encoding of Lists of Other Objects

Algorithms not only process natural numbers but also other objects such as integers or words. For objects of type \mathcal{X} suitable encodings

$$\text{code}_{\mathcal{X}} : \mathcal{X} \rightarrow \mathbb{N}$$

will be defined. This allows us to encode a list (x_1, \dots, x_n) of objects of type $x_1 \in \mathcal{X}_1, \dots, x_n \in \mathcal{X}_n$ by the natural number

$$\langle \text{code}_{\mathcal{X}_1}(x_1), \dots, \text{code}_{\mathcal{X}_n}(x_n) \rangle.$$

If the types of the objects x_1, \dots, x_n are clear, we write in abbreviated form

$$\langle x_1, \dots, x_n \rangle.$$

Having clarified that it does not matter which kind of inputs we consider, we start modelling computers.

2.3

While Programs

Learning Objectives

After this section you should

1. understand inductive definitions.
2. know the syntax, semantics and working mechanism of While and Loop programs.
3. understand and be able to apply the concept of While-computability in detail.
4. be able to create simple While programs yourselves and be able to determine the computed function for simple While programs.
5. know the relationship between LOOP and WHILE and understand the role of the Ackermann function.

Programming Languages

Programming languages not only make programming easier, but also offer an *abstraction* of real computers. This means that once a program has been written, it can be transferred to other hardware. A programming language is therefore a mathematical model of a computer, a so-called *computation model*.

The list of known programming languages is long and new ones are added every year.

List of Programming Languages

Wikipedia list of programming languages (as of 07/2024): A.NET, A-0 System, A+, ABAP, ABC, ABC ALGOL, ACC, Accent, Ace Distributed Application Specification Language, Action!, ActionScript, Actor, Ada – ISO/IEC 8652, Adenine, AdvPL, Agda, Agilent VEE, Agora, AlIMMS, Aldor, Alef, ALF, ALGOL 58, ALGOL 60, ALGOL 68, ALGOL W, Alice ML, Alma-0, AmbientTalk, Amiga E, AMPL, Analitik, AngelScript, Apache Pig latin, Apex, APL, App Inventor for Androids visual block language, AppleScript, APT, Arc, ArkTS, ARexx, Argus, Assembly language, AutoHotkey, Autot!, AutoLISP / Visual LISP, Averest, AWW, Axum, B, Babbage, Ballerina, Bash, BASIC, Batch file, bc, BCPL, BeanShell, BETA, BLISS, Blocky, BlooP, Boo, Boomerang, Bosque, C, C-, C++, C*, C#, C/AL, Caché ObjectScript, C Shell, Caml, Cangjie, Carbon, Catrobat, Cayenne, Cecil, CESIL, Céu, Ceylon, CFEngine, Cg, Ch, Chapel, Charm, CHILL, CHIP-8, ChucK, Cilk, Claire, Clarion, Clean, Clipper, CLIPS, CLIST, Clojure, CLU, CMS-2, COBOL, CoboScript – COBOL Scripting language, Cobra, CoffeeScript, ColdFusion, COMAL, COMIT, Common Intermediate Language, Common Lisp, COMPASS, Component Pascal, CONTRAN, Concurrent Pascal, Constraint Handling Rules, Control Language, Coq, CORAL, Coral 66, CoRVision, COWSEL, CPL, Cryptol, Crystal, Csound, Cuneiform, Curl, Curry, Cybil, Cyclone, Cypher Query Language, Cython, CEEMAC, D, Dart, Darwin, DataFlex, Datalog, DATATRIEVE, dBase, dc, DCL, Delphi, DIBOL, DinkC, Dog, Draco, DRAGON, Dylan, DYNAMO, DAX, E, Ease, Easy PL/I, EASYTRIEVE PLUS, ECMAScript, Edinburgh IMP, EGL, Eiffel, ELAN, Elixir, Elm, Emacs Lisp, Emerald, Epigram, EPL, Erlang, es, Escher, ESPOL, Esterel, Etoys, Euclid, Euler, Euphoria, EusLisp Robot Programming Language, CMS EXEC, EXEC 2, Executable UML, Ezhil, F, F#, F*, Factor, Fantom, FAUST, FFP, fish, Fjölfnir, FL, Flavors, Flex, Flix, FlooP, FLOW-MATIC, FOCAL, FOCUS, FOIL, FORMAC, @Formula, Forth, Fortran – ISO/IEC 1539, Fortress, FP, FoxBase/FoxPro, Franz Lisp, Futhark, Game Maker Language, GameMonkey Script, General Algebraic Modeling System, GAP, G-code, GDScript, Genie, Geometric Description Language, GEORGE, OpenGL Shading Language, GNU E, GNU Ubiquitous Intelligent Language for Extensions, Go, Go!, Game Oriented Assembly Lisp, Gödel, Golo, Good Old Mad, Google Apps Script, Gosu, GOTRAN, General Purpose Simulation System, GraphTalk, GRASS, Grasshopper, Groovy, Hack, HAGGIS, HAL/S, Halide, Hamilton C shell, Harbour, Hartmann pipelines, Haskell, Haxe, Hermes, High Level Assembly, High Level Shader Language, Hollywood, HolyC, Hop, Hopscotch, Hope, Hume, HyperTalk, Hy, Io, Icon, IBM Basic assembly language, IBM Informix-4GL, IBM RPG, IDL, Idris, Inform, ISLISP, J, J#, J++, JADE, Jai, JAL, Janus, Janus, JASS, Java, JavaFX Script, JavaScript, Jess, JCL, JEAN, Join Java, JOSS, Joule, JOVIAL, Joy, jq, JScript, JScript .NET, Julia, Jython, K, Kaleidoscope, Karel, KEE, Kixtart, Klerer-May System, KIF, Kojo, Kotlin, KRC, KRL, KRL, KRYPTON, KornShell, Kodu, Kv, LabVIEW, Ladder, LANSa, Lasso, Lava, LC-3, Lean, Legoscript, LIL, LilyPond, Limbo, LINC, Lingo, LINQ, LIS, LISA, Language H, Lisp – ISO/IEC 13816, Lithé, Little b, LLL, Logo, Logtalk, LotusScript, LPC, LSE, LSL, LiveCode, LiveScript, Lua, Lucid, Lustre, LYaPAS, Lynx, M, M4, M#, Machine code, MAD, MAD/I, Magik, Magma, Maple, MAPPER, MARK-IV, Mary, MATLAB, MASM Microsoft Assembly x86, MATH-MATIC, Maude system, Maxima, Max, MaxScript internal language 3D Studio Max, Maya, MDL, Mercury, Mesa, MHEG-5, Microcode, Microsoft Power Fx, MIIS, MIMIC, Mirah, Miranda, MIVA Script, ML, Model 204, Modolica, Malbolge, Modula, Modula-2, Modula-3, Mohol, Mojo, MOO, Mortran, Mouse, MPD, MSL, MUMPS, MuPAD, Mutan, Mystic Programming Language, NASM, Napier88, Neko, Nemerle, NESL, Net.Data, NetLogo, NetRexx, NewLISP, NEWP, Newspeak, NewtonScript, Nial, Nickle, Nim, Nix, NPL, Not eXactly C, Not Quite C, NSIS, Nu, NWScript, NXT-G, oXML, Oak, Oberon, OBJ2, Object Lisp, ObjectLogo, Object REXX, Object Pascal, Objective-C, Obliq, OCaml, occam, occam-rr, Octave, OmniMark, Opa, Opal, Open Programming Language, OpenCL, OpenEdge Advanced Business Language, OpenVera, OpenQASM, OPSS, OptimJ, Orc, ORCA/Modula-2, Oriol, Orwell, Oxygene, Oz, P, P4, P*, ParaSail, PARI/GP, Pascal – ISO 7185, Pascal Script, PCASTL, PCF, PEARL, PeopleCode, Perl, PDL, Pharo, PHP, Pico, PicoLisp, Pict, Pike, PILOT, Pipelines, Pizza, PL-11, PL/0, PL/B, PL/C, PL/I – ISO 6160, PL/M, PL/P, PL/S, PL/SQL, PL360, PLANC, Plankalkül, Planner, PLEX, PLEXIL, Plus, POP-11, POP-2, PostScript, Portable!, POV-Ray SDL, Powerhouse, PowerBuilder – 4GL GUI application generator from Sybase, PowerShell, PPL, Processing, Processing.js, Prograph, Project Verona, Prolog, PROMAL, Promela, PROSE modeling language, PROTEL, Pro*C, Pure, Pure Data, PureScript, PWCT, Python, Q, Q#, Qalb, Quantum Computation Language, QtScript, QuakeC, QPL, QL, R, R++, Racket, Raku, RAPID, Rapira, Ratfiv, Ratfor, rc, Reason, REBOL, Red, Redcode, REFAL, REXX, Ring, ROOP, RPG, RPL, RSL, RTL/2, Ruby, Rust, S, S2, S3, S-Lang, S-PLUS, SA-C, SabreTalk, SAIL, SAKO, SAS, SASL, Sather, Sawzall, Scala, Scheme, Scilab, Scratch, ScratchJr, Script.NET, Sed, Seed7, Self, SenseTalk, SequenC, Serpent, SETL, Short code, SIMPOL, SIGNAL, SIMPLE, SIMSCRIPT, Simula, Simulink, SISAL, SKILL, SLIP, SMALL, Smalltalk, SML, Strongtalk, Snap!, SNOBOL, Snowball, SOL, Solidity, SOPHAEROS, Source, SPARK, Speakeasy, Speedcode, SPIN, SP/k, SPL, SPS, SQL, SQR, Squeak, Squirrel, SR, S/SL, Starlogo, Strand, Stata, Stateflow, Subtext, SBL, SuperCollider, Superplan, SuperTalk, Swift, Swift, SYMPL, T, TACL, TADS, TAL, Tcl, Tea, TECO, TELCOMP, TeX, TEX, TIE, TMG, compiler-company, Tom, Toi, Toppseed, TPU, Trac, TTM, T-SQL, Transcrypt, TTCN, Turing, TUTOR, TXL, TypeScript, Tynker, UBERcode, UCSD Pascal, Umple, Unicorn, Uniface, UNITY, UnrealScript, V, Vala, Verse, Vim script, Viper, Visual DataFlex, Visual DialogScript, Visual FoxPro, Visual J++, Visual LISP, Visual Objects, Visual Prolog, WATFIV, WATFOR, WebAssembly, WebDNA, Whiley, Winbatch, Wolfram Language, Wyvern, X++, X10, xBase++, XBL, XC, xHarbour, XL, Xojo, XOTcl, Xod, XPL, XPLO, XQuery, XSB, XSLT, Xtend, Yorick, YQL, Z++, Z shell, Zebra, ZPL, ZPL2, Zeno, ZetaLisp, Zig, Zonnon, ZOPL, ZPL.

We will study the computation models based on programming languages on the basis of While programs, whose syntax is based on the Python programming language.

In principle, we could also choose another programming language, and we would get the same results. The advantage of While programs is that they only have a limited syntax. This makes it easier for us to carry out mathematical proofs later on.

Programming languages are usually defined in two stages:

1. The **definition of the syntax** determines *which* language element is used and *how*. The syntax clearly determines the set of syntactically correct programs.
2. The **definition of semantics** determines the meaning/way of using the language elements. Semantics thus describes the behavior of syntactically correct programs.

At the latest when programming the compiler, it becomes clear that the precise definition of a programming language is necessary in order to automatically translate source code into machine code.

To define the syntax and semantics of While programs we need the following logical connectors.

Definition 2.16 (logical connectors)

Let a and b be truth values (i.e., true=1 and false=0).

$$\begin{aligned}\neg a &\stackrel{df}{=} \begin{cases} \text{true,} & \text{if } a = \text{false} \\ \text{false} & \text{else} \end{cases} \\ a \wedge b &\stackrel{df}{=} \begin{cases} \text{true,} & \text{if } a = b = \text{true} \\ \text{false} & \text{else} \end{cases} \\ a \vee b &\stackrel{df}{=} \begin{cases} \text{true,} & \text{if } a = \text{true or } b = \text{true} \\ \text{false} & \text{else} \end{cases}\end{aligned}$$

Syntax and Semantics of While Programs

The syntax defines the following language elements: Identifiers, constants, variables, expressions, conditions, statement blocks, function declarations, While programs.

For a better understanding, we define the syntax and semantics of While programs in parallel.

The semantics are defined in boxes like this one.

1. Identifiers:

An identifier is a finite sequence of letters `a,...,z,A,...,Z` and digits `0,...,9` that starts with a letter and is none of the following words:

`not, and, or, if, else, while, for, in, range, def, return, print, False, class, finally, is, None, continue, lambda, as, try, True, from, nonlocal, del, global, with, elif, yield, assert, async, await, import, pass, break, except, raise`

- ▶ Differentiation between upper and lower case letters
- ▶ Used for variable and function names

Examples: `a, b3, ab15cd, house, h0use`

The reserved identifiers are Python keywords. The words `not,...,print` are keywords of While programs.

2. Constants:

- ▶ 0 and -0 are constants.
- ▶ If a is a non-empty, finite sequence of digits $0, \dots, 9$ that does not start with 0, then a and $-a$ are constants.

Constants describe integers in decimal representation.

Examples: 6362, -53, 0, -0

3. Variables:

If a is an identifier, then a is a variable.

The value of a variable is not defined until the first assignment. Afterwards the variable describes an integer (which is also called the assignment of the variable).

Examples: a, b3, ab15cd, house, h0use

4. Expressions: (Inductive Definition)

An expression describes an integer (if it is defined).

BC Basic expressions

- ▶ Each constant and each variable is an expression.

The integer described by the constant or variable.

IS Compound expressions

- ▶ If a and b are expressions, so are $(a + b)$ and $(a - b)$.

a, b defined \Rightarrow expressions $(a + b)$, $(a - b)$ stand for the integers $a + b$, $a - b$.

a or b not defined $\Rightarrow (a + b)$, $(a - b)$ not defined.

- ▶ If f is an identifier, $n \geq 0$, and b_1, \dots, b_n are expressions, then $f(b_1, \dots, b_n)$ is an expression (**function call**).

The semantics will be defined under “Function declarations”.

Examples: $w5$, 7 , $f(b)$, $f()$, $h((a+b))$, $(f((3+g(9))), 2) - 7$

5. Conditions: (Inductive definition)

BC If a, b are expressions, then $(a < b)$, $(a \leq b)$, $(a > b)$, $(a \geq b)$, $(a == b)$, $(a != b)$ are conditions.

a, b defined \Rightarrow the conditions are assigned the usual truth values *true* or *false*.
 a or b not defined \Rightarrow conditions not defined.

IS If a, b conditions, so are $(\text{not } a)$, $(a \text{ and } b)$ and $(a \text{ or } b)$.

a, b defined \Rightarrow the conditions have the truth values
 $\neg a, a \wedge b, a \vee b$.
 a not defined \Rightarrow $(\text{not } a)$, $(a \text{ and } b)$, $(a \text{ or } b)$ not defined.
 a true, b not defined \Rightarrow $(a \text{ or } b)$ true, $(a \text{ and } b)$ not defined.
 a false, b not defined \Rightarrow $(a \text{ and } b)$ false, $(a \text{ or } b)$ not defined

Examples: $(0 == 1)$, $((5 - a) < f(3))$, $(\text{not } (a < f(3)))$

6. Instruction blocks: (Inductive definition)

BC Assignments

- ▶ If a is a variable and b an expression, then the following is an instruction block.

$$a = b$$

b defined \Rightarrow the value of b is assigned to the variable a b not defined \Rightarrow computation of instruction block undefined.

Examples: $a = 2$, $a = b$, $a = ((d + e) - f(10))$

IS Compound instruction blocks

- ▶ **Consecutive execution:** If $n \geq 2$ and s_1, \dots, s_n are instruction blocks, then the following is an instruction block.

 S_1 s_2

•
•
•

 S_n

(s_1, \dots, s_n are placed one below the other with the same indentation)

The instruction blocks are executed in the order s_1, \dots, s_n . If one of the s_i is not defined, then the computation of the complete instruction block is not defined.

Example:

```
a = (a - 10000000000000000000000000000000)
```

$$b = f(a)$$
$$c = ((b + b) - a)$$

- **If-instructions:** For a condition b and instruction blocks s_1, s_2 the following two constructs are also instruction blocks:

if b :

s_1

else:

s_2

if b :

s_1

(s_1 and s_2 are indented further than if and else)

if b :

s_1

else:

s_2

b true $\Rightarrow s_1$ is executed.

b false $\Rightarrow s_2$ is executed.

b not defined \Rightarrow computation not defined

if b :

s_1

b true $\Rightarrow s_1$ is executed.

b false \Rightarrow next instruction is executed.

b not defined \Rightarrow computation not defined

Example:

```
a = 0
if (a <= a):
    a = (a - 10)
    b = f(a)
    c = ((b + b) - a)
if (1 == 1):
    a = 1
else:
    a = 2
a = 0
```


- ▶ **While-loops:** For a condition b and an instruction block s the following is an instruction block.

```
while  $b$ :
```

```
     $s$ 
```

(s is indented further than `while`)

The condition b is evaluated first. If it is not defined, the computation of the while-loop is not defined. If b is false, the while-loop is ended and the next instruction is executed. If b is true, s is executed first and then the while-loop is started again (with the evaluation of the condition b). The instruction block s is therefore repeated as long as b is true. If b is never false, the computation of the while-loop is not defined.

Example:

```
while ( $a \neq 0$ ):
```

```
     $a = (a - 1)$ 
```

```
     $b = (b + a)$ 
```

- **For-loops:** For a variable i , expressions a_1 , a_2 and an instruction block s the following is an instruction block:

```
for  $i$  in range( $a_1, a_2$ ):
```

```
     $s$ 
```

(s is indented further than for)

The working mechanism corresponds to the following instruction block, where j, k are two variables not used in the rest of the program.

```
 $j = a_1$ 
```

```
 $k = a_2$ 
```

```
while ( $j < k$ ):
```

```
     $i = j$ 
```

```
     $s$ 
```

```
     $j = (j + 1)$ 
```

Example:

```
for  $i$  in range(1,3):
```

```
     $i = 1$ 
```

7. Function declarations:

- **Function declaration without initialization:** Let f be an identifier, $n \geq 0$, a_1, \dots, a_n pairwise distinct variables, s an instruction block and c an expression. Then the following two constructs are declarations of the function f .

```
def f( $a_1, \dots, a_n$ ):  
    s  
    return c
```

```
def f( $a_1, \dots, a_n$ ):  
    return c
```

Semantics of the function call $f(c_1, \dots, c_n)$, where c_1, \dots, c_n are expressions: If one of the expressions c_1, \dots, c_n is not defined, the computation of the function call is not defined. Otherwise, s (if available) is executed, whereby the variables a_1, \dots, a_n are assigned the values c_1, \dots, c_n and the values of the remaining variables are not defined.⁷ If this computation is defined, the value of the function call $f(c_1, \dots, c_n)$ is defined as the value of c (at the end of the computation). Otherwise, the computation of the function call is not defined.

⁷Local variable visibility

- **Function declaration with initialization:** Let $f, n, a_1, \dots, a_n, s, c$ be as above. Let b_1, \dots, b_m with $m \geq 0$ the variables used in s and c without a_1, \dots, a_n . Then the following two constructs are declarations of the function f .

```
def f( $a_1, \dots, a_n$ ):  
    [ $b_1, \dots, b_m$ ] = [0, ..., 0]  
    s  
    return c
```

```
def f( $a_1, \dots, a_n$ ):  
    [ $b_1, \dots, b_m$ ] = [0, ..., 0]  
    return c
```

The above lists $[0, \dots, 0]$ contain m zeros.

The semantics correspond to the following function declarations:

```
def f( $a_1, \dots, a_n$ ):  
     $b_1 = 0$   
     $\vdots$   
     $b_m = 0$   
    s  
    return c
```

```
def f( $a_1, \dots, a_n$ ):  
     $b_1 = 0$   
     $\vdots$   
     $b_m = 0$   
    return c
```

Examples:

```
def f(x,y):  
    z = 0  
    while (x > 0):  
        z = (z + y)  
        x = (x - 1)  
    return z
```

```
def g(x,y):  
    [i,j,z] = [0,0,0]  
    for i in range(x,y):  
        j = (j + i)  
        z = (j + j)  
    return z
```

```
def h(x,y):  
    [] = []  
    return (x + y)
```

8. While-programs:

A While-Program is a finite non-empty sequence of function declarations with the following properties:

1. Each function called in the program is declared with the appropriate arity.
2. If $\text{def } f(\dots):$ and $\text{def } g(\dots):$ are two function declarations, the identifiers f and g are distinct.
3. If $\text{def } f(\dots):$ is a function declaration and v a variable of the program, then the identifiers f and v are distinct.

If

```
def f( $a_1, \dots, a_n$ ):  
    ...
```

is the last function declaration in a While program P , then **the function computed by P** is the function $\psi_P : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with:

$$\psi_P(x_1, \dots, x_n) \stackrel{\text{df}}{=} \begin{cases} f(x_1, \dots, x_n), & \text{if this is defined} \\ \text{not defined} & \text{else} \end{cases}$$

Visibility of variables

The semantics of function calls are defined in such a way that variables are only visible in the corresponding function.

Example:

```
def h(x):  
    if (0 == 0):  
        a = 0  
        if (x == 0):  
            b = 0  
            c = g(0)  
    return a      # no error because a was defined  
  
def g(y):  
    z = b          # program termination as b not defined  
    return z
```

Definition 2.17 (While computability)

- ▶ An **algorithm** is a While program.
- ▶ A function $\varphi : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with $n \geq 0$ is called **While computable** if there is a While program P such that $D_{\psi_P} = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ A function $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is called **While computable** if there is a While program P such that $D_{\psi_P} \cap \mathbb{N}^n = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ **WHILE** $\stackrel{\text{df}}{=} \{\varphi \mid \varphi : \mathbb{N}^n \rightarrow \mathbb{N} \text{ is While computable}\}$.

We thus have our first mathematical characterization of the terms *algorithm* and *computability*.

Self calls of functions

We name the call of a function within its own declaration **self-call** or also **recursion**.

The possibility of self-calls or mutual calls of several functions is called **recursive programming**.

Example:

```
def f(x):  
    y = 1  
    if (x > 0):  
        y = (f((x - 1)) + f((x - 1)))  
    return y
```

```

def f(x):
    y = 1
    if (x > 0):
        y = (f((x - 1)) + f((x - 1)))
    return y

```

By the definition of the program, it computes the function $g : \mathbb{Z} \rightarrow \mathbb{Z}$ defined via

$$g(x) = \begin{cases} 1 & \text{if } x \leq 0 \\ 2 \cdot g(x - 1) & \text{else.} \end{cases}$$

We now prove that for all $n \in \mathbb{N}$ it holds $g(n) = 2^n$ via induction.

Proof.

BC: For $k = 0$ it holds $2^k = 1$ and $g(k) = 1$.

IS: Let's assume that for some $k \in \mathbb{N}$ it holds $g(k) = 2^k$. We have to prove that then also $g(k + 1) = 2^{k+1}$.

This is so: $g(k + 1) = 2 \cdot g(k) \stackrel{\text{ind. hypothesis}}{=} 2 \cdot 2^k = 2^{k+1}$. □

Causes of undefinedness in While-computable functions:

- ▶ endless loops, e.g.:

```
def f(x):  
    while (x != 0):  
        x = (x - 1)  
    return x
```

$\Rightarrow f(x) = 0$ for $x \geq 0$ and $f(x)$ not defined for $x < 0$.

- ▶ certain recursive calls, e.g.:

```
def f(x):  
    x = f((x + 1))  
    return x
```

$\Rightarrow f(x)$ not defined for all $x \in \mathbb{Z}$.

- ▶ access to undefined variables, e.g.:

```
def f(x):  
    return z
```

$\Rightarrow f(x)$ not defined for all $x \in \mathbb{Z}$.

We eliminate these causes in the following computation model.

Definition 2.18 (Loop program)

A **Loop program** is a While program with the following properties:

1. The program does not contain any while-loops.
2. Only functions that are declared above can be called by each function. In particular, no self-calls are permitted.
3. The program only contains function declarations with initialization.

Definition 2.19 (Loop computability)

- ▶ A function $\varphi : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with $n \geq 0$ is called **Loop-computable** if there is a Loop program P such that $D_{\psi_P} = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ A function $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is called **Loop-computable** if there is a loop program P such that $D_{\psi_P} \cap \mathbb{N}^n = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ **LOOP** $\stackrel{df}{=} \{\varphi \mid \varphi : \mathbb{N}^n \rightarrow \mathbb{N} \text{ is Loop-computable}\}$.

Special constructs for While and Loop programs

In order to simplify working with While and Loop programs we define in addition to the previously defined syntax the following “comfort elements”:

- ▶ If a_1, \dots, a_n are expressions with $n \geq 1$, then `print(a_1, \dots, a_n)` is a statement block. It outputs the values of expressions to the screen for testing purposes.
- ▶ Comments are marked with #, i.e., the text between # and the end of the line is ignored.

Example:

```
def f(a,b):           # computes the sum of two numbers
    print(a,b)        # outputs the values of a and b
    return (a + b)    # return the result
```

Notes on working with While and Loop programs

- ▶ We have defined While and Loop programs in such a way that they form sublanguages of the Python 3 programming language. In particular, While and Loop programs can be executed directly in Python 3.
- ▶ You will find a syntax checker online that checks whether a given source code is a While or Loop program.
- ▶ In addition to the syntax of While and Loop programs, we also allow the insertion/omission of spaces, provided that this does not change the semantics of the program. (e.g. `a=(b+c)` instead of `a = (b + c)`)
- ▶ Do not use tabs and avoid empty lines in functions (use lines with `#` instead).
- ▶ When accessing undefined expressions, Python sometimes throws exceptions.

Example 2.20

Loop program for the multiplication in \mathbb{Z} .

```
def prodZ(x,y):  
    [i,z] = [0,0]           # Initialization  
    if (x < 0):  
        x = (0 - x)         # remove negative sign from x  
        y = (0 - y)         # and transfer to y  
    for i in range(0,x):    # x loop executions  
        z = (z + y)         # y is added x times to z  
    return z
```


For the input $x = -3$ and $y = 12$, we note the situation after each line. This results in the following flow chart.

Action	x	y	i	z	Comment
	-3	12	n.d.	n.d.	
<code>[i,z] = [0,0]</code>	-3	12	0	0	
<code>if (x < 0):</code>	-3	12	0	0	true
<code>x = (0 - x)</code>	3	12	0	0	
<code>y = (0 - y)</code>	3	-12	0	0	
<code>for i in range(0,x):</code>	3	-12	0	0	$i = 0$
<code>z = (z + y)</code>	3	-12	0	-12	
<code>for i in range(0,x):</code>	3	-12	1	-12	$i = 1$
<code>z = (z + y)</code>	3	-12	1	-24	
<code>for i in range(0,x):</code>	3	-12	2	-24	$i = 2$
<code>z = (z + y)</code>	3	-12	2	-36	
<code>for i in range(0,x):</code>	3	-12	2	-36	end of loop
<code>return z</code>	3	-12	2	-36	return -36

Example 2.21

While program for the rounding off division in \mathbb{Z} .

$$\text{divZ}(x, y) \stackrel{\text{df}}{=} \begin{cases} \lfloor x/y \rfloor, & \text{if } y \neq 0 \\ \text{n.d.}, & \text{if } y = 0, \end{cases}$$

where for $y \neq 0$

$$\lfloor x/y \rfloor \stackrel{\text{df}}{=} \begin{cases} \text{largest } z \in \mathbb{Z} \text{ with } y \cdot z \leq x, & \text{if } y > 0 \\ \lfloor (-x)/(-y) \rfloor, & \text{if } y < 0 \end{cases}^8$$

⁸Generally, for a real number α we denote by $\lfloor \alpha \rfloor$ the least integer that is greater or equal α . Analogously $\lceil \alpha \rceil$ denotes the greatest integer that is less or equal α .

```

def divZ(x,y):
    if (y != 0):          # return undefined if y == 0
        if (y < 0):
            y = (0 - y)   # remove neg. sign from y
            x = (0 - x)   # and transfer to x
        z = 0
        if (x < 0):       # z = min(0,x)
            z = x

            # here y>0 and y*z <= x
        while (prodZ(y,z) <= x): # find max z with y*z<=x
            z = (z + 1)
        z = (z - 1)        # correction, counted too far
    return z

```

Example 2.22

A While program with several function declarations. Three functions are declared in the program, which are defined for $n, m, i \in \mathbb{Z}$ as follows.

$$\text{modZ}(x, y) \stackrel{df}{=} x - (y \cdot \text{divZ}(x, y))$$

$$\text{divisors}(x) \stackrel{df}{=} \begin{cases} \text{number of divisors of } x, & \text{if } x \geq 1 \\ 0, & \text{else} \end{cases}$$

$$\text{prime}(n) \stackrel{df}{=} \begin{cases} n\text{-th prime number}, & \text{if } n \geq 0 \\ 2, & \text{else} \end{cases}$$

For $x \in \mathbb{N}$ and $y \in \mathbb{N}^+$, $\text{modZ}(x, y)$ is the remainder of the division x/y .

In the definition of `prime`, we consider 2 to be the 0th prime. The While program itself computes the function `prime`.

```

def modZ(x,y):
    return (x - prodZ(y,divZ(x,y)))

def divisors(x):
    k = 0                                # set counter to 0
    for i in range(1,(x + 1)):          # i = 1, ..., x
        if (modZ(x,i) == 0):
            k = (k + 1)                  # increment counter
    return k

def prime(n):
    k = 0
    m = 2                                # 0-th prime number
    while (n > k):                        # in {2,3,...,m} there are the
        m = (m + 1)                      # 0-th, 1-st, ..., k-th prime
        if (divisors(m) == 2):           # (k+1)-th prime found
            k = (k + 1)
    return m

```

Example 2.23 (While Programs for the Factorial)

```
def fac(x):          # fac(x) = x!, if x>=1
    y = 1            # fac(x) = 1, otherwise
    for i in range(1,(x + 1)):
        y = prodZ(i,y)
    return y
```

```
def fac2(x):          # fac2(x) = x!, if x>=1
    if (x > 1):       # fac2(x) = x, otherwise
        x = prodZ(x,fac2((x - 1))) # Self-call of fac2
    return x
```

```
def fac3(x):          # fac3(x) = x!, if x>=1
    y = 1            # fac3(x) = n.d., otherwise
    while (x != 1):
        y = prodZ(x,y)
        x = (x - 1)
    return y
```


⇒ 3 different functions are computed

Example 2.24

Which functions are computed by the following While programs?

```
def f1(x,y):  
    z = 0  
    while (x > y):  
        x = (x + 2)  
        y = (y + 3)  
        z = (z + 1)  
    return z
```

```
def f2(x):  
    while (x>1):  
        z=x  
        x=0  
        while (z>1):  
            z=(z-2)  
            x=(x+1)  
        if (z>0):  
            x=((x+x)+x)  
            x=((x+x)+4)  
    return 0
```

$$(-3x+4) = 7u$$



Example 2.25

Which functions are computed by the following Loop programs?

```
def f3(x):  
    [y,i]=[0,0]  
    for i in range (0,x):  
        y = (y + x)  
    return y
```

```
def f4(x):  
    [i]=[0]  
    for i in range (0,x):  
        x = (x + 2)  
    return x
```

```
def f5(x):  
    [i]=[0]  
    for i in range (0,x):  
        x = (x + x)  
    return x
```

$$(-3x+4)=7u$$


Theorem 2.26 (Loop vs While computability)

1. Every Loop-computable function $\varphi : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with $n \geq 0$ is While-computable.
2. Every Loop-computable function $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is While-computable.
3. $\text{LOOP} \subsetneq \text{WHILE}$.

Proof idea:

- ▶ Loop programs are special While programs
- ▶ Strictness of inclusion: Loop programs are always total, but While programs are not

Proof.

Since Loop programs are special While programs, statements 1 and 2 and $\text{LOOP} \subseteq \text{WHILE}$ hold.

Loop programs do not contain infinite loops and infinite cycles of function calls are not possible in Loop programs. It is also not possible to access undefined variables.

This means that Loop programs halt on every input with a defined result. So all Loop-computable functions $\mathbb{N}^n \rightarrow \mathbb{N}$ are total.

Consequently, $g : \mathbb{N} \rightarrow \mathbb{N}$ with $g(x) \stackrel{df}{=} \text{n.d.}$ is not Loop-computable.

However, the function g is While-computable:

```
def g(x):  
    while (0 == 0):  
        x = 0  
    return 0
```

Thus $g \in \text{WHILE} - \text{LOOP}$ and therefore $\text{LOOP} \neq \text{WHILE}$. □

Ackermann Function

Theorem 2.26 raises the following question:

Does $\text{LOOP} = \{f \in \text{WHILE} \mid f \text{ is total}\}$ hold?

In 1926 Wilhelm Ackermann proved that the following function $a : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ (also called **Ackermann function**) is total and While-computable, but is not Loop-computable.

$$a(n, m) \stackrel{\text{df}}{=} \begin{cases} m + 1 & \text{if } n = 0 \\ a(n - 1, 1) & \text{if } n > 0 \text{ and } m = 0 \\ a(n - 1, a(n, m - 1)) & \text{if } n > 0 \text{ and } m > 0 \end{cases}$$

Theorem 2.27

The Ackermann function is a total function in WHILE – LOOP.

Without proof.

The Ackermann function grows extremely quickly:

$$a(1, 3) = 5$$

$$a(2, 3) = 9$$

$$a(3, 3) = 61$$

$$a(4, 3) \approx 10^{10^{19728}}$$

2.4

Register Machines

Learning Objectives

After this section you should

1. understand the structure and working mechanism of RAMs.
2. know the RAM instructions and their semantics.
3. understand and be able to apply the concept of RAM computability in detail.
4. be able to create simple RAM programs yourselves and be able to determine the computed function for simple RAM programs.
5. know the differences between While and Python programs.
6. be able to outline the translation of RAMs into While programs.
7. understand the idea of dynamic programming.

We want to develop a mathematical model for *computers*.

Reason: real computers are very different in terms of

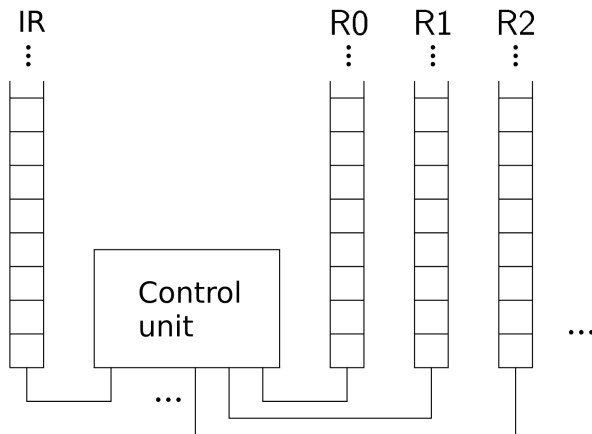
- ▶ Processor
- ▶ Architecture
- ▶ Computing power
- ▶ Memory capacity
- ▶ ...

Advantage of mathematical models:

Abstraction of technical details leads to better understanding.

With *random access machines (RAM)* we get to know another computation model. It captures the essential properties of real computers, but abstracts from technical details.

Structure of a RAM:



- ▶ each of the registers IR, R0, R1, ... contains a natural number
- ▶ the control unit has a program
(finite list of instructions, numbered with 0, 1, 2, ...).
- ▶ different RAMs differ only in their program

Differences to real computers:

- ▶ RAM has *infinitely many* registers
- ▶ RAM registers have *unlimited number of bits*
(can store arbitrarily large natural numbers)
- ▶ more *technical details*

When creating the model, we abstract from the specific number of registers and bits per register.

Working mechanism of a RAM:

- ▶ works *in cycles*
- ▶ exactly one instruction is executed in each cycle (the one whose number is in the instruction register IR)
- ▶ Start: 0 is in IR, i.e., we start at the instruction with number 0
- ▶ Stop: if the instruction register points to the stop instruction or there is no instruction with the corresponding number

$[Ri]$ = Contents of register Ri

$[IR]$ = Contents of register IR

Definition 2.28 (modified difference)

The function $md: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is defined as follows.

$$x \dot{-} y = md(x, y) \stackrel{df}{=} \begin{cases} x - y, & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases}$$

The following instructions can be used in RAM programs:

Instructions (syntax)	Effect (semantics)
Transport instructions	
$R_i \leftarrow R_j$ (direct address)	$[R_i] := [R_j], [IR] = [IR] + 1$
$R_i \leftarrow RR_j$ (indir. Address.)	$[R_i] := [R[R_j]], [IR] = [IR] + 1$
$RR_i \leftarrow R_j$ (indir. address.)	$[R[R_i]] := [R_j], [IR] := [IR] + 1$
Arithmetic instructions	
$R_i \leftarrow k$	$[R_i] := k, [IR] = [IR] + 1$
$R_i \leftarrow R_j + R_k$	$[R_i] := [R_j] + [R_k], [IR] = [IR] + 1$
$R_i \leftarrow R_j - R_k$	$[R_i] := [R_j] \div [R_k], [IR] = [IR] + 1$

Instructions (syntax)	Effect (semantics)
Jump instructions	
GOTO m	$[IR] := m$
IF $R_i = 0$ GOTO m	$[IR] := \begin{cases} m, & \text{if } [R_i] = 0 \\ [IR] + 1 & \text{else} \end{cases}$
IF $R_i > 0$ GOTO m	$[IR] := \begin{cases} m, & \text{if } [R_i] > 0 \\ [IR] + 1 & \text{else} \end{cases}$
Stop instruction	
STOP	end of computation

Example 2.29

Computation of the multiplication using a RAM: $x \cdot y$

Starting situation: $[R0] = x$, $[R1] = y$, $[Ri] = 0$ for $i \geq 2$

Final situation: $[R0] = x \cdot y$

Auxiliary register: $[R2] = \text{intermediate total}$, $[R3] = 1$

Idea: $x \cdot y = \underbrace{x + x + \dots + x}_{y\text{-times}}$

Program:

0	$R3 \leftarrow 1$
1	IF $R1 = 0$ GOTO 5
2	$R2 \leftarrow R2 + R0$
3	$R1 \leftarrow R1 - R3$
4	GOTO 1
5	$R0 \leftarrow R2$
6	STOP

For the computation $5 \cdot 3$ we denote the situation after cycle i :

Cycle i	[IR]	instruction to be executed	[R0]	[R1]	[R2]	[R3]
	0	$R3 \leftarrow 1$	5	3	0	0
1	1	IF $R1 = 0$ GOTO 5	5	3	0	1
2	2	$R2 \leftarrow R2 + R0$	5	3	0	1
3	3	$R1 \leftarrow R1 - R3$	5	3	5	1
4	4	GOTO 1	5	2	5	1
5	1	IF $R1 = 0$ GOTO 5	5	2	5	1
6	2	$R2 \leftarrow R2 + R0$	5	2	5	1
7	3	$R1 \leftarrow R1 - R3$	5	2	10	1
8	4	GOTO 1	5	1	10	1
9	1	IF $R1 = 0$ GOTO 5	5	1	10	1
10	2	$R2 \leftarrow R2 + R0$	5	1	10	1
11	3	$R1 \leftarrow R1 - R3$	5	1	15	1
12	4	GOTO 1	5	0	15	1
13	1	IF $R1 = 0$ GOTO 5	5	0	15	1
14	5	$R0 \leftarrow R2$	5	0	15	1
15	6	STOP	15	0	15	1

Example 2.30

Transfer maximum of $[R10], [R11], [R12], \dots, [R[R1]]$ to $R0$.

Starting situation: input is in $R1, R10, R11, \dots, R[R1]$,
Content of other registers is arbitrary

Idea: Consider R_i for
 $i = [R1], [R1] - 1, \dots, 12, 11, 10$;
the maximum of $[R_i]$ and $[R0]$
is moved to $R0$

```
0  R0 ← 0
1  IF R1 ≤ 9 GOTO 6      // disallowed instruction
2  IF RR1 ≤ R0 GOTO 4    // disallowed instruction
3  R0 ← RR1
4  R1 ← R1 - 1           // disallowed instruction
5  GOTO 1
6  STOP
```

We are now eliminating the disallowed instructions:

0	0	$R0 \leftarrow 0$	
1	1	$R2 \leftarrow 9$	
2		$R3 \leftarrow R1 - R2$	$(a \leq b \Leftrightarrow a \div b = 0)$
3		IF $R3 = 0$ GOTO 11	
2	4	$R3 \leftarrow RR1$	
5		$R3 \leftarrow R3 - R0$	$(a \leq b \Leftrightarrow a \div b = 0)$
6		IF $R3 = 0$ GOTO 8	
3	7	$R0 \leftarrow RR1$	
4	8	$R2 \leftarrow 1$	
9		$R1 \leftarrow R1 - R2$	
5	10	GOTO 1	
6	11	STOP	

RAMs provide a second mathematical characterization of the term *computability*.

Definition 2.31 (RAM computability)

- ▶ Let M be a RAM. A function $\varphi: \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is **computed by M** if for all $x_1, \dots, x_n \in \mathbb{N}$:

$$\varphi(x_1, \dots, x_n) = \begin{cases} y, & \text{if } M \text{ starting with } [IR] = 0, [R0] = x_1, \dots, [R(n-1)] = x_n, [Ri] = 0 \\ & \text{for } i \geq n \text{ stops after a finite number of steps with } [R0] = y \\ \text{n.d.} & \text{else} \end{cases}$$

- ▶ A function $\varphi: \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is called **RAM-computable**, if there is a RAM that computes φ .
- ▶ **RAM** $\stackrel{\text{df}}{=} \{\varphi \mid \varphi: \mathbb{N}^n \rightarrow \mathbb{N} \text{ is RAM-computable}\}.$

Remark 2.32

Each RAM computes an infinite number of functions, namely for every $n \geq 0$ exactly one n -ary function. For example, the RAM from Example 2.29 computes among others

- ▶ *the 2-ary function $\text{prod} : \mathbb{N}^2 \rightarrow \mathbb{N}$ with $\text{prod}(x, y) \stackrel{\text{df}}{=} x \cdot y$,*
- ▶ *the 1-ary function $C_0^1 : \mathbb{N} \rightarrow \mathbb{N}$ with $C_0^1(x) \stackrel{\text{df}}{=} 0$ and*
- ▶ *the 3-ary function $f : \mathbb{N}^3 \rightarrow \mathbb{N}$ with $f(x, y, z) \stackrel{\text{df}}{=} (x \cdot y) + z$.*

Encoding of Lists of Integers in While Programs

We will later show $\text{RAM} = \text{WHILE}$. To do this, we will first consider how to represent and process lists of *integers* in While programs and process them.

With the help of $\text{code}_{\mathbb{Z}}$ and the list encoding of natural numbers a list of integers (y_1, \dots, y_n) can be encoded by the following natural number:

$$\langle y_1, \dots, y_n \rangle \stackrel{\text{df}}{=} \langle \text{code}_{\mathbb{Z}}(y_1), \dots, \text{code}_{\mathbb{Z}}(y_n) \rangle.$$

The en- and decoding of such lists can be implemented using corresponding While programs. Therefore, the computation power does not change if the use of lists is permitted in While programs.

We are now adding lists and other language constructs to While programs. These make programming easier but do not change the computation power. We call the extended programs *Python programs*.

Python Programs

- ▶ All constructs from While programs are allowed
- ▶ Allows the use of lists (see next page)
- ▶ No strict bracketing of expressions and conditions
- ▶ multiplication $*$, division $//$, modulo $%$ are allowed (correspond to the functions prodZ , divZ , modZ)
- ▶ functions without a return value are permitted
- ▶ All constructs for which we have shown that they can be simulated in While programs are allowed

Theorem 2.33

A function $f : \mathbb{Z}^n \rightarrow \mathbb{Z}$ or $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is While-computable if and only if it can be computed by a Python program.

Proof.

All new language constructs in Python programs can be simulated in While programs. □

Usage of Lists in Python Programs

- ▶ Lists are denoted in the form `[3,-7,8]`
- ▶ `[]` creates the empty list
- ▶ `a = [3,-7,8]` creates the list `[3,-7,8]` and assigns a the *reference* to this list
- ▶ `b = a` also assigns b the *reference* to `[3,-7,8]` (so a and b point to the same object)
- ▶ `a[2]` is the 2nd element of the list, where the count starts at 0 (e.g. `b = a[2]` or also `a[2]=10000`)
- ▶ `a[3:6]` creates the partial list from the 3rd to 5th element of a , first index is at 0 (e.g. `b=a[3:6]` or `a[3:6]=[3,4,5]`)
- ▶ `len(a)` returns the number of elements in the list a
- ▶ `a + b` generates the list that is created by writing the lists a and b one behind the other
- ▶ `a += b` appends the elements of the list b to the list a
- ▶ Details: Python Language Reference

Example 2.34 (Usage of Lists)

The **Fibonacci function** $\text{fib} : \mathbb{Z} \rightarrow \mathbb{Z}$ is inductively defined by:

$$\text{fib}(n) \stackrel{\text{df}}{=} \begin{cases} 1, & \text{if } n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Named after the Italian mathematician Fibonacci (around 1170–1240), who was looking for a formula for the multiplication of rabbits. His model:

- ▶ Initially (after 0 months) there is one female rabbit.
- ▶ After 2 months, a female rabbit has babies for the first time and then again every month.
- ▶ There is exactly one female young in each litter.
- ▶ Female rabbits never die.

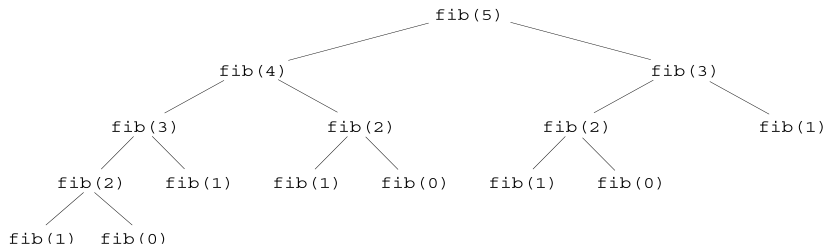
For $n \geq 0$ you can easily compute:

$\text{fib}(n)$ = number of female rabbits after n months.

The following Python program computes the Fibonacci function.

```
def fib(n):
    if (n <= 1):
        n = 1
    else:
        n = fib(n - 1) + fib(n - 2)
    return n
```

Two self-calls, which in turn cause self-calls again:



Short algorithm, but multiple calls (e.g. 5 times `fib(1)`).
The following Python program is much faster.

```
def fibFast(n):  
    f = [1,1]                                # list [fib(0), fib(1)]  
    if (n <= 1):                             # handle negative inputs  
        n = 1  
    for i in range(2,n+1):                   # i = 2,3,...,n  
        f += [f[i-1] + f[i-2]]               # append fib(i) to the list  
    return f[n]                              # read fib(n) from the list
```

Non-recursive algorithm. Each value is only computed once:

$$f[0] \rightarrow f[1] \rightarrow f[2] \rightarrow f[3] \rightarrow f[4] \rightarrow f[5]$$

Storing and reusing intermediate solutions is also known as *dynamic programming*.

Theorem 2.35

RAM \subseteq WHILE.

Proof idea:

- Step-by-step simulation of the RAM, whereby the registers are stored in two lists u and v

Proof.

Let $\varphi \in \text{RAM}$ and $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$.

Assume φ is computed by the following RAM M :

0	b_0
1	b_1
\vdots	\vdots
k	b_k

W.l.o.g.: $b_k = \text{STOP}$ and $b_i \neq \text{STOP}$ for $i < k$.

We simulate M with a Python program:

- ▶ the variable `ir` stores contents of instruction register IR
- ▶ the list `u` contains the indices and the list `v` contains the corresponding contents of the previously used registers
- ▶ if a register R_i that has not yet been used is accessed, then i is appended to u and 0 to v (i.e. $[R_i] = 0$)

Example: $u = [5, 2]$ and $v = [24, 7]$ means that R_5 and R_2 have been used so far, $[R_5] = 24$ and $[R_2] = 7$.

We use the following functions “read” and “write” to access the R_i registers. They perform the search in the lists and extend the lists if necessary.

Attention: “read” and “write” change the lists visible in the main program, as lists are passed to functions by reference.

```

def read(u,v,a):                                # returns the content of Ra
    i = 0
    while (i < len(u) and u[i] != a): # search for index a
        i = i + 1
    if (i == len(u)):                          # Extend lists
        u += [a]
        v += [0]
    return v[i]                                # returns content of Ra

def write(u,v,a,b):                             # write b to Ra
    i = 0
    while (i < len(u) and u[i] != a): # search for index a
        i = i + 1
    if (i == len(u)):                         # extend lists
        u += [a]
        v += [0]
    v[i] = b                                  # write b in Ra

```

```

def phi(x1,x2,...,xn):
    u = [0,1,...,n-1]
    v = [x1,x2,...,xn]
    ir = 0
    while (ir < k):
        if (ir == 0):
            s0
        if (ir == 1):
            s1
            ⋮
        if (ir == k - 1):
            sk-1
    return read(u,v,0)

```

} simulation of at
least one cycle of M

The instruction block s_i simulates the RAM instruction b_i as follows:

If b_i is instruction ...	then s_i is instruction block ...
$Ra \leftarrow Rb$	$i = \text{read}(u,v,b)$ $\text{write}(u,v,a,i)$ $ir = ir + 1$
$Ra \leftarrow R Rb$	$i = \text{read}(u,v,b)$ $j = \text{read}(u,v,i)$ $\text{write}(u,v,a,j)$ $ir = ir + 1$
$R R a \leftarrow Rb$	$i = \text{read}(u,v,b)$ $j = \text{read}(u,v,a)$ $\text{write}(u,v,j,i)$ $ir = ir + 1$
$Ra \leftarrow b$	$\text{write}(u,v,a,b)$ $ir = ir + 1$
$Ra \leftarrow Rb + Rc$	$i = \text{read}(u,v,b) + \text{read}(u,v,c)$ $\text{write}(u,v,a,i)$ $ir = ir + 1$

If b_i is instruction ...	then s_i is instruction block ...
$Ra \leftarrow Rb - Rc$	$i = \text{read}(u,v,b) - \text{read}(u,v,c)$ if ($i < 0$): $i = 0$ $\text{write}(u,v,a,i)$ $ir = ir + 1$
GOTO a	$ir = a$
IF $Ra = 0$ GOTO b	if ($\text{read}(u,v,a) == 0$): $ir = b$ else: $ir = ir + 1$
IF $Ra > 0$ GOTO b	if ($\text{read}(u,v,a) > 0$): $ir = b$ else: $ir = ir + 1$

Thus $D_{\psi_{\text{phi}}} \cap \mathbb{N}^n = D_{\varphi}$ and $\psi_{\text{phi}}(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_{\varphi}$.

Then $\varphi \in \text{WHILE}$ by Definition 2.17.



2.5

Mini-While Programs

Learning Objectives

After this section you should

1. know the main differences between While programs and Mini-While programs.
2. be able to outline the translation of While programs into RAMs.

Goal: we want to prove $\text{WHILE} \subseteq \text{RAM}$.

Idea: we take an intermediate step via a variant of While programs with a reduced range of functions

While program \Rightarrow Mini-While program \Rightarrow RAM program

The generation of intermediate code is also carried out with real compilers in order to

- ▶ simplify the translation process,
- ▶ be able to optimize better, and
- ▶ increase the portability of the compiler.

Syntax of Mini-While-Programs

1. **Identifiers, constants, variables:** as for While-Programs
2. **Expressions:**
 - (a) a is constant or variable $\Rightarrow a$ is expression
 - (b) a, b are variables $\Rightarrow (a + b), (a - b)$ are expressions
3. **Conditions:** a is variable $\Rightarrow (a > 0)$ is condition

Syntax of Mini-While-Programs ctd.

4. **Instruction blocks:** (inductive definition)

BC *Assignments*

- ▶ a is Variable, b is Constant $\Rightarrow a = b$ is instr. block
- ▶ a, b, c are variables $\Rightarrow a = (b + c), a = (b - c)$ are instr. blocks

IS *Compound instruction blocks*

- ▶ s_1, \dots, s_n are instr. blocks $\Rightarrow \left. \begin{array}{c} s_1 \\ \vdots \\ s_n \end{array} \right\}$ is instr. block
- ▶ a is variable, s is instr. block $\Rightarrow \left. \begin{array}{c} \text{while}(a > 0) : \\ s \end{array} \right\}$ is instr. block

5. **Function declarations:** same as for While programs.

6. **Mini-While programs:** consist of exactly one function decl. without initialization and without function calls.

Semantics of Mini-While Programs

same ;-)

Definition 2.36 (Mini-While Computability)

- ▶ $\varphi : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with $n \geq 0$ is called **Mini-While computable**, if there is a Mini-While program P such that $D_{\psi_P} = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is called **Mini-While computable**, if there is a Mini-While program P such that $D_{\psi_P} \cap \mathbb{N}^n = D_\varphi$ and $\psi_P(x_1, \dots, x_n) = \varphi(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_\varphi$.
- ▶ **MINIWHILE** $\stackrel{\text{df}}{=} \{\varphi \mid \varphi : \mathbb{N}^n \rightarrow \mathbb{N} \text{ is Mini-While computable}\}.$

Theorem 2.37

WHILE \subseteq MINIWHILE.

Proof sketch.

We eliminate all constructs of While programs step by step in the subsequent order that are not allowed in Mini-While programs.

1. For-loops
2. If-instructions
3. Conditions not of the form $(a > 0)$, where a is a variable
4. Function calls and thus also the declaration of several functions in a program
5. Assignments that do not have the form $a = d$, $a = (b + c)$ or $a = (b - c)$ for a constant d and variables b, c

The step 4 is the most difficult. Among others, recursive self-calls of functions must be eliminated. To do this, we store the variable assignments and return addresses of the nested computation in lists and simulate the calls of functions in a while-loop. The lists are encoded in a natural number using $\langle y_1, \dots, y_n \rangle$. □

Theorem 2.38

WHILE \subseteq RAM.

Proof sketch.

- ▶ Let $g \in \text{WHILE} \subseteq \text{MINIWHILE}$ with $g : \mathbb{N}^n \rightarrow \mathbb{N}$.
- ▶ There is a Mini-While program P with $D_{\psi_P} \cap \mathbb{N}^n = D_g$ and $\psi_P(x_1, \dots, x_n) = g(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_g$.
- ▶ P can be simulated on a RAM as follows:
- ▶ Each variable is represented by 2 registers, which store the absolute value and the sign.
- ▶ Assignments of the Mini-While program can now be carried out directly on the registers.
- ▶ Signed addition/subtraction is simulated by corresponding RAM programs (case study for $x \geq 0, y \geq 0, |x| \geq |y|$).
- ▶ While-loops: simulated with conditional jump instructions.



This means that the computations models considered so far are equivalent.

Corollary 2.39

RAM = WHILE = MINIWHILE.

Proof.

Follows from Theorems 2.35, 2.37, and 2.38. □

Corollary 2.40

Every RAM-computable function $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ can be computed by a RAM without indirect addressing.

Proof.

Follows from Corollary 2.39 and the proof of theorem 2.38, since the simulation described there does not use indirect addressing. □

2.6

Turing Machines

Learning Objectives

After this section you should

1. understand the structure and working mechanism of TMs.
2. understand and be able to apply the concept of Turing computability in detail.
3. be able to create simple TM programs yourselves and be able to determine the function computed by simple TM programs.
4. understand the formal definition of a k -tape TM as a quintuple.
5. be able to sketch the translation of TMs into While programs.
6. be able to sketch the translation of RAMs into TMs.

2.6.1

The Search for the Right Definition

- ▶ 1926: David Hilbert (1862-1943) comes across the primitive-recursive functions in his search for a proof of the continuum hypothesis. They can be defined from simple functions by inserting one into the other and recursion.



- ▶ 1926: Hilbert's students Gabriel Sudan (1899-1977) and Wilhelm Ackermann (1896-1962) construct total functions that are computable in the intuitive sense but not primitive-recursive. The primitive-recursive functions therefore do not fully capture the concept of computability.

- ▶ 1931–1934: Alonzo Church (1903–1995) develops the lambda calculus and shows that all common number-theoretic functions are lambda-definable.



- ▶ 1934: Church makes the following conjecture to Kurt Gödel (1906–1978): A total function is computable in the intuitive sense if it is lambda-definable. Gödel rejects the conjecture as completely inadequate.

- ▶ 1934: Gödel defines the concept of the general recursive function in his lectures at Princeton.



- ▶ 1936: Church formulates his conjecture using Gödel's terms: A total function is computable in the intuitive sense if and only if it is general-recursive.

Together with Stephen Kleene (1909-1994), he shows the equivalence between lambda-calculus and general-recursive functions.



- ▶ Despite these indications, Gödel continues to reject the conjecture. He is not sure whether the general recursive functions cover all possibilities of recursion. Gödel expects a definition that *obviously* covers the computable.

- ▶ 1936: At the age of 22, Alan Turing (1912-1954) presents his definition of the Turing machine and the functions it can compute. Gödel enthusiastically accepts it as the correct definition, as it obviously captures the concept of finite computation.



- ▶ Gödel says later in a lecture:
“Turing shows that the computable functions are exactly those, for which you can build a machine with a finite number of parts that performs the following:
If you write a number x on a piece of paper, insert it into the machine and operate the crank, the machine will stop after a finite number of revolutions and the function value $f(x)$ is written on the piece of paper.”

2.6.2

Definition and Operation of Turing Machines

The Turing Machine (TM) was designed by Alan Turing in 1936. It was conceived as a model of a human being who, equipped with pencil and paper, works according to certain rules.

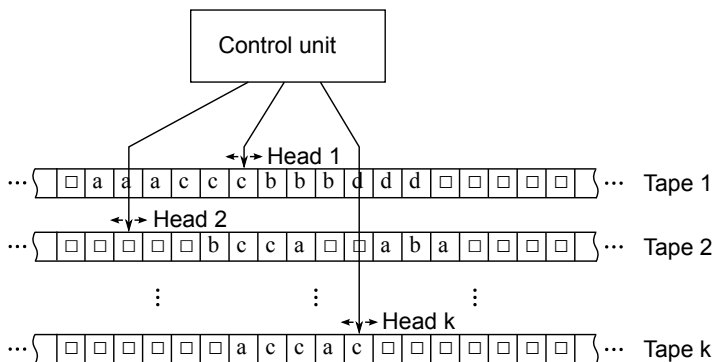
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTSCHIEDUNGSPROBLEM

By A. M. TURING.

⋮

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q_1, q_2, \dots, q_n , which will be called " m -configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the r -th, bearing the symbol $\mathfrak{S}(r)$ which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m -configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m -configuration q_n and the scanned symbol $\mathfrak{S}(r)$.

Structure of a TM



Control unit:

- ▶ is always in a state from the finite set of states S
- ▶ a program f controls the activity
- ▶ start state s_0 , stop state s_1

Tapes:

- ▶ divided into cells, each cell contains a symbol from Σ
- ▶ infinite on both sides
- ▶ empty symbol $\square \in \Sigma$, separator symbol $*$ $\in \Sigma$

Heads:

- ▶ one per tape
- ▶ can change field content and move to neighboring field if necessary

Working mechanism:

- ▶ TMs work in cycles
- ▶ start in a special initial situation
- ▶ stop if state s_1 is reached

For each cycle, depending on

- ▶ the state and
- ▶ the k symbols on which the heads are located

the TM can do the following at the same time:

- ▶ keep or change the state
- ▶ change the k symbols under the heads
- ▶ move each head by a maximum of one cell

This behavior is described by the **transition function** f .

$$f: (S \setminus \{s_1\}) \times \Sigma^k \rightarrow S \times \Sigma^k \times \{L, O, R\}^k$$

For $f(s, a_1, \dots, a_k) = (s', a'_1, \dots, a'_k, \sigma_1, \dots, \sigma_k)$ we also write:

$$sa_1 \cdots a_k \rightarrow s' a'_1 \cdots a'_k \sigma_1 \cdots \sigma_k$$

Definition 2.41

Let $k \geq 1$. A **k-tape Turing machine** (short: k -tape TM or TM) is a quintuple (Σ, S, f, s_0, s_1) with:

- ▶ Σ is a finite set with $\square, * \in \Sigma$ (alphabet)
- ▶ S is a finite set (set of states)
- ▶ $f: (S \setminus \{s_1\}) \times \Sigma^k \rightarrow S \times \Sigma^k \times \{L, O, R\}^k$ is a total function (transition function)
- ▶ $s_0 \in S$ (start state)
- ▶ $s_1 \in S$ (stop state)

Remark 2.42 (Omit unnecessary instructions)

In programs of k -tape TMs, we can omit unnecessary instructions for the sake of simplicity.

If the program of the machine $M = (\Sigma, S, f, s_0, s_1)$ is missing the instruction

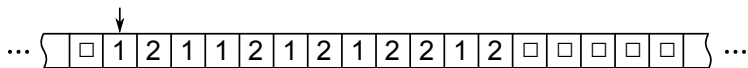
$$sa_1 \cdots a_k \rightarrow$$

the machine behaves according to the imaginary instruction

$$sa_1 \cdots a_k \rightarrow s_1 a_1 \cdots a_k O \cdots O.$$

This means that even incomplete programs always describe total transition functions.

Example 2.43 (TM – incrementing dyadic numbers)



$s_0 1 \rightarrow s_0 1 R$ // run to the right

$s_0 2 \rightarrow s_0 2 R$

$s_0 \square \rightarrow s_2 \square L$ // $s_2 = \text{carryover } 1$

$s_2 1 \rightarrow s_1 2 O$

$s_2 2 \rightarrow s_2 1 L$

$s_2 \square \rightarrow s_1 1 O$ // if we leave the number with a carryover,
write a 1 in front

A word $a_1 \cdots a_n$ with $a_1, \dots, a_n \in \Sigma$ is called **symmetric** (or also **palindrome**) if $a_1 \cdots a_n = a_n \cdots a_1$.

Example 2.44

A TM with one tape is to determine whether a word $x \in \{a, b\}^*$ is symmetric.

Head is on the first symbol of x at the start. If x is (not) symmetric, a (b) should be on the tape at the stop.

Idea: Compare first letter with last letter, delete both,
run to the beginning, repeat as long as letters are the same

Meaning of the states:

s_0/s_1 = start/stop state

s_a/s_b = a/b memorized, go right

s'_a/s'_b = go one step to the left and test a/b

s_2/s_3 = test positive/negative, move to the left

$s_0 \square \rightarrow s_1 a O$	// even length, stop positively
$s_0 a \rightarrow s_a \square R$	// memorize a
$s_0 b \rightarrow s_b \square R$	// memorize b
$s_a a \rightarrow s_a a R$	// go right with a being memorized
$s_a b \rightarrow s_a b R$	// go right with a being memorized
$s_a \square \rightarrow s'_a \square L$	// at end of word, move on step back, test a
$s_b a \rightarrow s_b a R$	// go right with b being memorized
$s_b b \rightarrow s_b b R$	// go right with b being memorized
$s_b \square \rightarrow s'_b \square L$	// at end of word, move on step back, test b
$s'_a a \rightarrow s_2 \square L$	// test is positive, go back left
$s'_a b \rightarrow s_3 \square L$	// test is negative, go back left
$s'_a \square \rightarrow s_1 a O$	// word removed, odd length, pos. stop
$s'_b b \rightarrow s_2 \square L$	// test is positive, go back left
$s'_b a \rightarrow s_3 \square L$	// test is negative, go back left
$s'_b \square \rightarrow s_1 a O$	// word removed, odd length, pos. stop

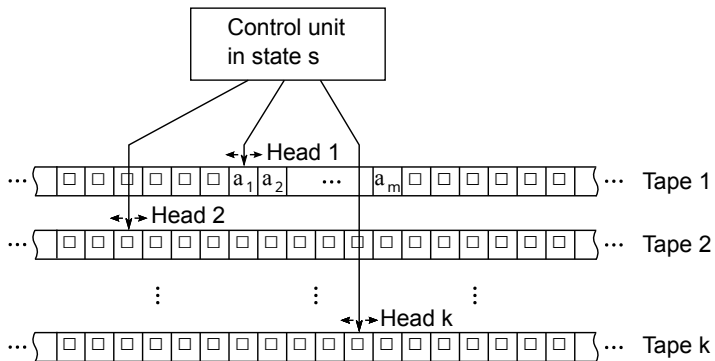
$s_2 a \rightarrow s_2 a L$ // go back left, continue to test
 $s_2 b \rightarrow s_2 b L$ // go back left, continue to test
 $s_2 \square \rightarrow s_0 \square R$ // at beginning of word, new round

 $s_3 a \rightarrow s_3 \square L$ // go back left to stop
 $s_3 b \rightarrow s_3 \square L$ // go back left to stop
 $s_3 \square \rightarrow s_1 b O$ // at beginning of word, negative stop

Definition 2.45 (Initial situation)

Let $M = (\Sigma, S, f, s_0, s_1)$ be a k -tape TM, $a_1, \dots, a_m \in \Sigma \setminus \{\square\}$ and $s \in S$. $\mathbf{M}(s, \mathbf{a}_1 \cdots \mathbf{a}_m)$ denotes the following situation of M :

- ▶ the control unit is in state s
- ▶ the 1st tape contains the word $a_1 a_2 \cdots a_m$, where the head is at a_1 (if $m = 0$, the head is on a \square)
- ▶ the tapes $2, \dots, k$ only contain \square



Definition 2.46 (Turing computability)

- Let $M = (\Sigma, S, f, s_0, s_1)$ be a TM and $\Sigma_1 \subseteq \Sigma \setminus \{\square, *\}$. A function $\varphi : (\Sigma_1^*)^n \rightarrow \Sigma_1^*$ with $n \geq 0$ is **computed by M**, if for all $x_1, \dots, x_n \in \Sigma_1^*$ holds:

$$\varphi(x_1, \dots, x_n) = \begin{cases} \text{n.d., if } M \text{ does not stop after starting in} \\ \quad M(s_0, x_1 * x_2 * \dots * x_n) \\ y, \quad \text{if } M \text{ stops after starting in} \\ \quad M(s_0, x_1 * x_2 * \dots * x_n) \text{ and tape 1} \\ \quad \text{contains } \dots \square y \square \dots \text{ with } y \in \Sigma_1^* \\ \varepsilon, \quad \text{otherwise} \end{cases}$$

- A function $\varphi : (\Sigma_1^*)^n \rightarrow \Sigma_1^*$ with $n \geq 0$ is called **Turing-computable** if there is a TM M that computes φ .

- ▶ A function $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ with $n \geq 0$ is **Turing computable** if there is a TM M computing φ in dyadic representation. I.e., M computes the function $\varphi_{\text{dya}} : (\{1, 2\}^*)^n \rightarrow \{1, 2\}^*$ with $\varphi_{\text{dya}}(x_1, \dots, x_n) \stackrel{\text{df}}{=} \text{dya}(\varphi(\text{dya}^{-1}(x_1), \dots, \text{dya}^{-1}(x_n)))$.
- ▶ **TM** $\stackrel{\text{df}}{=} \{\varphi \mid \varphi : \mathbb{N}^n \rightarrow \mathbb{N} \text{ is Turing computable}\}$

Remark 2.47

Each TM $M = (\Sigma, S, f, s_0, s_1)$ computes a function $\varphi : (\Sigma_1^)^n \rightarrow \Sigma_1^*$ for every $n \geq 0$ and every $\Sigma_1 \subseteq \Sigma \setminus \{\square, *\}$.*

Remark 2.48

If $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ is Turing-computable, then there exists a TM $M = (\Sigma, S, f, s_0, s_1)$, such that for all $x_1, \dots, x_n \in \mathbb{N}$:

- ▶ If $\varphi(x_1, \dots, x_n) = y$, then M stops after starting in $M(s_0, dya(x_1)*dya(x_2)*\dots*dya(x_n))$, where tape 1 has the content $\dots \square dya(y) \square \dots$.
- ▶ If $\varphi(x_1, \dots, x_n)$ is not defined, then M does not stop after starting in $M(s_0, dya(x_1)*dya(x_2)*\dots*dya(x_n))$.

Example 2.49

Let $S : \mathbb{N} \rightarrow \mathbb{N}$ with $S(x) \stackrel{df}{=} x + 1$. Example 2.43 yields $S \in \text{TM}$.

Theorem 2.50

TM \subseteq WHILE.

Proof idea:

- ▶ the tapes of the TM are represented by lists
- ▶ a simple Python program simulates the TM step by step

Proof.

Let $f \in \text{TM}$ with $f : \mathbb{N}^n \rightarrow \mathbb{N}$. Then there is a k -tape TM $M = (\Sigma, S, f', s_0, s_1)$, which computes f in dyadic representation.

Let $S = \{s_0, \dots, s_r\}$ and $\Sigma = \{a_0, \dots, a_s\}$, where $a_0 = \square$, $a_1 = 1$, $a_2 = 2$ and $a_3 = *$.

The Python program uses the following variables:

- ▶ s : index of the state of M
- ▶ b_1, \dots, b_k : lists for the tapes $1, \dots, k$, where
 $b_i[j]$ is index of the character in cell j on tape i
- ▶ h_1, \dots, h_k : positions of the heads on the tapes $1, \dots, k$

```

def g(x1,...,xn):
    store dya(x1)*...*dya(xn) in b1[0],b1[1],...
    s = 0                                # state  $s_0$ 
    b2 = b3 = ... = bk = [0] # tapes have only blank characters
    h1 = h2 = ... = hk = 0    # heads point to 0th list element
    while (s != 1):           # as long as stop state not reached
        simulate the behavior of M using if-statements
        if heads run over the lists, then append [0]
    w = result on tape 1 (cf. Def. Turing computability)
    return dya-1(w)

```

For the function g computed by the above Python program it holds: $D_f = D_g \cap \mathbb{N}^n$ and $f(x_1, \dots, x_n) = g(x_1, \dots, x_n)$ for all $(x_1, \dots, x_n) \in D_f$.

This shows $f \in \text{WHILE}$. □

Example 2.51 (Simulation of an instruction)

Let $M = (\Sigma, S, f', s_0, s_1)$ be the k -tape TM from the proof of Theorem 2.50.

If $s_2 a_5 a_3 \rightarrow s_3 a_7 a_1$ RL is an instruction of M , then we simulate this by:

```
if (s==2 and b1[h1]==5 and b2[h2]==3): # situation s2 a5 a3
    s = 3                                # go into state s3
    b1[h1] = 7                           # write a7 on tape 1
    b2[h2] = 1                           # write a1 on tape 2
    h1 = h1 + 1                          # on tape 1 go right
    h2 = h2 - 1                          # on tape 2 go left
    if h1==len(b1):                      # if 'end' of tape 1 is reached
        b1=b1+[0]
    if h2==-1:                           # if 'end' of tape 2 is reached
        h2=0
        b2=[0]+b2
```

Theorem 2.52

RAM \subseteq TM.

Proof idea:

- ▶ it is sufficient to simulate a RAM without indirect addressing by a $(k+2)$ -tape TM, where k basically is the max. index of register used
- ▶ registers are stored in the tapes $1, \dots, k+1$
- ▶ tape $k+2$ is used for arithmetic instructions

Proof.

Let $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ be a function of the set RAM.

$\xrightarrow{2.40}$ φ is computed by a RAM M without indirect addressing.

$k \stackrel{df}{=} \max(\{i \mid Ri \text{ occurs in the program of } M\} \cup \{n-1\})$

Only the registers R_0, \dots, R_k are required for the simulation.

We simulate M using the following $(k+2)$ -tape TM M' :

- ▶ the contents of the registers R_0, \dots, R_k are transferred to the tapes $1, \dots, k+1$ in dyadic representation
- ▶ IR is stored in the state of M'

Initial situation: $M'(s_0, \text{dya}(x_1) * \text{dya}(x_2) * \dots * \text{dya}(x_n))$

At the start, M' generates the initial situation of the RAM M , i.e. M' writes $\text{dya}(x_1), \dots, \text{dya}(x_n)$ to the tapes $1, \dots, n$.

Simulation of an instruction of M :

- ▶ *transport instruction*: by copying the corresponding tapes
- ▶ *arithmetic instruction*: Create the sum or modified difference of the corresponding tapes on tape $k + 2$; copy the content of tape $k + 2$ to the tape of the target register
- ▶ *jump instruction*: test whether corresponding tape is empty (testing for 0 as $\text{dya}(0) = \varepsilon$); execute jump by changing the state
- ▶ *stop instruction (or jump out of program)*: go to state s_1

Final situation: tape 1 has content $\dots \square \text{dya}([R0]) \square \dots$

Thus M' computes the function φ in dyadic representation.

Hence $\varphi \in \text{TM}$. □

2.6.3

Universal Turing machines

Turing machines are able to simulate themselves. A TM U that can simulate every TM, is called **universal**.

When given the program of a TM M and the input word w as input, U simulates the computation of M on w step by step.

In contrast to early computing machines, the function of universal Turing machines is not limited to a special task. Alan Turing formulated this important idea as early as 1936:

It is possible to invent a single machine which can be used to compute any computable sequence.

The concept of keeping both the program and the data in memory is an example of the tremendous power of ideas.

The first realization took place at the end of the 1940s by John von Neumann and computers are still based on Turing's idea today.

2.7

Fundamental Theorem of the Theory of Algorithms

Learning Objectives

After this section you should

1. be able to explain the main theorem and the Church-Turing thesis.
2. know examples of computable functions and be able to prove the computability of simple functions.

Theorem 2.53 (Fundamental Theorem of the Theory of Algorithms)

RAM = WHILE = MINIWHILE = TM

Proof.

Follows from 2.39, 2.50, and 2.52. □

All meaningful computability terms can be included here.

This supports the thesis that the computability terms examined so far capture what is *intuitively computable*.

Thesis 2.54 (Church-Turing thesis/thesis by Church)

Every intuitively computable function is Turing-computable.

This assertion is not provable in principle, since the term *intuitively computable function* cannot be formalized.

We use 2.54 as an axiom and extend computability to functions $\mathbb{N}^n \rightarrow \mathbb{N}^m$.

Definition 2.55

Let $m, n \geq 0$.

- ▶ A function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is **computable** if $f \in \text{TM}$.
- ▶ A function $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$ with $m \neq 1$ is **computable function** if the following function $f' : \mathbb{N}^n \rightarrow \mathbb{N}$ is computable:

$$f'(x_1, \dots, x_n) \stackrel{\text{df}}{=} \langle f(x_1, \dots, x_n) \rangle.$$

According to the Church-Turing thesis, we can now also admit verbally formulated algorithms. However, the formulation must indicate the possibility of an implementation, e.g. by a While program.

Theorem 2.56

The following functions $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ are computable.

$$\text{sum}(x, y) \stackrel{\text{df}}{=} x + y \qquad \text{md}(x, y) \stackrel{\text{df}}{=} \begin{cases} x - y, & \text{if } x > y \\ 0 & \text{else} \end{cases}$$

$$\text{prod}(x, y) \stackrel{\text{df}}{=} x \cdot y \qquad \text{div}(x, y) \stackrel{\text{df}}{=} \begin{cases} \lfloor \frac{x}{y} \rfloor, & \text{if } y > 0 \\ x & \text{otherwise} \end{cases}$$

$$\text{exp}(x, y) \stackrel{\text{df}}{=} x^y \text{ (where we define } 0^0 = 1)$$

$$\text{log}(x, y) \stackrel{\text{df}}{=} \begin{cases} \lfloor \log_x(y) \rfloor, & \text{if } x \geq 2, y > 0 \\ x & \text{else} \end{cases}$$

Proof.

The functions `sum` and `md` are already available in RAMs. Our While programs for `prodZ` and `divZ` can be easily adapted to the definitions of `prod` and `div`. Proving `exp` and `log` to be computable is left as an exercise. □

Property 2.57

The class of computable functions is closed under consecutive execution. I.e. if $g : \mathbb{N}^k \rightarrow \mathbb{N}^n$ and $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$ are computable, then $f \circ g$ is also computable. As a reminder: $(f \circ g)(x) \stackrel{\text{df}}{=} f(g(x))$.

Proof.

by simply combining two While programs. □

2.8

Runtime of Algorithms

Learning Objectives

After this section you should

1. know the definition of the runtime for While programs, RAMs and TMs.
2. know the difference between “runtime for concrete inputs” and “runtime limit with regard to the input length”.
3. be able to estimate the runtime of simple algorithms in a meaningful way and specify it in O -notation.

To compare algorithms, we need a sensible measure for their run-time. To do this, we proceed as follows for each of our computation models (WHILE/RAM/TM):

1. Definition of the exact runtime for *concrete inputs*
2. Definition of runtime bounds with respect to the *input* lengths

We define the **length of a number** $x \in \mathbb{Z}$ as the length of its dyadic representation, omitting the sign:

$$|x| \stackrel{df}{=} |\text{dya}(\text{abs}(x))|$$

We write the absolute value of x as **abs(x)**, so that there is no confusion with the length of x .

For $x \in \mathbb{Z}$ we will use the following inequality that can be proven by induction.

$$2^{|x|} - 1 \leq \text{abs}(x) \leq 2^{|x|+1} - 2$$

Definition 2.58 (Runtime funct. for While programs)

Let M be a While program that computes a total function $f: \mathbb{Z}^m \rightarrow \mathbb{Z}$ (or $f: \mathbb{N}^m \rightarrow \mathbb{N}$). The runtime function $t_M: \mathbb{Z}^m \rightarrow \mathbb{N}$ (or $t_M: \mathbb{N}^m \rightarrow \mathbb{N}$) is defined by

$t_M(x_1, \dots, x_m) \stackrel{df}{=} \text{number of computation steps until } M \text{ stops when given } x_1, \dots, x_m \text{ as inputs,}$

where the computation steps are counted as follows:

- ▶ 1 computational step for the operations $+$ and $-$, for comparisons ($<$, \leq , $>$, \geq , $==$, $!=$), for logical operations (not, and, or), for value assignments ($=$), for initializations ($[b_1, \dots, b_m] = [0, \dots, 0]$) and for return.
- ▶ The number of computation steps for $\text{for } i \text{ in range}(a_1, a_2)$:
 s
is the sum of the computation steps needed for a_1 , a_2 , and s .
(The latter depend on the number of loop iterations)

Remark 2.59

The runtimes defined in Definition 2.58 are in part not realistic. A real computer can e.g. not compute the sum of two n -digit numbers in one step. A more realistic estimate for the number of steps for computing $x + y$ would be $\min(|x|, |y|)$.

However, assuming it to cost only one computation step simplifies runtime analyses, is not uncommon in the theory of algorithms, and is—in some sense—not way off as we will see later. Moreover, as long as we only work with numbers that fit into a register, addition indeed only requires one step.

Definition 2.60 (Runtime function for RAMs)

Let M be a RAM which computes a total function $f : \mathbb{N}^m \rightarrow \mathbb{N}$. The associated runtime function $t_M : \mathbb{N}^m \rightarrow \mathbb{N}$ is defined by

$$t_M(x_1, \dots, x_m) \stackrel{df}{=} \text{number of cycles}^9 \text{ until } M \text{ stops when} \\ \text{given } x_1, \dots, x_m \text{ as input}$$

Definition 2.61 (Runtime function for TMs)

Let M be a TM which computes a total function $f : (\Sigma_1^*)^m \rightarrow \Sigma_1^*$ (resp. $f : \mathbb{N}^m \rightarrow \mathbb{N}$). The associated runtime function $t_M : (\Sigma_1^*)^m \rightarrow \mathbb{N}$ (or $t_M : \mathbb{N}^m \rightarrow \mathbb{N}$) is defined by

$$t_M(x_1, \dots, x_m) \stackrel{df}{=} \text{number of cycles until } M \text{ stops when} \\ \text{given } x_1, \dots, x_m \text{ as input}$$

⁹Again, the assumption that addition of and comparisons between “big” numbers occur in one step is a simplification.

Runtime Limit with Regard to the Input Length

We abstract from the concrete input and consider the runtime for all inputs of a certain length.

Definition 2.62 (Runtime limit wrt input length)

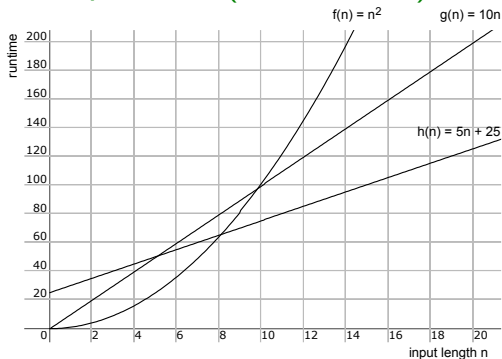
Let $t: \mathbb{N} \rightarrow \mathbb{N}$ be total, M an algorithm of type WHILE/RAM/TM, and $f: A^m \rightarrow A$ a total function with $A \in \{\mathbb{Z}, \mathbb{N}, \Sigma_1^*\}$ computed by M .

- ▶ **M computes f in time t** $\stackrel{df}{\iff}$ for all inputs $x_1, \dots, x_m \in A$ it holds $t_M(x_1, \dots, x_m) \leq t(|x_1| + \dots + |x_m|)$.

We call $|x_1| + \dots + |x_m|$ the **length of the input** x_1, \dots, x_m .

- ▶ **$O(t)$** $\stackrel{df}{=} \{f : \mathbb{N} \rightarrow \mathbb{N} \mid f \text{ total and } \exists n_0, c \in \mathbb{N}^+ \forall n \geq n_0, f(n) \leq c \cdot t(n)\}$. (**Landau notation** or also **O notation**)
- ▶ **M computes f in time O(t)** $\stackrel{df}{\iff}$ there is $t' \in O(t)$ such that M computes f in time t' .

Example 2.63 (O-Notation)



- ▶ no function is greater than the other
- ▶ f grows faster than g, h
- ▶ $f(n) \geq g(n)$ and $f(n) \geq h(n)$ for $n \geq 10$
- ▶ $g(n) \leq 2 \cdot h(n)$ for $n \geq 0$
- ▶ $h(n) \leq g(n)$ for $n \geq 5$
- ▶ g and h have similar growth

$$f, g, h \in O(f), \quad g, h \in O(g) = O(h), \quad f \notin O(g) = O(h).$$

Consider $f(n) = n^2$ and $h(n) = 5n + 25$.

$h \in O(f)$ (\approx “ h grows at most as fast as f ”).

Proof.

to be proven: there exist $n_0 \in \mathbb{N}^+$ and $c \in \mathbb{N}^+$ such that for all $n \geq n_0$ it holds $h(n) \leq c \cdot f(n)$.

We choose $c = 30$ and $n_0 = 1$. Then for all $n \geq n_0$ we have $h(n) = 5n + 25 \leq 30 \cdot n \leq 30 \cdot n^2 = c \cdot n^2 = c \cdot f(n)$. \square

Consider $f(n) = n^2$ and $h(n) = 5n + 25$.

$f \notin O(h)$ (\approx “ f does not grow at most as fast as h ”).

Proof.

to be proven: for all $n_0 \in \mathbb{N}^+$ and all $c \in \mathbb{N}^+$ there exists $n \geq n_0$ so that $f(n) > c \cdot h(n)$.

Let $n_0, c \in \mathbb{N}^+$ be given.

Note that $c \cdot h(n) = 5cn + 25c \leq 30cn$ for all positive n .

But then $f(n) = n^2 > 30cn \geq c \cdot h(n)$ for $n > 30c$. So, when we choose $n = \max(30c, n_0) + 1$, it holds $n \geq n_0$ and $f(n) > c \cdot h(n)$. □

Notation: It is common to write e.g. $O(n^2)$ as a shortcut for $O(f: \mathbb{N} \rightarrow \mathbb{N}, f(n) = n^2)$.

Careful: $f \notin O(g)$ does not imply $g \in O(f)$. Consider e.g.
 $g: \mathbb{N} \rightarrow \mathbb{N}, g(n) = n^2$ and $f: \mathbb{N} \rightarrow \mathbb{N}, f(n) = \begin{cases} n^3 & n \text{ even} \\ n & \text{else.} \end{cases}$

Here $f \notin O(g)$ and $g \notin O(f)$.

Example 2.64

The following While program M computes $|x|$ for an input $x \in \mathbb{Z}$:

```
def length(x):          # Runtime
    if (x < 0):          # 1
        x = (0 - x)     # 2
    y = 0                # 1
    l = 0                # 1
    while(x > y):        # 1    |x| iterations
        y = ((y + y) + 2) # 3    6 steps per iteration
        l = l + 1        # 2    1 step for last test x>y
    return l             # 1
```

Thus $t_M(x) \leq 6 \cdot |x| + 7$.

The algorithm does *not* work in time $1000 \cdot n$ as for input $x = 0$ it holds $t_M(x) > 1000 \cdot |x|$.

But for inputs x with $|x| \geq 1$ we have $t_M(x) \leq 13 \cdot |x|$. So the algorithm works in time t for $t: \mathbb{N} \rightarrow \mathbb{N}$, $t(n) \stackrel{df}{=} \begin{cases} 13 \cdot n & \text{if } n \geq 1 \\ 7 & \text{else.} \end{cases}$

Example Ctd

The O -notation allows to avoid the tedious counting of computation steps:

The Algorithm works in time $O(n)$:

Proof.


Reminder: the algorithm works in time t for $t: \mathbb{N} \rightarrow$

$$\mathbb{N}, t(n) \stackrel{\text{df}}{=} \begin{cases} 13 \cdot n & \text{if } n \geq 1 \\ 7 & \text{else.} \end{cases}$$


So we have to prove $t \in O(n)$.

Choose $n_0 = 1$ and $c = 13$. Then for all $n \geq n_0$ it holds $t(n) = 13n = cn$. □

Example 2.65 (Runtime of a RAM)

$$(-3x+4) = 7u$$


Example 2.66 (Runtime of a While program)

$$(-3x+4) = 7u$$


Runtime for Python Programs

We define the runtime of the Python constructs we use. In the table, $x, y \in \mathbb{Z}$ and a, b are lists of integers.

Operation	Runtime
Multiplication ($x*y$)	$O(x + y)$
Division ($x//y$)	$O(x + y)$
Modulo ($x\%y$)	$O(x + y)$
Access to a list element	$O(1)$
Assignment of a reference to a list	$O(1)$
<code>len(a)</code>	$O(1)$
<code>a += b</code>	$O(\text{len}(b))$
Create/copy list a	$O(\text{len}(a))$
<code>(a+b)</code>	$O(\text{len}(a) + \text{len}(b))$

Remark 2.67

The runtimes for multiplication, division, and modulo are not realistic in a real setting as addition actually requires time $O(n)$.

However, assuming that addition is possible in one step causes that our estimations of runtimes will be off by at most a factor in $O(n)$.

2.9

Further Computation Models

2.9.1

Partial Recursive Functions

Partial Recursive Functions

Computability can also be defined purely algebraically:

The class **PART** (**partial-recursive functions**) can be defined inductively as follows:

BC PART contains 0-functions (e.g. $f(x, y) \stackrel{df}{=} 0$), the successor function and projections (e.g. $g(x, y, z) \stackrel{df}{=} y$).

IS If $f, g, h \in \text{PART}$, so are:

- ▶ Consecutive executions
(such as $r(x, y) \stackrel{df}{=} h(g(x, y), f(x, y))$)
- ▶ recursions (e.g. $r(x, y, 0) \stackrel{df}{=} f(x, y)$,
 $r(x, y, n + 1) \stackrel{df}{=} g(r(x, y, n), x, y)$)
- ▶ μ -functions (such as $r(x) = \mu f(x) \stackrel{df}{=} \text{smallest } n \text{ with } f(x, n) = 0 \text{ and } f(x, 0), \dots, f(x, n - 1) \text{ defined}$)

If the use of the μ operator is prohibited, this results in the class **PRIM** (**primitive-recursive functions**).

One can show:

1. $\text{PART} = \text{WHILE}$
2. $\text{PRIM} = \text{LOOP}$

In particular, $\text{PRIM} \subsetneq \text{PART}$.
(because of the Ackermann function)

2.9.2

Quantum Computers

With the help of mathematical computation models one can study systems even before they are technically feasible.

Quantum mechanics (developed around 1900–1930):

States such as 0 and 1 can overlap,
a classical state is only assumed during the measurement

Richard Feynman (1982):

Classical computers cannot efficiently simulate quantum mechanical systems.



Idea: Quantum computers should use quantum mechanical superpositions, to process data *in parallel*.

Model of a Quantum Computer

All possible states of the memory are available at the same time (superposition).

Each state of the memory has an *individual amplitude*. It determines the probability with which the state is measured.

On a quantum computer with 100 bits, you can easily compute

- ▶ a superposition of all $2^{100} \approx 10^{30}$ memory states
- ▶ and then process these in each step *simultaneously*.

⇒ *massive parallelism*

Restriction: Quantum mechanics does not allow classical instructions, but only unitary transformations

⇒ *Quantum programs = sequences of unitary transformations*

The development of quantum computers is still in its infancy.

We know:

- ▶ A function can be computed by a quantum computer if and only if it is Turing-computable.
- ▶ Quantum computers can simulate classical computers (in about the same runtime).
- ▶ Quantum computers can factorize numbers quickly. For classical computers, no fast factorization algorithm is known (the security of the RSA cryptosystem depends on it).
- ▶ Quantum computers are probably only faster than classical computers for special computations. In particular, it is assumed that many optimization problems (e.g. round trip problem) **cannot** be solved much faster by quantum computers than by classical computers.

2.10

Decidability

Learning Objectives

After this section you should

1. be able to define and apply the terms decidability, reducibility, and characteristic function, with mathematical precision.
2. be able to give examples of decidable and undecidable sets.
3. be able to determine and prove the decidability/undecidability of sets. Rice's theorem and the concept of reducibility are important for proving undecidability.
4. be able to explain the terms Gödel numbering and halting problem and prove the undecidability of the latter.
5. be able to understand and sketch the proofs presented.

With Turing computability, we have captured algorithmic controllability of *functions*.

We are now also interested in the algorithmic controllability of *sets*. For this purpose, we will get to know the concept of *decidability*.

Here, an algorithm —when given x as input— computes the answer to the question “ $x \in A$ ”.

Decision algorithm for a set A :

$$\text{input } x \implies \text{output } \begin{cases} 1 \text{ (yes),} & \text{if } x \in A \\ 0 \text{ (no),} & \text{if } x \notin A \end{cases}$$

This is the computation of the characteristic function of A .

Definition 2.68 (characteristic function)

Let \mathbb{G} be a base set such as \mathbb{N} , \mathbb{Z} or Σ^* . For $A \subseteq \mathbb{G}$, the **characteristic function** $c_A: \mathbb{G} \rightarrow \{0, 1\}$ is defined as follows:

$$c_A(x) \stackrel{df}{=} \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

Definition 2.69 (decidability)

Let $n \geq 0$ and $t: \mathbb{N} \rightarrow \mathbb{N}$ a total function.

- ▶ $A \subseteq \mathbb{N}^n$ is **decidable** $\stackrel{df}{\iff} c_A$ is computable.
- ▶ **REC** $\stackrel{df}{=} \{A \mid \exists n \geq 0 \text{ with } A \subseteq \mathbb{N}^n \text{ and } A \text{ is decidable}\}$
(recursive sets)
- ▶ An algorithm **M decides** $A \subseteq \mathbb{N}^n$ **in time t**
(**resp. O(t)**) $\stackrel{df}{\iff} M$ computes c_A in time t (resp. $O(t)$).

Remark 2.70

As mentioned before, we now investigate the algorithmic controllability of sets.

An alternative point of view would be: We now investigate the computability of certain specific functions: namely those whose range is a subset of $\{0, 1\}$. It will turn out that it suffices to consider such specific functions in order to see that many “interesting” functions are not computable.

Example 2.71

The following subsets of \mathbb{N} are decidable.

- ▶ the set of square numbers: $\{n^2 \mid n \geq 0\}$
- ▶ the set of powers of two: $\{2^n \mid n \geq 0\}$
- ▶ the set of all prime numbers: \mathbb{P}
- ▶ the set $G = \{n \mid n \geq 2 \text{ and there exist prime numbers } p \text{ and } q \text{ with } 2n = p + q\}$ ¹⁰
- ▶ the set $F = \{n \mid n > 2 \text{ and } \forall a, b, c \in \mathbb{N}^+ [a^n + b^n \neq c^n]\}$ ¹¹
- ▶ the set $H = \{n \leq 49 \mid n \text{ will be drawn in the next lottery}\}$ ¹²

¹⁰Goldbach's conjecture says $G = \mathbb{N} \setminus \{0, 1\}$. Although G is decidable, we do not know whether Goldbach's conjecture holds.

¹¹In 1995, Andrew Wiles showed $F = \mathbb{N} \setminus \{0, 1, 2\}$, i.e., Fermat's last theorem. Only this theorem makes it clear that F is decidable.

¹²This set is decidable because there *exists an algorithm computing its characteristic function*. Knowing the algorithm is another matter. This would make you rich.


Theorem 2.72

1. Every finite subset of \mathbb{N}^n is decidable.
2. $A \subseteq \mathbb{N}^n$ decidable $\Rightarrow \bar{A} = (\mathbb{N}^n \setminus A)$ decidable.
3. $A, B \subseteq \mathbb{N}^n$ decidable $\Rightarrow A \cup B$ and $A \cap B$ decidable.
4. If $A \subseteq \mathbb{N}^n$ is decidable and $f : \mathbb{N}^m \rightarrow \mathbb{N}^n$ is computable and total, then $f^{-1}(A)$ is decidable, where $f^{-1}(A) \stackrel{\text{df}}{=} \{x \in \mathbb{N}^m \mid f(x) \in A\}$.

Proof idea:

- The statements can be easily derived from the definitions

Proof.

$$(-3x+4) = 7u$$




Example 2.73 (Collatz problem)

($3n+1$ conjecture, Syracuse algorithm, Ulam numbers)

The question of whether the following $U \subseteq \mathbb{N}$ is a decidable set is unknown.

$x \in U \stackrel{df}{\iff}$ after repeated application of the rule
 "If x is even, divide by 2, otherwise form $3x + 1$."
 finally 1 is generated.

It is conjectured that $U = \mathbb{N} \setminus \{0\}$. This has not yet been proven or disproven. The conjecture holds if and only if the While program f_2 in Example 2.24 stops on all inputs.

e.g. $27 \rightarrow 82 \rightarrow 41 \rightarrow 124 \rightarrow 62 \rightarrow 31 \rightarrow 94 \rightarrow 47 \rightarrow 142 \rightarrow 71 \rightarrow 214 \rightarrow 107 \rightarrow 322 \rightarrow 161 \rightarrow 484 \rightarrow 242 \rightarrow 121 \rightarrow 364 \rightarrow 182 \rightarrow 91 \rightarrow 274 \rightarrow 137 \rightarrow 412 \rightarrow 206 \rightarrow 103 \rightarrow 310 \rightarrow 155 \rightarrow 466 \rightarrow 233 \rightarrow 700 \rightarrow 350 \rightarrow 175 \rightarrow 526 \rightarrow 263 \rightarrow 790 \rightarrow 395 \rightarrow 1186 \rightarrow 593 \rightarrow 1780 \rightarrow 890 \rightarrow 445 \rightarrow 1336 \rightarrow 668 \rightarrow 334 \rightarrow 167 \rightarrow 502 \rightarrow 251 \rightarrow 754 \rightarrow 377 \rightarrow 1132 \rightarrow 566 \rightarrow 283 \rightarrow 850 \rightarrow 425 \rightarrow 1276 \rightarrow 638 \rightarrow 319 \rightarrow 958 \rightarrow 479 \rightarrow 1438 \rightarrow 719 \rightarrow 2158 \rightarrow 1079 \rightarrow 3238 \rightarrow 1619 \rightarrow 4858 \rightarrow 2429 \rightarrow 7288 \rightarrow 3644 \rightarrow 1822 \rightarrow 911 \rightarrow 2734 \rightarrow 1367 \rightarrow 4102 \rightarrow 2051 \rightarrow 6154 \rightarrow 3077 \rightarrow 9232 \rightarrow 4616 \rightarrow 2308 \rightarrow 1154 \rightarrow 577 \rightarrow 1732 \rightarrow 866 \rightarrow 433 \rightarrow 1300 \rightarrow 650 \rightarrow 325 \rightarrow 976 \rightarrow 488 \rightarrow 244 \rightarrow 122 \rightarrow 61 \rightarrow 184 \rightarrow 92 \rightarrow 46 \rightarrow 23 \rightarrow 70 \rightarrow 35 \rightarrow 106 \rightarrow 53 \rightarrow 160 \rightarrow 80 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

Reducibility

We get to know a concept that translates one decision problem into another.

Decidability is translated in a certain way.

This property can be used to prove sets to be (un)decidable.

Definition 2.74 (reducibility)

Let $A \subseteq \mathbb{N}^m$ and $B \subseteq \mathbb{N}^n$. **A is reducible to B** $\stackrel{df}{\iff}$ there is a total, computable $f : \mathbb{N}^m \rightarrow \mathbb{N}^n$, such that for all $x \in \mathbb{N}^m$:

$$x \in A \iff f(x) \in B.$$

The equivalence can be equivalently expressed by $c_A = c_B \circ f$.

Property 2.75

Let $A \subseteq \mathbb{N}^m$ and $B \subseteq \mathbb{N}^n$. If A reducible to B , the following implication holds.

$$B \in \text{REC} \Rightarrow A \in \text{REC}$$

Proof.

Let A be reducible to B , i.e., there exists total, computable f with $c_A = c_B \circ f$.

If $B \in \text{REC}$, then c_B computable. By Lemma 2.57, $c_B \circ f = c_A$ computable, thus $A \in \text{REC}$. □

Property 2.75 can be applied in two ways:

1. To prove that a set A lies in REC.
(choose B from REC; show A reducible to B)
2. To prove that a set B is *not* in REC.
(choose A outside REC; show A reducible to B ; by Lemma 2.75, B is outside REC)

The following example uses the property to prove the decidability of a set.

Example 2.76 (Proof of decidability)

The set of Mersenne primes is defined as

$$A = \{x \mid x \in \mathbb{P} \text{ and } x = 2^n - 1 \text{ for an } n \in \mathbb{N}\}.$$

The following function is total and computable.

$$f(x) = \begin{cases} x & \text{if } x + 1 \text{ is a power of two} \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, for all $x \in \mathbb{N}$: $x \in A \Leftrightarrow f(x) \in \mathbb{P}$.

This shows: A is reducible to \mathbb{P} .

We already know $\mathbb{P} \in \text{REC}$.

By Lemma 2.75, $A \in \text{REC}$.

To use Property 2.75 to show the undecidability of sets B you need a set A , whose undecidability is already known.

We now prove the undecidability of a first set.

The Halting Problem

We are looking for a set that is not decidable.

To do this, we encode all algorithms (RAMs in this case) using natural numbers. This procedure goes back to Kurt Gödel and is called *Gödel numbering*.

Gödel Numbering of RAMs

1. Encoding of individual RAM instructions:

Instruction b	Encoding $\langle b \rangle$
$Ri \leftarrow Rj$	$\langle 0, i, j, 0 \rangle$
$Ri \leftarrow RRj$	$\langle 1, i, j, 0 \rangle$
$RRi \leftarrow Rj$	$\langle 2, i, j, 0 \rangle$
$Ri \leftarrow j$	$\langle 3, i, j, 0 \rangle$
$Ri \leftarrow Rj + Rk$	$\langle 4, i, j, k \rangle$
$Ri \leftarrow Rj - Rk$	$\langle 5, i, j, k \rangle$
GOTO i	$\langle 6, i, 0, 0 \rangle$
IF $Ri = 0$ GOTO j	$\langle 7, i, j, 0 \rangle$
IF $Ri > 0$ GOTO j	$\langle 8, i, j, 0 \rangle$
STOP	$\langle 9, 0, 0, 0 \rangle$

2. Encoding of RAM M with instructions b_1, \dots, b_s :

$$\langle M \rangle \stackrel{df}{=} \langle \langle b_1 \rangle, \dots, \langle b_s \rangle \rangle$$

3. For $i \in \mathbb{N}$ let $M_i \stackrel{df}{=} \begin{cases} M, & \text{if } i = \langle M \rangle \text{ for a RAM } M \\ M^*, & \text{if } i \text{ is not code of a RAM,} \end{cases}$ where M^* is the RAM with the single instruction STOP.

Observation 2.77

1. Every RAM is contained in the set $\{M_0, M_1, \dots\}$.
2. There is an algorithm that for a given i computes and simulates the RAM M_i .
3. There is an algorithm that computes an i with $M = M_i$ for a given RAM M .

An enumeration with these properties is called **Gödel numbering**. There are also Gödel numberings for While programs and TMs.

Definition 2.78 (Halting problem)

$K_0 \stackrel{df}{=} \{x \mid M_x \text{ halts on input } x\}$ **Special halting problem**

$K \stackrel{df}{=} \{(x, y) \mid M_x \text{ halts on input } y\}$ **General halting problem**


Theorem 2.79

K_0 is not decidable.

Proof idea:

- ▶ Assume K_0 decidable; construct a machine M so that $M(x)$ behaves differently from $M_x(x)$; since $M = M_i$ for an i , M must also behave differently from itself; contradiction

Proof.

$$(-3x+4)=7u$$




Corollary 2.80

K is not decidable.

Proof.

Exercise. □

Much more general, even the question of whether the function computed by a given source code has a property S cannot be solved algorithmically.

Rice's Theorem

Theorem 2.81 (Rice's Theorem)

Let $n \geq 0$. If $S \neq \emptyset$ is a proper subset of the set of all computable functions $\mathbb{N}^n \rightarrow \mathbb{N}$, then the following set is undecidable.

$$I(S) \stackrel{\text{df}}{=} \{i \in \mathbb{N} \mid \text{the function } \mathbb{N}^n \rightarrow \mathbb{N} \text{ computed by } M_i \text{ is in } S\}$$

Proof idea:

- ▶ $h(x) \stackrel{\text{df}}{=} n.d.$; assume w.l.o.g. $h \notin S$
- ▶ Choose computable $g \in S$
- ▶ Construct RAM $M[a]$ such that $(a \in K_0 \Rightarrow M[a] \text{ computes } g)$ and $(a \notin K_0 \Rightarrow M[a] \text{ computes } h)$
- ▶ K_0 is reducible to $I(S)$ by $f(x) = \langle M[x] \rangle$; thus $I(S) \notin \text{REC}$

Proof.

Let $h : \mathbb{N}^n \rightarrow \mathbb{N}$ with $h(x_1, \dots, x_n) \stackrel{\text{df}}{=} n.d.$ We assume $h \notin S$, the other case is treated analogously.

Choose comp. $g : \mathbb{N}^n \rightarrow \mathbb{N}$ with $g \in S$ (exists by assumption).

For $a \in \mathbb{N}$ let $M[a]$ be the RAM that on input y_1, \dots, y_n

1. simulates M_a on a until the computation stops
2. computes $g(y_1, \dots, y_n)$ and returns this value

The function $f : \mathbb{N} \rightarrow \mathbb{N}$ with $f(x) = \langle M[x] \rangle$ is total and computable (determine the RAM $M[x]$ and then its encoding).

If $x \in K_0$, then M_x halts on x , thus the machine computes $M[x] = M_{\langle M[x] \rangle}$ the function g , hence $f(x) = \langle M[x] \rangle \in I(S)$.

If $x \notin K_0$, then M_x does not halt on x , so the machine $M[x] = M_{\langle M[x] \rangle}$ computes the function h , thus $f(x) = \langle M[x] \rangle \notin I(S)$.

Hence $x \in K_0 \Leftrightarrow f(x) \in I(S)$, i.e. K_0 is reducible to $I(S)$. Due to $K_0 \notin \text{REC}$ and Lemma 2.75 it holds $I(S) \notin \text{REC}$. □

Example 2.82 (Undecidable sets)


The undecidability of the following sets results from Rice's theorem.

$$A = \{i \mid M_i \text{ computes } \varphi\} \quad \text{for each comp. } \varphi : \mathbb{N}^n \rightarrow \mathbb{N}$$

$$B = \{i \mid \exists x \in \mathbb{N} \text{ such that } M_i \text{ on input } x \text{ returns } 17\}$$

$$C = \{i \mid M_i \text{ halts on input } 17\}$$

$$D = \{(i, j) \mid M_i \text{ and } M_j \text{ compute the same function } \mathbb{N}^n \rightarrow \mathbb{N}\}$$

$$(-3x+4) = 7u$$


The term *decidable* was initially only defined for subsets of \mathbb{N}^n . We now also want to use the term for other selected base sets \mathbb{G} such as $\mathbb{G} = \Sigma^*$ or $\mathbb{G} = \mathbb{Z}$.

Definition 2.83 (Extension of Definition 2.69)

Let \mathbb{G} be a base set, $A \subseteq \mathbb{G}$ and $t : \mathbb{N} \rightarrow \mathbb{N}$ a total function.

- ▶ A is called **decidable** $\stackrel{df}{\iff} c_A : \mathbb{G} \rightarrow \{0, 1\}$ is computable.
- ▶ An algorithm **M decides A in time t (resp. $O(t)$)** $\stackrel{df}{\iff} M$ computes $c_A : \mathbb{G} \rightarrow \{0, 1\}$ in the time t (or $O(t)$).

Example 2.84 (An undecidable set of polynomials)

$$D = \{p \mid p \text{ is polynomial in several variables with coefficients from } \mathbb{Z} \text{ and there exist } a_1, \dots, a_n \in \mathbb{Z} \text{ with } p(a_1, \dots, a_n) = 0\}$$

The question for the decidability of D is known as *Hilbert's 10th problem*. In 1970, Yuri Matiyasevich proved its undecidability.