Theoretical Computer Science

Lecture Notes

Prof. Dr. Titus Dose

OTH Regensburg – Faculty of Computer Science and Mathematics

January 15, 2025

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

3. Finite Automata
3.1 Deterministic Finite Automata
3.2 Non-Deterministic Finite Automata
3.3 Regular Expressions and Regular Sets
3.4 The Pumping Lemma for Regular Languages
3.5 Equivalence and Minimization of Automata

Table of Contents IV

4. Formal Languages
4.1 The Chomsky Hierarchy
4.2 Languages of Type 3
4.3 Context-Free Languages
4.4 Non-Determinism
4.5 Context-Sensitive Languages
4.6 Languages of Type 0
4.7 Summary

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]

[1]This 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

$$\begin{aligned} x = 2 &\Leftrightarrow x + 1 = 3 \\ &\Leftrightarrow (x+1)^2 = 9 \text{ WRONG!} \\ &\Leftrightarrow 2^{(x+1)^2} = 512 \end{aligned}$$

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 inferences[2] that B is true.

[2] It 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. $\square$

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 $\neg A$. □

[3] Write down the truth value table if you are not convinced. Cf. page 11.

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 no 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. □

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] $\sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n$

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

$$\sum_{i=1}^{k+1} i = k+1+\underbrace{\sum_{i=1}^{k} i}_{\overset{A_k}{=}\frac{k^2+k}{2}} = k+1+\frac{k^2+k}{2} = \frac{2(k+1)+k^2+k}{2}$$

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

holds[5]. Thus A_{k+1} holds, IS is proven, and the proof is complete. $\square$

[5]Note: 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 $s_1, s_2, \ldots$ be natural numbers. We define:

BC: $\sum_{i=1}^{0} s_i := 0$.
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 $\sum_{i=1}^{5} i$?

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

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.

[6]Note: 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 { }.

$$\begin{aligned}
\mathbb{N} &\overset{df}{=} \{0,1,\ldots\} = \text{Set of natural numbers}\\
\mathbb{N}^+ &\overset{df}{=} \{1,2,\ldots\} = \text{Set of positive natural numbers}\\
\mathbb{P} &\overset{df}{=} \{2,3,5,7,11,13,17,\ldots\} = \text{Set of primes}\\
\mathbb{Z} &\overset{df}{=} \{\ldots,-2,-1,0,1,2,\ldots\} = \text{Set of integers}\\
\emptyset &\overset{df}{=} \text{empty set}
\end{aligned}$$

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 $zy \le x$.

Sets

Notation for the definition of sets:

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

Element relationship and inclusion:

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

Set Operations

$A \cap B$	$\overset{df}{=}$	$\{a \mid a \in A \text{ and } a \in B\}$	(Intersection of A and B)
$A \cup B$	$\overset{df}{=}$	$\{a \mid a \in A \text{ or } a \in B\}$	(Union of A and B)
$A \setminus B$	$\overset{df}{=}$	$\{a \mid a \in A \text{ and } a \notin B\}$	(Difference of A and B)
$A - B$	$\overset{df}{=}$	$A \setminus B$	(Difference of A and B)
$\overline{A}$	$\overset{df}{=}$	$M \setminus A$	(Complement of A relative to a fixed base set M)
$\mathcal{P}(A)$	$\overset{df}{=}$	$\{B \mid B \subseteq A\}$	(Power set of A)

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

Tuple (Vector) and Cartesian Product

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

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

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 $f : A \to B$) 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 $G_g = G_h$.

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

Definition 1.1

Let $f\colon A \to B$ and $g\colon B \to C$ be functions.

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

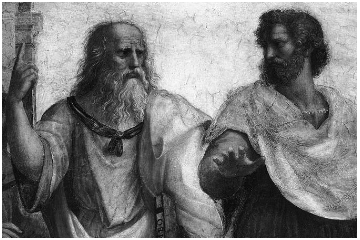
Later, the term *algorithm* was generally used to describe a procedure for solving a problem.

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:

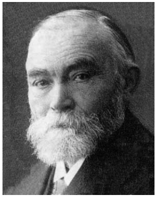
- Even in ancient times, there were discussions about whether there are fundamental limitations to knowledge and predictability. The liar's paradox has been known since the 4th century BC: "This statement is false". The basic ideas behind today's concept of axiomatization can already be found in Plato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

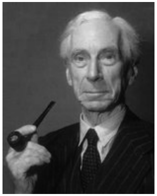
- 1893: Gottlob Frege (1848–1925) writes his "Grundgesetze der Arithmetik" (Foundations of arithmetic), which are intended to give all of mathematics a purely logical foundation (logicism). All theorems of arithmetic should be traceable to logical truths.



- 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 Russel (1872–1970) points out the following paradox to Frege in a letter (*Russel's antinomy*). For $R \overset{df}{=} \{x \mid x \notin x\}$ 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.

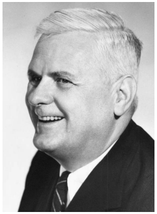
- 1928: Hilbert formulates the decision problem: Is there an algorithm that, given a mathematical statement, determines whether it follows from a given set of axioms?

- 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 runtime 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 $\boldsymbol{\Sigma}$** 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 $a_1, a_2, \ldots, a_n \in \Sigma$.

- The **length of a word** $w = a_1 a_2 \cdots a_n$ is $|w| \overset{df}{=} n$.
- $\Sigma^n \overset{df}{=} \{w \mid w \text{ is a word over } \Sigma \text{ with } |w| = n\}$
  It holds $\Sigma^0 = \{\varepsilon\}$.
- $\Sigma^* \overset{df}{=} \{w \mid w \text{ is a word over } \Sigma\} = \bigcup_{n \geq 0} \Sigma^n$ 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 $\boldsymbol{v \cdot w} \overset{df}{=} b_1 b_2 \cdots b_m c_1 c_2 \cdots c_n$ (the $\cdot$ may be omitted).
- 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 $\Sigma^* \to \Sigma^*$ 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 $\Sigma^* \to \Sigma^*$, $\mathbb{N} \to \mathbb{N}$, or $\mathbb{Z} \to \mathbb{Z}$.
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 $a_0, \ldots, a_{m-1} \in \{0, 1\}$.

Proof.

Exercise. $\square$

Definition 2.4 (Binary representation)

- The **binary representation** of 0 is 0.
- Let $n \geq 1$ and let $m \in \mathbb{N}$ and $a_0, \ldots, a_m \in \{0,1\}$ 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**: $\mathbb{N} \to \{0,1\}^*$ so that $\mathrm{bin}(n)$ 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)

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

The next property shows how to compute binary representations.

Property 2.6

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

Proof.

Exercise. $\square$

Example 2.7 (Computing the binary representation)

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

$$\begin{array}{rcl} 54 & = & 2 \cdot 27 + \mathbf{0} \\ 27 & = & 2 \cdot 13 + \mathbf{1} \\ 13 & = & 2 \cdot 6 + \mathbf{1} \\ 6 & = & 2 \cdot 3 + \mathbf{0} \\ 3 & = & 2 \cdot 1 + \mathbf{1} \\ 1 & = & 2 \cdot 0 + \mathbf{1} \end{array}$$

Encoding between Words and Natural Numbers

Recall that by Property 2.6, the binary representation of natural numbers is an injective mapping bin : $\mathbb{N} \to \{0,1\}^*$. 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 $a_0, \ldots, a_m \in \{1, 2\}$.

Proof.

Exercise. □

Definition 2.9 (Dyadic representation)

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

$$\begin{aligned} \text{dya}(0) &\overset{df}{=} \varepsilon \\ \text{dya}(n) &\overset{df}{=} a_m \cdots a_0, \quad \text{if } n \geq 1,\ n = \textstyle\sum_{i=0}^{m} a_i \cdot 2^i \text{ and} \\ & \qquad\qquad\qquad a_0, \ldots, a_m \in \{1, 2\}. \end{aligned}$$

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

Property 2.10

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

Proof.

Follows from Definition 2.9. $\square$

The dyadic representation thus provides a bijection $\mathbb{N} \to \Sigma^*$ 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 $\mathbb{N} \to \Sigma^*$ 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 $a_0, \ldots, a_m \in \{1, \ldots, k\}$.

Definition 2.12 (k-adic representation)

For $k \geq 2$ is $\mathrm{ad}_k : \mathbb{N} \to \{1, \ldots, k\}^*$ defined by

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

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

Property 2.13

For $k \geq 2$ the function $ad_k : \mathbb{N} \to \{1, \ldots, k\}^*$ is bijective.

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

Property 2.14

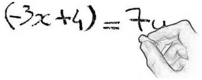
Let $k \geq 2$.

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* $a \in \{1, \ldots, k\}$
4. $ad_k{}^{-1}(xa) = k \cdot ad_k{}^{-1}(x) + a$ *for* $x \in \{1, \ldots, k\}^*$, $a \in \{1, \ldots, k\}$

Proof.

Exercise. □

Example 2.15 (Determine the k-adic representation)



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

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 $\mathsf{code}_{\Sigma^*} : \Sigma^* \to \mathbb{N}$ to encode words $w \in \Sigma^*$ by natural numbers.

$$\mathbf{code}_{\boldsymbol{\Sigma^*}}(w) \overset{df}{=} \mathsf{ad}_k{}^{-1}(w)$$

Encodings between Naturals and Integers

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

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

We denote the inverse function of $\mathrm{code}_{\mathbb{Z}}$ by $\mathrm{code}_{\mathbb{Z}}^{-1}$.
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 $\mathbb{N} \to \mathbb{N}$ (or alternatively, $\mathbb{Z} \to \mathbb{Z}$ or $\Sigma^* \to \Sigma^*$ 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 $x_1, \ldots, x_n$ be natural numbers with the following binary representations.

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

The encoding of the list $(x_1, \ldots, x_n)$ 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.[7] The encoding of the list is thus the natural number with binary representation

$$\begin{array}{c} 1\,0\ a_{1,1}\,a_{1,1}\,a_{1,2}\,a_{1,2}\cdots a_{1,r_1}\,a_{1,r_1}\ 1\,0 \\ a_{2,1}\,a_{2,1}\,a_{2,2}\,a_{2,2}\cdots a_{2,r_2}\,a_{2,r_2}\ 1\,0 \\ \vdots \\ a_{n,1}\,a_{n,1}\,a_{n,2}\,a_{n,2}\cdots a_{n,r_n}\,a_{n,r_n}\ 1\,0. \end{array}$$

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

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

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

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

[7] In case of the empty list the start marker is the end marker. So the code for the empty list is $\text{bin}^{-1}(10)$ and not $\text{bin}^{-1}(1010)$.

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

$$\mathsf{code}_{\mathcal{X}} : \mathcal{X} \to \mathbb{N}$$

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

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

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

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

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, AIMMS, 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, AutoIt, AutoLISP / Visual LISP, Averest, AWK, Axum, B, Babbage, Ballerina, Bash, BASIC, Batch file, bc, BCPL, BeanShell, BETA, BLISS, Blockly, 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, CobolScript – COBOL Scripting language, Cobra, CoffeeScript, ColdFusion, COMAL, COMIT, Common Intermediate Language, Common Lisp, COMPASS, Component Pascal, COMTRAN, 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, DRAKON, 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, Fjölnir, 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, Lithe, 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, Modelica, 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, o:XML, Oak, Oberon, OBJ2, Object Lisp, ObjectLOGO, Object REXX, Object Pascal, Objective-C, Obliq, OCaml, occam, occam-π, Octave, OmniMark, Opa, Opal, Open Programming Language, OpenCL, OpenEdge Advanced Business Language, OpenVera, OpenQASM, OPS5, OptimJ, Orc, ORCA/Modula-2, Oriel, 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, SequenceL, 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-compiler, Tom, Toi, Topspeed, TPU, Trac, TTM, T-SQL, Transcript, TTCN, Turing, TUTOR, TXL, TypeScript, Tynker, Ubercode, UCSD Pascal, Umple, Unicon, 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, XPL0, 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).

$$\neg a \overset{df}{=} \begin{cases} \text{true,} & \text{if } a = \text{false} \\ \text{false} & \text{else} \end{cases}$$

$$a \wedge b \overset{df}{=} \begin{cases} \text{true,} & \text{if } a = b = \text{true} \\ \text{false} & \text{else} \end{cases}$$

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

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,...,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 $b_1, \ldots, b_n$ are expressions, then f`(`b_1`,` $\ldots$`,` 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 `<=` b`)`, `(`a `>` b`)`, `(`a `>=` 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 `(not` a`)`, `(`a `and` b`)` and `(`a `or` b`)`.

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

Examples: `(0 == 1)`, `((5 - a) < f(3))`, `(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 $s_1, \ldots, s_n$ are instruction blocks, then the following is an instruction block.

  s_1
  s_2
  $\vdots$
  s_n

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

  $$\boxed{\text{The instruction blocks are executed in the order } s_1, \ldots, s_n. \text{ If one of the } s_i \text{ is not defined, then the computation of the complete instruction block is not defined.}}$$

  Example:

  ```
  a = (a - 1000000000000000000000000000000)
  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:                        if b:
  s1                           s1
else:
  s2
```

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

`if b:`	b true $\Rightarrow$ s_1 is executed.
`  s1`	b false $\Rightarrow$ s_2 is executed.
`else:`	b not defined $\Rightarrow$ coputation not defined
`  s2`	
`if b:`	b true $\Rightarrow$ s_1 is executed.
`  s1`	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 != 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, $n \geq 0$, $a_1, \ldots, a_n$ pairwise distinct variables, s an instruction block and c an expression. Then the following two constructs are declarations of the function f.

```
def f(a1,...,an):                def f(a1,...,an):
  s                                return c
  return c
```

Semantics of the function call $f(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are expressions: If one of the expressions $c_1, \ldots, c_n$ is not defined, the computation of the function call is not defined. Otherwise, s (if available) is executed, whereby the variables $a_1, \ldots, a_n$ are assigned the values $c_1, \ldots, c_n$ and the values of the remaining variables are not defined.[8] If this computation is defined, the value of the function call $f(c_1, \ldots, c_n)$ is is defined as the value of c (at the end of the computation). Otherwise, the computation of the function call is not defined.

[8] Local variable visibility

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

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

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

The above lists `[0,...,0]` contain m zeros.

The semantics correspond to the following function declarations:

```
def f(a_1,...,a_n):
  b_1 = 0
    ⋮
  b_m = 0
  s
  return c
```

```
def f(a_1,...,a_n):
  b_1 = 0
    ⋮
  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 `def` f`(...):` and `def` g`(...):` are two function declarations, the identifiers f and g are distinct.
3. If `def` f`(...):` is a function declaration and v a variable of the program, then the identifiers f and v are distinct.

If

```
    def f(a_1,...,a_n):
      ...
```

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

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

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 $\varphi : \mathbb{Z}^n \to \mathbb{Z}$ 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 $\varphi : \mathbb{N}^n \to \mathbb{N}$ 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** $\overset{df}{=} \{\varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $g : \mathbb{Z} \to \mathbb{Z}$ 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: $g(k+1) = 2 \cdot g(k) \overset{\text{ind. hypothesis}}{=} 2 \cdot 2^k = 2^{k+1}$. □

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 $\varphi : \mathbb{Z}^n \to \mathbb{Z}$ with $n \geq 0$ is called **Loop-computable** if there is a Loop program P such that $D_{\psi_P} = D_\varphi$ and $\psi_P(x_1, \ldots, x_n) = \varphi(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_\varphi$.
- A function $\varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $\psi_P(x_1, \ldots, x_n) = \varphi(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_\varphi$.
- **LOOP** $\overset{df}{=} \{\varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $a_1, \ldots, a_n$ are expressions with $n \geq 1$, then `print`$(a_1, \ldots, a_n)$ 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.	
`[i,z] = [0,0]`	-3	12	0	0	
`if (x < 0):`	-3	12	0	0	true
`x = (0 - x)`	3	12	0	0	
`y = (0 - y)`	3	-12	0	0	
`for i in range(0,x):`	3	-12	0	0	$i = 0$
`z = (z + y)`	3	-12	0	-12	
`for i in range(0,x):`	3	-12	1	-12	$i = 1$
`z = (z + y)`	3	-12	1	-24	
`for i in range(0,x):`	3	-12	2	-24	$i = 2$
`z = (z + y)`	3	-12	2	-36	
`for i in range(0,x):`	3	-12	2	-36	end of loop
`return z`	3	-12	2	-36	return -36

Example 2.21

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

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

where for $y \neq 0$

$$\lfloor x/y \rfloor \overset{df}{=} \begin{cases} \text{largest } z \in \mathbb{Z} \text{ with } y \cdot z \leq x, & \text{if } y > 0 \\ \lfloor (-x)/(-y) \rfloor, & \text{if } y < 0 \end{cases}$$ [9]

[9]Generally, 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.

$$\begin{aligned}
\mathrm{modZ}(x, y) &\overset{df}{=} x - (y \cdot \mathrm{d}ivZ(x, y)) \\
\mathrm{divisors}(x) &\overset{df}{=} \begin{cases} \text{number of divisors of } x, & \text{if } x \geq 1 \\ 0, & \text{else} \end{cases} \\
\mathrm{prime}(n) &\overset{df}{=} \begin{cases} n\text{-th prime number}, & \text{if } n \geq 0 \\ 2, & \text{else} \end{cases}
\end{aligned}$$

For $x \in \mathbb{N}$ and $y \in \mathbb{N}^+$, $\mathrm{m}odZ(x, y)$ 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
```

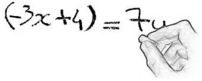
$\Rightarrow$ 3 different functions are computed

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
```



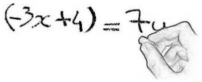
Example 2.25

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
```



Theorem 2.26 (Loop vs While computability)

1. *Every Loop-computable function* $\varphi : \mathbb{Z}^n \to \mathbb{Z}$ *with* $n \geq 0$ *is While-computable.*
2. *Every Loop-computable function* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *with* $n \geq 0$ *is While-computable.*
3. LOOP $\subsetneq$ 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 LOOP $\subseteq$ 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 $\mathbb{N}^n \to \mathbb{N}$ are total.

Consequently, $g : \mathbb{N} \to \mathbb{N}$ with $g(x) \overset{df}{=}$ 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 $\text{LOOP} \neq \text{WHILE}$. $\square$

Ackermann Function

Theorem 2.26 raises the following question:

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

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

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

Theorem 2.27

The Ackermann function is a total function in $\mathrm{WHILE} - \mathrm{LOOP}$*.*

Without proof.

The Ackermann function grows extremely quickly:

$$\begin{aligned} a(1,3) &= 5 \\ a(2,3) &= 9 \\ a(3,3) &= 61 \\ a(4,3) &\approx 10^{10^{19728}} \end{aligned}$$

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*.

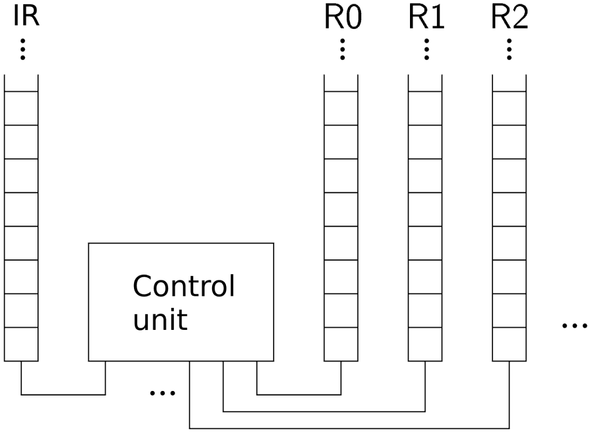
Reason: real computers are very different in terms of

- Processor
- Architecture
- Computation 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 $0, 1, 2, \ldots$) .
- 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

$[\mathrm{R}i]$ = Contents of register Ri
$[\mathrm{IR}]$ = Contents of register IR

Definition 2.28 (modified difference)

The function $\mathrm{md}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined as follows.

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

The following instructions can be used in RAM programs:

Instructions (syntax)	**Effect** (semantics)
Transport instructions	
R$i \leftarrow$ Rj (direct address)	$[\mathrm{R}i] := [\mathrm{R}j]$, $[\mathrm{IR}] = [\mathrm{IR}] + 1$
R$i \leftarrow$ RRj (indir. Address.)	$[\mathrm{R}i] := [\mathrm{R}[\mathrm{R}j]]$, $[\mathrm{IR}] = [\mathrm{IR}] + 1$
RR$i \leftarrow$ Rj (indir. address.)	$[\mathrm{R}[\mathrm{R}i]] := [\mathrm{R}j]$, $[\mathrm{IR}] := [\mathrm{IR}] + 1$
Arithmetic instructions	
R$i \leftarrow k$	$[\mathrm{R}i] := k$, $[\mathrm{IR}] = [\mathrm{IR}] + 1$
R$i \leftarrow$ R$j +$ Rk	$[\mathrm{R}i] := [\mathrm{R}j] + [\mathrm{R}k]$, $[\mathrm{IR}] = [\mathrm{IR}] + 1$
R$i \leftarrow$ R$j -$ Rk	$[\mathrm{R}i] := [\mathrm{R}j] \dot{-} [\mathrm{R}k]$, $[\mathrm{IR}] = [\mathrm{IR}] + 1$

Instructions (syntax)	**Effect** (semantics)
Jump instructions	
GOTO m	$[\mathrm{IR}] := m$
IF R$i = 0$ GOTO m	$[\mathrm{IR}] := \begin{cases} m, & \text{if } [\mathrm{R}i] = 0 \\ [\mathrm{IR}] + 1 & \text{else} \end{cases}$
IF R$i > 0$ GOTO m	$[\mathrm{IR}] := \begin{cases} m, & \text{if } [\mathrm{R}i] > 0 \\ [\mathrm{IR}] + 1 & \text{else} \end{cases}$
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:	$[R2] =$ intermediate total, $[R3] = 1$
Idea:	$x \cdot y = \underbrace{x + x + \cdots + x}_{y\text{-times}}$

Program:

```
0   R3 ← 1
1   IF R1 = 0 GOTO 5
2   R2 ← R2 + R0
3   R1 ← R1 − R3
4   GOTO 1
5   R0 ← 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 $[R10], [R11], [R12], \ldots, [R[R1]]$ to R0.

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

Idea: Consider Ri for
$i = [R1], [R1] - 1, \ldots, 12, 11, 10$;
the maximum of [Ri] 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	$R3 \leftarrow R1 - R2$	$(a \leq b \Leftrightarrow a \dot{-} b = 0)$
	3	IF $R3 = 0$ GOTO 11	
2	4	$R3 \leftarrow RR1$	
	5	$R3 \leftarrow R3 - R0$	$(a \leq b \Leftrightarrow a \dot{-} b = 0)$
	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 $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ with $n \geq 0$ **is computed by M** if for all $x_1, \dots, x_n \in \mathbb{N}$:

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

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

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* $\mathrm{prod} : \mathbb{N}^2 \to \mathbb{N}$ *with* $\mathrm{prod}(x, y) \overset{df}{=} x \cdot y$*,*
- *the 1-ary function* $C^1_0 : \mathbb{N} \to \mathbb{N}$ *with* $C^1_0(x) \overset{df}{=} 0$ *and*
- *the 3-ary function* $f : \mathbb{N}^3 \to \mathbb{N}$ *with* $f(x, y, z) \overset{df}{=} (x \cdot y) + z$*.*

Encoding of Lists of Integers in While Programs

We will later show RAM $=$ 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 $(y_1, \ldots, y_n)$ can be encoded by the following natural number:

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

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 $f : \mathbb{Z}^n \to \mathbb{Z}$ or $f : \mathbb{N}^n \to \mathbb{N}$ 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. $\square$

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** $\mathrm{fib} : \mathbb{Z} \to \mathbb{Z}$ is inductively defined by:

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

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.

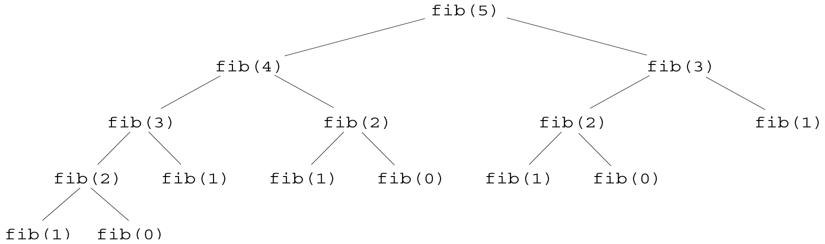
For $n \geq 0$ you can easily compute:

$\quad\quad$ 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:

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

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 $\varphi \in$ RAM and $\varphi : \mathbb{N}^n \to \mathbb{N}$.
Assume φ is computed by the following RAM M:

$$\begin{array}{ll} 0 & b_0 \\ 1 & b_1 \\ \vdots & \vdots \\ k & b_k \end{array}$$

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 Ri that has not yet been used is accessed, then i is appended to u and 0 to v (i.e. $[\mathrm{R}i] = 0$)

Example: $u = [5, 2]$ and $v = [24, 7]$ means that R5 and R2 have been used so far, $[\mathrm{R}5] = 24$ and $[\mathrm{R}2] = 7$.

We use the following functions "read" and "write" to access the Ri 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):           }
            $s_0$               }
        if (ir == 1):           }
            $s_1$               }  simulation of at
                 ⋮              }  least one cycle of $M$
        if (ir == k - 1):       }
            $s_{k-1}$           }
    return read(u,v,0)
```

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

If b_i is instruction ...	then s_i is instruction block ...

$\mathsf{R}a \leftarrow \mathsf{R}b$	`i = read(u,v,`b`)`
	`write(u,v,`a`,i)`
	`ir = ir + 1`

$\mathsf{R}a \leftarrow \mathsf{RR}b$	`i = read(u,v,`b`)`
	`j = read(u,v,i)`
	`write(u,v,`a`,j)`
	`ir = ir + 1`

$\mathsf{RR}a \leftarrow \mathsf{R}b$	`i = read(u,v,`b`)`
	`j = read(u,v,`a`)`
	`write(u,v,j,i)`
	`ir = ir + 1`

$\mathsf{R}a \leftarrow b$	`write(u,v,`a`,`b`)`
	`ir = ir + 1`

$\mathsf{R}a \leftarrow \mathsf{R}b + \mathsf{R}c$	`i = read(u,v,`b`) + read(u,v,`c`)`
	`write(u,v,`a`,i)`
	`ir = ir + 1`

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

Thus $D_{\psi_{\texttt{phi}}} \cap \mathbb{N}^n = D_\varphi$ and $\psi_{\texttt{phi}}(x_1, \ldots, x_n) = \varphi(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_\varphi$.
Then $\varphi \in$ 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 WHILE $\subseteq$ 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*

   - $s_1, \ldots, s_n$ are instr. blocks $\Rightarrow$ $\left.\begin{matrix} s_1 \\ \vdots \\ s_n \end{matrix}\right\}$ is instr. block
   - a is variable, s is instr. block $\Rightarrow$ $\left.\begin{matrix} \texttt{while(a} > \texttt{0)}: \\ \quad s \end{matrix}\right\}$ 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)

- $\varphi : \mathbb{Z}^n \to \mathbb{Z}$ 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 $\psi_P(x_1, \ldots, x_n) = \varphi(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_\varphi$.
- $\varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $\psi_P(x_1, \ldots, x_n) = \varphi(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_\varphi$.
- **MINIWHILE** $\overset{df}{=} \{\varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $\langle y_1, \ldots, y_n \rangle$. □

Theorem 2.38

WHILE $\subseteq$ RAM.

Proof sketch.

- Let $g \in$ WHILE $\subseteq$ MINIWHILE with $g : \mathbb{N}^n \to \mathbb{N}$.
- There is a Mini-While program P with $D_{\psi_P} \cap \mathbb{N}^n = D_g$ and $\psi_P(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in D_g$.
- 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.

$\square$

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 $\varphi : \mathbb{N}^n \to \mathbb{N}$ 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

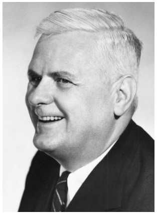
1. understand the structure and working mechanism of TMs.
2. understand and be able to apply the concept of Turing computability in detail.
3. be able to create simple TM programs yourselves and be able to determine the function computed by simple TM programs.
4. understand the formal definition of a 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.

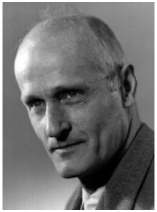
- 1931–1934: Alonzo Church (1903–1995) develops the lambda calculus and shows that all common number-theoretic functions are lambda-definable.



- 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.



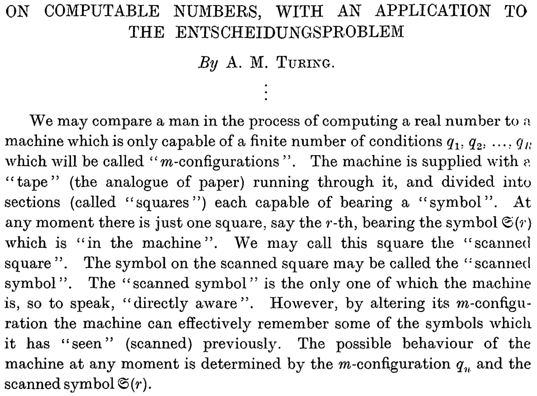
- Despite these indications, Gödel continues to reject the conjecture. He is not sure whether the general recursive functions cover all possibilities of recursion. Gödel expects a definition that *obviously* covers the computable.

- 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.



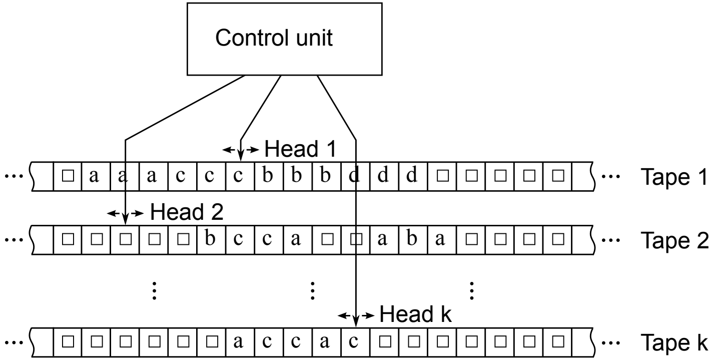
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

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 $q_1, q_2, \ldots, q_R$ 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
Head 1
Head 2
Head k
Tape 1
Tape 2
Tape k
□ a a a c c c b b b d d d □ □ □ □ □
□ □ □ □ □ b c c a □ □ a b a □ □ □ □
□ □ □ □ □ □ a c c a c □ □ □ □ □ □ □

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 cell 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.

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

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

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

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

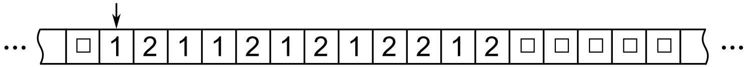
$$sa_1 \cdots a_k \to$$

the machine behaves according to the imaginary instruction

$$sa_1 \cdots a_k \to s_1 a_1 \cdots a_k \mathrm{O} \cdots \mathrm{O}.$$

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

Example 2.43 (TM – incrementing dyadic numbers)



$s_0\ 1 \ \rightarrow\ s_0\ 1\ \mathrm{R}$ // run to the right

$s_0\ 2 \ \rightarrow\ s_0\ 2\ \mathrm{R}$

$s_0\ \Box \ \rightarrow\ s_2\ \Box\ \mathrm{L}$ // $s_2 =$ carryover 1

$s_2\ 1 \ \rightarrow\ s_1\ 2\ \mathrm{O}$

$s_2\ 2 \ \rightarrow\ s_2\ 1\ \mathrm{L}$

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

A word $a_1 \cdots a_n$ with $a_1, \ldots, a_n \in \Sigma$ 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

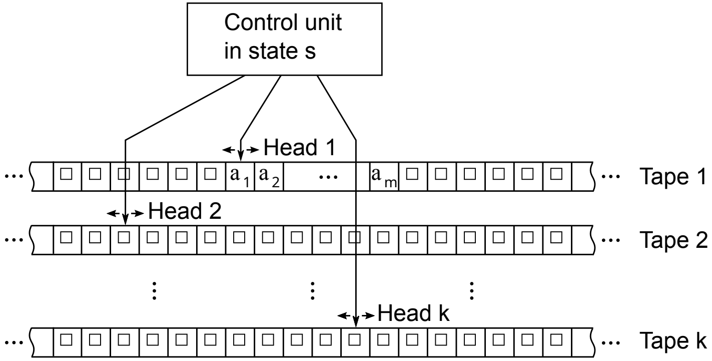
$s_2\ a \rightarrow s_2\ a$ L // go back left, continue to test
$s_2\ b \rightarrow s_2\ b$ L // go back left, continue to test
$s_2\ \square \rightarrow s_0\ \square$ R // at beginning of word, new round

$s_3\ a \rightarrow s_3\ \square$ 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, $a_1, \ldots, a_m \in \Sigma \setminus \{\square\}$ and $s \in S$. $\mathbf{M(s, a_1 \cdots a_m)}$ 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 $2, \ldots, k$ only contain $\square$


Control unit
in state s
Head 1
Head 2
Head k
a_1
a_2
a_m
Tape 1
Tape 2
Tape k

Definition 2.46 (Turing computability)

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

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

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

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

Remark 2.47

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

Remark 2.48

If $\varphi : \mathbb{N}^n \to \mathbb{N}$ 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 $M(s_0, dya(x_1) * dya(x_2) * \cdots * dya(x_n))$, where tape 1 has the content $\cdots \Box dya(y) \Box \cdots$.*
- *If $\varphi(x_1, \dots, x_n)$ is not defined, then M does not stop after starting in $M(s_0, dya(x_1) * dya(x_2) * \cdots * dya(x_n))$.*

Example 2.49

Let $\mathrm{S} : \mathbb{N} \to \mathbb{N}$ with $\mathrm{S}(x) \overset{df}{=} x + 1$. Example 2.43 yields $\mathrm{S} \in \mathrm{TM}$.

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

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

The Python program uses the following variables:

- `s`: index of the state of M
- `b1,...,bk`: lists for the tapes $1, \ldots, k$, where `bi[j]` is index of the character in cell j on tape i
- `h1,...,hk`: positions of the heads on the tapes $1, \ldots, k$

```
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 $\mathrm{dya}^{-1}(w)$
```

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

This shows $f \in \mathrm{WHILE}$. □

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 $s_2 a_3 a_1 \to s_3 a_0 a_2$RL is an instruction of M, then we simulate this by:

```
if (s==2 and b1[h1]==3 and b2[h2]==1): # situation s2 a5 a3
  s = 3                 # go into state s3
  b1[h1] = 0            # write a0 on tape 1
  b2[h2] = 2            # write a2 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 the registers used
- registers are stored in the tapes $1, \ldots, k+1$
- tape $k+2$ is used for arithmetic instructions

Proof.

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

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

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

Only the registers R0, …, Rk are required for the simulation.

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

- the contents of the registers R0, …, Rk are transferred to the tapes $1, \ldots, k+1$ in dyadic representation
- IR is stored in the state of M'

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

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

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 $\cdots \Box \text{dya}([\text{R0}]) \Box \cdots$

Thus M' computes the function φ in dyadic representation. Hence $\varphi \in \text{TM}$. $\Box$

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 $\mathbb{N}^n \to \mathbb{N}^m$.

Definition 2.55

Let $m, n \geq 0$.

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

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

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 $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ are computable.

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

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

$\mathrm{exp}(x,y) \overset{df}{=} x^y$ *(where we define $0^0 = 1$)*

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

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 $g : \mathbb{N}^k \to \mathbb{N}^n$ and $f : \mathbb{N}^n \to \mathbb{N}^m$ are computable, then $f \circ g$ is also computable. As a reminder: $(f \circ g)(x) \overset{df}{=} f(g(x))$.

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 runtime. 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:

$$|\boldsymbol{x}| \overset{df}{=} |\mathrm{dya}(\mathrm{abs}(x))|$$

We write the absolute value of x as **abs($\boldsymbol{x}$)**, 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.

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

Definition 2.58 (Runtime funct. for While programs)

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

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

where the computation steps are counted as follows:

- 1 computational step for the operations + and −, for comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`), for logical operations (`not`, `and`, `or`), for value assignments (`=`), for initializations (`[`b_1`,...,`b_m`] = [0,...,0]`) and for `return`.
- The number of computation steps for $\underset{s}{\texttt{for}}$ i `in range(`a_1`,`a_2`):` 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 $f : \mathbb{N}^m \to \mathbb{N}$. The associated runtime function $t_M : \mathbb{N}^m \to \mathbb{N}$ is defined by

$$\boldsymbol{t_M(x_1, \ldots, x_m)} \overset{df}{=} \text{number of cycles}^{10} \text{ until } M \text{ stops when given } x_1, \ldots, x_m \text{ as input}$$

Definition 2.61 (Runtime function for TMs)

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

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

[10] Again, 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

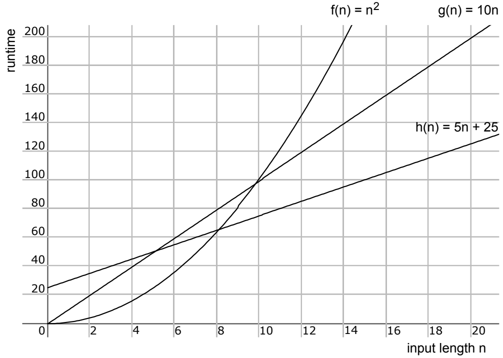
We abstract from the concrete input and consider the runtime for all inputs of a certain length.

Definition 2.62 (Runtime limit wrt input length)

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

- **M computes f in time t** $\overset{df}{\iff}$ 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** $x_1, \dots, x_m$.
- $\boldsymbol{O(t)} \overset{df}{=} \{f : \mathbb{N} \to \mathbb{N} \mid f$ 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)** $\overset{df}{\iff}$ 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 $f(n) > c \cdot h(n)$. $\square$

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

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

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
   r = 0                    # 1
   while(x > y):            # 1   |x| iterations
      y = ((y + y) + 2)     # 3   6 steps per iteration
      r = r + 1             # 2   1 step for last test x>y
   return r                 # 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 $t\colon \mathbb{N} \to \mathbb{N}, t(n) \overset{df}{=} \begin{cases} 13 \cdot n & \text{if } n \geq 1 \\ 7 & \text{else.} \end{cases}$

Example Ctd

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

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

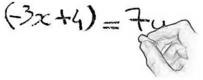
Proof.

Reminder: the algorithm works in time t for $t\colon \mathbb{N} \to \mathbb{N}$, $t(n) \overset{df}{=} \begin{cases} 13 \cdot n & \text{if } n \geq 1 \\ 7 & \text{else.} \end{cases}$

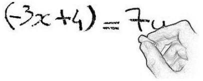
So we have to prove $t \in O(n)$.

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)



Example 2.66 (Runtime of a While program)



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`)	$O(\lvert x\rvert + \lvert y\rvert)$
Division (`x//y`)	$O(\lvert x\rvert + \lvert y\rvert)$
Modulo (`x%y`)	$O(\lvert x\rvert + \lvert y\rvert)$
Access to a list element	$O(1)$
Assignment of a reference to a list	$O(1)$
len(a)	$O(1)$
a += b	$O(\text{len}(b))$
Create/copy list a	$O(\text{len}(a))$
(a+b)	$O(\text{len}(a) + \text{len}(b))$

Remark 2.67

The runtimes for multiplication, division, and modulo are not realistic in a real setting as addition actually requires more than computation step.

However, assuming that addition is possible in one step causes that the runtime in a more realistic scenario (where computing `x + y` *requires* $\min(|x| + |y|)$ *steps) is at most the square of the runtime in "our" model.*

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. $f(x, y) \overset{df}{=} 0$), the successor function and projections (e.g. $g(x, y, z) \overset{df}{=} y$).

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

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

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

One can show:

1. PART $=$ WHILE
2. PRIM $=$ LOOP

In particular, PRIM $\subsetneq$ 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*.

$\Rightarrow$ *massive parallelism*

Restriction: Quantum mechanics does not allow classical instructions, but only unitary transformations
$\Rightarrow$ *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 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** $c_A\colon \mathbb{G} \to \{0,1\}$ is defined as follows:

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

Definition 2.69 (decidability)

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

- $A \subseteq \mathbb{N}^n$ is **decidable** $\overset{df}{\Longleftrightarrow}$ c_A is computable.
- **REC** $\overset{df}{=} \{A \mid \exists n \geq 0 \text{ with } A \subseteq \mathbb{N}^n \text{ and } A \text{ is decidable}\}$ (recursive sets)
- An algorithm **M decides $A \subseteq \mathbb{N}^n$ in time t (resp. O(t))** $\overset{df}{\Longleftrightarrow}$ 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 $G = \{n \mid n \geq 2$ and there exist prime numbers p and q with $2n = p + q\}$[11]
- the set $F = \{n \mid n > 2 \text{ and } \forall a, b, c \in \mathbb{N}^+[a^n + b^n \neq c^n]\}$[12]
- the set $H = \{n \leq 49 \mid n$ will be drawn in the next lottery$\}$[13]

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

[12] In 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.

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

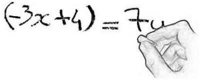
Theorem 2.72

1. *Every finite subset of* $\mathbb{N}^n$ *is decidable.*
2. $A \subseteq \mathbb{N}^n$ *decidable* $\Rightarrow$ $\overline{A} = (\mathbb{N}^n \setminus A)$ *decidable.*
3. $A, B \subseteq \mathbb{N}^n$ *decidable* $\Rightarrow$ $A \cup B$ *and* $A \cap B$ *decidable.*

Proof idea:

- The statements can be easily derived from the definitions

Proof.



$\square$

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.

$x \in U \overset{df}{\iff}$ 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 `f2` in Example 2.24 stops on all inputs.

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

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** $\overset{df}{\Longleftrightarrow}$ there is a total, computable $f : \mathbb{N}^m \to \mathbb{N}^n$, such that for all $x \in \mathbb{N}^m$:

$$x \in A \;\Leftrightarrow\; f(x) \in B.$$

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.

$$B \in \mathrm{REC} \quad \Rightarrow \quad A \in \mathrm{REC}$$

Proof.

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

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

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 Property 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 $\mathbb{P} \in$ REC.

By Property 2.75, $A \in$ REC.

To use Property 2.75 to show the undecidability of sets B we 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 R$i = 0$ GOTO j	$\langle 7, i, j, 0\rangle$
IF R$i > 0$ GOTO j	$\langle 8, i, j, 0\rangle$
STOP	$\langle 9, 0, 0, 0\rangle$

2. Encoding of RAM M with instructions $b_1, \ldots, b_s$:
   $\langle M\rangle \overset{df}{=} \langle\langle b_1\rangle, \ldots, \langle b_s\rangle\rangle$
3. For $i \in \mathbb{N}$ let $M_i \overset{df}{=} \begin{cases} M, & \text{if } i = \langle M\rangle \text{ for a RAM } M \\ M^\star, & \text{if } i \text{ is not code of a RAM,} \end{cases}$
   where $M^\star$ is the RAM with the single instruction STOP.

Observation 2.77

1. *Every RAM is contained in the set $\{M_0, M_1, \ldots\}$.*
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)

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

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

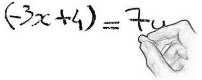
Theorem 2.79

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.



□

Corollary 2.80

K is not decidable.

Proof.

Exercise. $\square$

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 $\mathbb{N}^n \to \mathbb{N}$, then the following set is undecidable.

$$I(S) \overset{df}{=} \{i \in \mathbb{N} \mid \textit{the function } \mathbb{N}^n \to \mathbb{N} \textit{ computed by } M_i \textit{ is in } S\}$$

Proof idea:

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

Proof.

Let $h : \mathbb{N}^n \to \mathbb{N}$ with $h(x_1, \ldots, x_n) \overset{df}{=} n.d.$ We assume $h \notin S$, the other case is treated analogously.
Choose comp. $g : \mathbb{N}^n \to \mathbb{N}$ with $g \in S$ (exists by assumption).

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

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

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

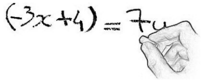
If $x \in K_0$, then M_x halts on x, thus the machine $M[x] = M_{\langle M[x] \rangle}$ computes the function g, hence $f(x) = \langle M[x] \rangle \in I(S)$.
If $x \notin K_0$, then M_x does not halt on x, so the machine $M[x] = M_{\langle M[x] \rangle}$ computes the function h, thus $f(x) = \langle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

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

Example 2.82 (Undecidable sets)

The following sets are undecidable by Rice's theorem.

$$A = \{i \mid M_i \text{ computes } \varphi\} \quad \text{for each comp. } \varphi : \mathbb{N}^n \to \mathbb{N}$$
$$B = \{i \mid \exists x \in \mathbb{N} \text{ such that } M_i \text{ on input } x \text{ returns } 17\}$$
$$C = \{i \mid M_i \text{ halts on input } 17\}$$
$$D = \{(i,j) \mid M_i \text{ and } M_j \text{ compute the same function } \mathbb{N}^n \to \mathbb{N}\}$$



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 $t\colon \mathbb{N} \to \mathbb{N}$ a total function.

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

3
Finite Automata

Finite Automata – Outline

- We learn about deterministic and non-deterministic finite automata (DFA and NFA). They are simple computation models that solve the decision problem of certain languages and are used in many situations (e.g. fast text search or verification).
- We will define DFAs and NFAs in a mathematically precise way and, based on these definitions, investigate their general properties.
- With the power set construction, we obtain an equivalent DFA for each NFA. This means that NFAs cannot accept more languages than DFAs despite model extension.

Finite Automata – Outline II

- With regular expressions, we deal with another formalism for describing languages. We will learn about methods that can be used to convert between NFAs and regular expressions. This shows that DFAs, NFAs and regular expressions have the same expressive power.
- From the fact that we come across the same class via DFAs, NFAs and regular expressions, we conclude that we have found a natural language class here (regular languages). Depending on the use case, we can choose the most suitable representation of regular languages.

Finite Automata – Outline III

- We ask which languages are not regular. In such a case, we have to use a more expressive formalism, to handle the language algorithmically. With the Pumping Lemma, we get to know a general method, with which we can show that a language is not regular.
- The simplicity of DFAs leads to the fact that certain questions can be decided here that are undecidable for RAMs and TMs. We learn procedures for minimization, the equivalence test, and the emptiness test.

3.1
Deterministic Finite Automata

Learning Objectives

After this section you should

1. understand and be able to apply the definition of DFAs (tuple notation), their working mechanism, and graphical representation.
2. understand and be able to apply the definitions of the extended transition function and acceptance by DFAs.
3. be able to construct a DFA for a given simple language that accepts these languages.
4. understand the fast search with DFAs.

With the computation models considered so far (While programs, RAMs, TMs) we have always had the option of storing any amount of data (variables, registers, tapes/cells).
With finite automata, we now examine a computation model where this is not possible. This corresponds to hardware components that can control but cannot store anything.
Despite the limitations, this model can be used to some practically relevant problems that can be handled well and solved quickly. For example:

- fast search in texts, files, DNA sequences, ...
- clear description of processes (e.g. GUI)
- Modeling the behavior of distributed systems

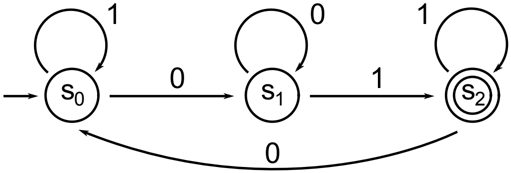
Operation of Deterministic Finite Automata

A deterministic finite automaton (DFA) . . .

- is in one of a finite number of states at any time.
- receives a word as input.
- reads this input word character by character from left to right.
- can change its state after each character read.

Representation of DFAs

- States are represented by circles, the name of the state is in the circle
- State transitions are represented by arrows, the input characters required for the transition are at the arrow
- the start state (there is exactly one) is marked by an unlabeled, incoming arrow
- accepting states are represented by double circles

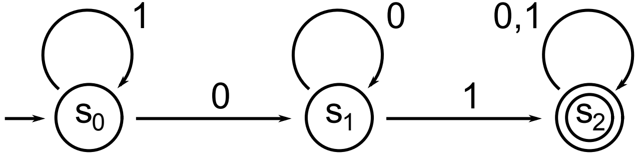

$$\begin{tikzcd}
\arrow[r] & s_0 \arrow[loop above, "1"] \arrow[r, "0"] & s_1 \arrow[loop above, "0"] \arrow[r, "1"] & s_2 \arrow[loop above, "1"] \arrow[ll, bend left, "0"]
\end{tikzcd}$$

Example 3.1 (DFA to test for the subword 01)

We want to determine with the help of a DFA whether the input word from $\{0,1\}^*$ contains the subword 01.

To do this, we need to set up the DFA so that e.g. the inputs 11010 and 1100 end in different states.

The following DFA does this. A word *w* transfers it from the start state to the accepting state if and only if *w* contains the subword 01.



Behavior of the automaton on input 11010:

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[r, "1"] & s_0 \arrow[r, "1"] & s_0 \arrow[r, "0"] & s_1 \arrow[r, "1"] & s_2 \arrow[r, "0"] & s_2 \end{tikzcd}$$

The word 11010 is *accepted* by the automaton because the last state reached (s_2) is accepting.

Behavior of the automaton on input 1100:

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[r, "1"] & s_0 \arrow[r, "1"] & s_0 \arrow[r, "0"] & s_1 \arrow[r, "0"] & s_1 \end{tikzcd}$$

The word 1100 is *not accepted* by the automaton because the last state reached (s_1) is not accepting.

The main features of a DFA are

- Input alphabet
- Finite set of states
- State transitions per state and read input symbol
- Labeled start state
- Accepting states

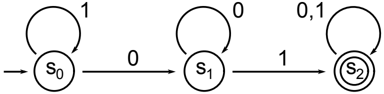
These components describe a DFA precisely.

Definition 3.2 (Deterministic finite automaton)

A **deterministic finite automaton** (**DFA**) is a quintuple $A = (\Sigma, S, \delta, s_0, F)$ with the following properties.

- Σ is a finite, non-empty set (*input alphabet*)
- S is a finite set (*state set*)
- δ is a total function $S \times \Sigma \to S$ (*transition function*)
- $s_0 \in S$ (*start state*)
- $F \subseteq S$ (*set of accepting states*)

Example 3.3 (Tuple representation for Example 3.1)

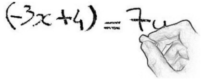


$A = (\Sigma, S, \delta, s_0, F)$ with

- $\Sigma \overset{df}{=} \{0, 1\}$, $S \overset{df}{=} \{s_0, s_1, s_2\}$, $F \overset{df}{=} \{s_2\}$
- $$\delta(s, a) \overset{df}{=} \begin{cases} s_1, & \text{if } s = s_0 \text{ and } a = 0 \\ s_0, & \text{if } s = s_0 \text{ and } a = 1 \\ s_1, & \text{if } s = s_1 \text{ and } a = 0 \\ s_2, & \text{if } s = s_1 \text{ and } a = 1 \\ s_2, & \text{if } s = s_2 \text{ and } a = 0 \\ s_2, & \text{if } s = s_2 \text{ and } a = 1 \end{cases}$$

Example 3.4 (even number 0s and even number 1s)

We design a DFA that recognizes whether the input word from $\{0, 1\}^*$ has an even number of 0s **and** contains an even number of 1s.



The transition function δ specifies how an automaton processes the stream of input characters.

Assuming $w = a_1 a_2 \cdots a_m$ is the input and the machine is in the start state s_0.

1. $\delta(s_0, a_1)$ returns the state after the 1st character, e.g. q_1
2. $\delta(q_1, a_2)$ returns the state after the 2nd character, e.g. q_2

   $\vdots$

m. $\delta(q_{m-1}, a_m)$ returns state after reading w, e.g. q_m

We can read from the last state q_m whether the automaton accepts the input. This is the case if the last state is contained in F.

To describe the behavior of a DFA more comfortably, we extend the transition function.

Definition 3.5 (Extended transition function (DFAs))

The extended transition function of a DFA $A = (\Sigma, S, \delta, s_0, F)$ is the mapping $\overline{\delta}\colon S \times \Sigma^* \to S$ defined as follows.

BC $\overline{\delta}(s, \varepsilon) \overset{df}{=} s$ for all $s \in S$

IS $\overline{\delta}(s, wa) \overset{df}{=} \delta(\overline{\delta}(s, w), a)$ for all $s \in S$, $w \in \Sigma^*$, $a \in \Sigma$

This means: $\overline{\delta}(s, w) =$ state that the DFA reaches, when it starts in s and reads the word w.

Example 3.6 (Extended transition function from 3.4)

The extended transition function of the DFA A from Example 3.4:

$$\begin{aligned} \overline{\delta}(s_{i,j}, w) &\overset{df}{=} s_{k,l} \quad \text{with} \\ k &\overset{df}{=} (i + \text{number of 0s in w}) \bmod 2 \\ l &\overset{df}{=} (j + \text{number of 1s in w}) \bmod 2 \end{aligned}$$

In particular:

$$\begin{aligned} \overline{\delta}(s_{0,0}, \varepsilon) &= s_{0,0} \\ \overline{\delta}(s_{0,0}, 0) &= s_{1,0} \\ \overline{\delta}(s_{0,0}, 1) &= s_{0,1} \\ \overline{\delta}(s_{0,0}, 00) &= s_{0,0} \\ \overline{\delta}(s_{0,0}, 01) &= s_{1,1} \\ \overline{\delta}(s_{0,0}, 10) &= s_{1,1} \\ \overline{\delta}(s_{0,0}, 11) &= s_{0,0} \end{aligned}$$

Definition 3.7 (Acceptance of languages by DFAs)

Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA.

- A word $w \in \Sigma^*$ is **accepted by A** $\overset{df}{\iff} \overline{\delta}(s_0, w) \in F$.
- The language accepted by **A** is

$$\mathbf{L(A)} \overset{df}{=} \{w \in \Sigma^* \mid w \text{ is accepted by } A\}.$$

- The **set of languages accepted by DFAs** is

$$\mathbf{FA} \overset{df}{=} \{L(A) \mid \text{A is a DFA}\}.$$

Which languages are in FA?

1. Set of even numbers in decimal notation?
2. Set of numbers divisible by 7 in decimal notation?
3. Set of prime numbers in decimal notation?
4. Set of square numbers in decimal notation?
5. Set of powers of two in decimal notation?
6. Set of palindromes over the alphabet Σ?

Languages accepted by DFAs are decidable.

Theorem 3.8

All $L \in$ FA are decidable.

Proof idea:

- Simulation of the DFA by a Python program

Proof.

Let $L \in$ FA, i.e. there exists a DFA $A = (\Sigma, S, \delta, s_0, F)$ with $L = L(A)$. We simulate A using a Python program.

```
def A(w):           # input word w[0] w[1] ...  w[n-1]
  n = len(w)        # input length
  s = $s_0$            # current state
  i = 0             # points to current letter
  while (i < n):
    s = $\delta$(s,w[i])  # Program out by case differentiation
    i = i + 1
  if (s $\in$ F):        # Program out by case differentiation
    return 1
  else:
    return 0
```

This means that c_L is computable, i.e. L is decidable. □

Example 3.9 (Fast search with DFAs)

We want to use a DFA to search a specific pattern within a word.

This corresponds to the search in texts, which is known from many applications.

Mathematically, we describe this task by the following pattern recognition problem.

$$\mathbf{PR} \overset{df}{=} \{(v, w) \mid v, w \in \Sigma^* \text{ and } \exists x, y \in \Sigma^* \text{ with } w = xvy\}.$$

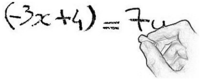
We will construct a DFA A_v for a given pattern v, that accepts exactly those texts w that contain the pattern v.

A_v requires only $O(|w|)$ computation steps for the search. The naive text search method requires $O(|v| \cdot |w|)$ computation steps.

1. Naive method (runtime $O(|v| \cdot |w|)$):

```
def SearchNaively(v,w):     # Input:
  k = len(v)                #         v[0] ...  v[k-1]
  m = len(w)                #         w[0] ...  w[m-1]
  flag = 0
  i = 0                     # Position in w
  while ((flag == 0) and (i <= m-k)):# as long as v not found
    flag = 1                          # and end not reached
    for j in range(0,k):    # run through all letters in v
      if (w[i+j] != v[j]):
        flag = 0
    i = i + 1               # next position in w
  return flag               # 1=found, 0=not found
```

2. Method with DFAs (runtime $O(|\Sigma| \cdot |v| + |w|)$):



3.2
Non-Deterministic Finite Automata

Learning Objectives

After this section you should

1. understand and be able to apply the definition (tuple notation) and working mechanism of NFAs.
2. understand and be able to apply the definitions of the extended transition function and acceptance by NFAs.
3. be able to construct an NFA for a given simple language that accepts this language.
4. understand the power set construction and its application.
5. know the closure properties of FAs dealt with in this section.

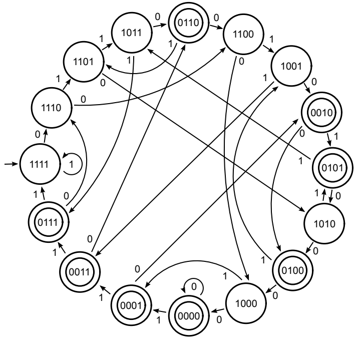
The DFA model is unwieldy for certain tasks.

Example 3.10 (DFA testing fourth last letter for 0)

We are looking for a DFA for the following language: $L = \{w \in \{0,1\}^* \mid |w| \geq 4 \text{ and the fourth character from the right is } 0\}$

As a DFA does not notice when the fourth last character comes when reading the input, the last 4 characters read must always be saved in the state.

This leads to the following DFA with 16 states (it is the smallest DFA that accepts L).



We extend the DFA model: Non-deterministic finite automata (NFAs) can simultaneously transition to several states at the same time. On the one hand, this makes the design more comfortable, but on the other hand, it makes simulations more complex.

Operation of non-deterministic finite automata

A non-deterministic finite automaton (NFA) ...

- can be in several states at the same time.
- receives a word as input.
- reads this input word character by character from left to right.
- can switch to no/one/multiple state(s) after each character read.
- accepts the input word if at least one of the states reached at the end is accepting.

NFAs are displayed graphically like DFAs.

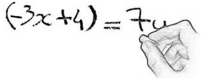
The extension of DFAs to NFAs has two advantages:

1. The design of automata is simplified.
2. The representation of languages by automata becomes more compact: NFAs often require fewer states for the same language than the smallest DFA.

Example 3.11 (NFA testing fourth last letter for 0)

We want to use an NFA to determine whether the input word belongs to the language L.

$L = \{w \in \{0,1\}^* \mid |w| \geq 4, \text{ fourth character from the right is } 0\}$



With the following definition we specify the NFA model.

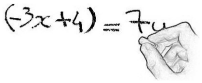
Definition 3.12 (Non-deterministic finite automaton)

A **nondeterministic finite automaton** (**NFA**) is a quintuple $A = (\Sigma, S, \delta, s_0, F)$ with the following properties.

- Σ is a finite, non-empty set (*input alphabet*)
- S is a finite set (*state set*)
- δ is a total function $S \times \Sigma \to \mathcal{P}(S)$ (*transition function*)
- $s_0 \in S$ (*initial state*)
- $F \subseteq S$ (*set of accepting states*)

The only difference to the definition of DFAs is that the target set of δ is now the power set of S. This means that δ specifies a *set* of subsequent states for each state and input symbol.

Example 3.13 (Tuple representation for Ex.3.11)



We want to extend δ again to words $w = a_1 a_2 \cdots a_m$. Note the following:

- After inputting $a_1 a_2 \cdots a_i$, an NFA is in a set P of states.
- For each state $p \in P$ reached so far, $\delta(p, a_{i+1})$ specifies a set of subsequent states.
- We obtain all states that can be reached after inputting a_{i+1} by unifying $\delta(p, a_{i+1})$ over all $p \in P$.

Definition 3.14 (Extended transition funct. f. NFAs)

The extended transition function of an NFA $A = (\Sigma, S, \delta, s_0, F)$ is the function $\overline{\delta} : S \times \Sigma^* \to \mathcal{P}(S)$ defined as follows.

BC $\overline{\delta}(s, \varepsilon) \overset{df}{=} \{s\}$ for all $s \in S$

IS $\overline{\delta}(s, wa) \overset{df}{=} \bigcup_{p \in \overline{\delta}(s,w)} \delta(p, a)$ for all $s \in S$, $w \in \Sigma^*$, $a \in \Sigma$

This means: $\overline{\delta}(s, w)$ = the set of states that the NFA reaches simultaneously when it starts in s and reads the word w.

Definition 3.15 (Acceptance of languages by NFAs)

Let $A = (\Sigma, S, \delta, s_0, F)$ be an NFA.

- A word $w \in \Sigma^*$ is **accepted by A** $\overset{df}{\iff} \overline{\delta}(s_0, w) \cap F \neq \emptyset$.
- The **language accepted by A** is

$$\mathbf{L}(\mathbf{A}) \overset{df}{=} \{w \in \Sigma^* \mid w \text{ is accepted by } A\}.$$

A word is therefore accepted by an NFA if at least one computation path leads to an accepting state.

It is irrelevant whether there are also paths for the same input that end earlier or lead to non-accepting states.

Example 3.16 (Extended transition funct. from 3.11)

For example, the following holds for the extended transition function of the NFA A from Example 3.11:

$$
\begin{array}{ll}
\overline{\delta}(s_0, \varepsilon) = \{s_0\} & \overline{\delta}(s_0, 0) = \{s_0, s_1\} \\
\overline{\delta}(s_0, 1) = \{s_0\} & \overline{\delta}(s_0, 00) = \{s_0, s_1, s_2\} \\
\overline{\delta}(s_0, 01) = \{s_0, s_2\} & \overline{\delta}(s_0, 10) = \{s_0, s_1\} \\
\overline{\delta}(s_0, 11) = \{s_0\} & \overline{\delta}(s_0, 000) = \{s_0, s_1, s_2, s_3\} \\
\overline{\delta}(s_0, 001) = \{s_0, s_2, s_3\} & \overline{\delta}(s_0, 010) = \{s_0, s_1, s_3\}
\end{array}
$$

Comparing Determinism and Non-Determinism

Despite the use of non-determinism, NFAs do not accept more languages than DFAs. To prove this, we use a general procedure (power set construction), to convert an arbitrary NFA into an equivalent DFA.

Theorem 3.17 (Rabin and Scott's theorem)

For every $L \subseteq \Sigma^*$:

$$L \in \mathrm{FA} \quad \Leftrightarrow \quad \textit{there exists an NFA that accepts } L$$

Proof idea:

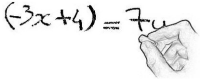
- An NFA can be in several states at the same time, but there are only finitely many possibilities: the number of subsets of S is $2^{|S|}$.
- When given an NFA $A = (\Sigma, S, \delta, s_0, F)$, the following DFA A' accepts the same language: $(\Sigma, \mathcal{P}(S), \delta', \{s_0\}, F')$ with $\delta'(M, a) = \bigcup_{s \in M} \delta(s, a)$ for $a \in \Sigma$ and $M \subseteq S$ and $F' = \{M \subseteq S \mid M \cap F \neq \emptyset\}$.

Without proof.

Example 3.18 (Power set construction)

We carry out the power set construction for the following NFA.

$$\begin{tikzcd}
\arrow[r] & s_0 \arrow[loop above, "{0,1}"] \arrow[r, "0"] & s_1 \arrow[r, "1"] & s_2 \arrow[r, "0"] & \boxed{s_3}
\end{tikzcd}$$



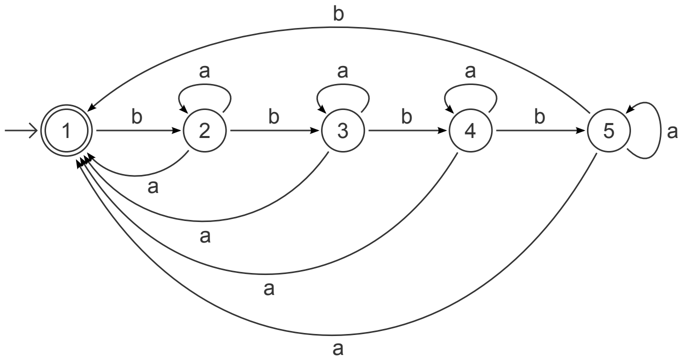
Note that the size of the set of states can grow from $|S|$ to $2^{|S|}$ when executing the power set construction.

This even holds if only the states that can be reached from the start state are counted.

Example 3.19 (Number of states goes from n to 2^n)

The following NFA has 5 states. Power set construction yields an equivalent DFA with 2^5 states.

One can show that there is no DFA with less than 2^5 states that accepts the same language. For each $n > 0$, an analogous example with n states exists.



Definition 3.20 (Concatenation of languages)

Let L, L' *be* $\subseteq \Sigma^*$.

$$\begin{aligned}
\boldsymbol{L \cdot L'} &\overset{df}{=} \{uv \mid u \in L \text{ and } v \in L'\} \\
\boldsymbol{L^0} &\overset{df}{=} \{\varepsilon\} \\
\boldsymbol{L^{k+1}} &\overset{df}{=} L \cdot L^k \quad \text{for } k \geq 0 \\
\boldsymbol{L^*} &\overset{df}{=} \bigcup_{k \geq 0} L^k = \{u_1 u_2 \cdots u_m \mid m \geq 0 \text{ and } u_1, \ldots, u_m \in L\}
\end{aligned}$$

We show that FA is closed under $\cup$, $\cap$, $\overline{\ }$, $\cdot$, and $*$.

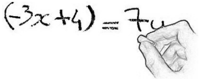
Theorem 3.21 (Closure properties of FA)

Let $L, L' \in \mathsf{FA}$. *Then* $\overline{L}, L \cup L', L \cap L', L \cdot L', L^* \in \mathsf{FA}$.

Proof idea:

- Simple constructions of DFAs and NFAs

Proof sketch.



$\square$

3.3
Regular Expressions and Regular Sets

Learning Objectives

After this section you should

1. understand the syntax and semantics of regular expressions.
2. be able to convert NFAs into regular expressions and regular expressions into NFAs. To do this, you need to understand the method used in the proof of Theorem 3.30.

Finite automata (DFAs and NFAs) are models to *accept* languages. (e.g. fast search with DFAs)

On the other hand, automata are often unsuitable for describing languages *easily*. (e.g. input of search patterns such as "?ouse")

Regular expressions are a formalism with which we can specify languages very easily.

We will see that regular expressions describe precisely those languages that can be described by DFAs.

Example 3.22 (Search in Word)

? stands for any character (or none)

***** stands for 0 or more characters

This can be used to search for the following patterns in texts.

- The search for the pattern **?ouse** finds **mouse**, **house**, **douse**, and **louse**, among others.
- The search for the pattern ***iss*iss*** finds **Mississippi** and **dissidents dissent**, among others.

With the help of regular expressions, these and other patterns can be described very easily (and automatically converted into corresponding DFAs for the fast search).

Definition 3.23 (Syntax and semantics of reg. expr.)

Let Σ be an alphabet. We define **regular expressions** γ and the languages $L(\gamma)$ described by them.

BC $\boldsymbol{\emptyset}$, $\boldsymbol{\varepsilon}$ and $\boldsymbol{a}$ are regular expressions (where $a \in \Sigma$).
Semantics: $\boldsymbol{L(\emptyset)} \overset{df}{=} \emptyset$, $\boldsymbol{L(\varepsilon)} \overset{df}{=} \{\varepsilon\}$, $\boldsymbol{L(a)} \overset{df}{=} \{a\}$.

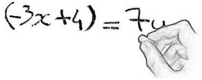
IS If α, β are reg. expressions, so are $\boldsymbol{(\alpha + \beta)}$, $\boldsymbol{(\alpha \cdot \beta)}$, $\boldsymbol{\alpha^*}$.
Semantics:
$$\begin{aligned}\boldsymbol{L(\alpha + \beta)} &\overset{df}{=} L(\alpha) \cup L(\beta)\\ \boldsymbol{L(\alpha \cdot \beta)} &\overset{df}{=} L(\alpha) \cdot L(\beta)\\ \boldsymbol{L(\alpha^*)} &\overset{df}{=} L(\alpha)^*\end{aligned}$$

Remarks:

1. Unnecessary brackets and $\cdot$ can be omitted.
2. Binding order: $*$ binds the strongest, then $\cdot$, then $+$.
   $0 + 01^*$ therefore stands for $(0 + (0 \cdot (1^*)))$.
3. Denotation of reg. expressions by lowercase Greek letters.
4. Σ must not contain any of the special characters $\emptyset$, ε, $+$, $\cdot$, $*$, (,).

Example 3.24

We are looking for a regular expression for the set of all words over the alphabet $\{0, 1\}$, in which there are never two 0s or two 1s in a row.



Example 3.25 (Further examples of reg. expr.)

1. Regular expression for the set of all words from $\{0, 1, 2\}^*$, in which there are two 2s between which neither a 1 nor a 2 occurs:
   $(0 + 1 + 2)^* \cdot 2 \cdot 1^* \cdot 2 \cdot (0 + 1 + 2)^*$
2. Regular expression for the set of all words from $\{a, b\}^*$, in which there are two a's with exactly 4 letters in between:
   $(a + b)^* a(a + b)(a + b)(a + b)(a + b)a(a + b)^*$

Syntax versus Semantics

Note the difference between a regular expression α (syntax) and the language $L(\alpha)$ described by α (semantics).

1. Each regular expression describes *exactly one* language.
2. However, a language can be described by *infinitely many* regular expressions, because
   $L(\alpha) = L(\alpha + \emptyset) = L(\alpha + \emptyset + \emptyset) = \cdots$.

Aware of this difference, we will use α instead of $L(\alpha)$ for simplicity (but not $L(\alpha)$ instead of α).

Examples:

- If we speak of "the language 0^*", we mean the language described by 0^*.
- If we write $A = \alpha \cdot B \cup C$ (with languages A, B, C), this stands for $A = L(\alpha) \cdot B \cup C$.

We now want to show that the languages describable by regular expressions are exactly the languages from FA.

Definition 3.26 (Regular languages)

- A language L is called **regular** $\overset{df}{\iff}$ there exists a regular expression α with $L = L(\alpha)$.
- **REG** $\overset{df}{=} \{L \mid L \text{ is regular}\}$

First, let's look at the inclusion easier to show.

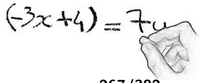
Theorem 3.27

REG $\subseteq$ FA.

Proof idea:

- Induction over the structure of regular expressions, using the closure properties from Theorem 3.21.

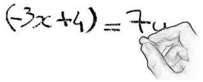
Proof.



$\square$

Example 3.28 (Computing an NFA for a fiven regular expression)

We create an NFA for the language $(ab^* + a)^*$.



For the inclusion FA $\subseteq$ REG one can show that every language accepted by an NFA can be described by a regular expression.

The proof is constructive, i.e. it provides a general procedure that constructs a regular expression for a given NFA.
We only sketch the proof idea and look at the general procedure using an example.

Lemma 3.29

Let $L, A, B \subseteq \Sigma^*$ with $\varepsilon \notin A$. If $L = A \cdot L \cup B$, then $L = A^* B$.

Without proof.

Theorem 3.30

FA $\subseteq$ REG.

Proof idea:

- Let $A = (\Sigma, S, \delta, s_0, F)$ be an NFA.
- For each state s we define the set of words accepted from s: $L_s \overset{df}{=} \{w \mid \overline{\delta}(s, w) \cap F \neq \emptyset\}$.
- For each state s we obtain an equation: $L_s = \bigcup_{a \in \Sigma} a \cdot \bigcup_{s' \in \delta(s,a)} L_{s'} \cup X$, where $X = \begin{cases} \{\varepsilon\} & \text{if } s \in F \\ \emptyset & \text{else.} \end{cases}$
- We solve the resulting system of equations for $L_{s_0} = L(A)$ by substitution and applying Lemma 3.29. In this way, we obtain a regular expression for $L(A)$.

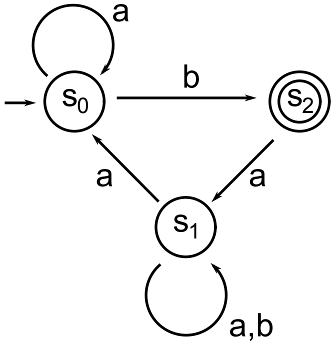
Without proof.

Corollary 3.31

FA = REG.

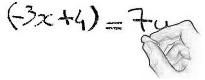
Example 3.32 (Computing a regular expression)

We use the method of the proof of Theorem 3.30 to construct a reg. expression for the following NFA A over the alphabet $\{a, b\}$.



Example 3.33

We construct a regular expression for the DFA created in Example 3.4.



3.4
The Pumping Lemma for Regular Languages

Learning Objectives

After this section you should

1. understand and be able to reproduce the exact formulation of the Pumping Lemma.
2. be able to apply the Pumping Lemma to show for suitable languages that they are not regular.
3. understand the proof of the Pumping Lemma.

We want to find out for a given language L whether it is regular or not.

- If yes, we can represent L by DFAs, NFAs, or regular expressions as required.
- If not, we must use a more expressive formalism, to be able to deal with L algorithmically.

Example 3.34

We are looking for a DFA or a regular expression for the following language.

$$L_{01} \overset{df}{=} \{0^k 1^k \mid k \in \mathbb{N}\} = \{\varepsilon, 01, 0011, 000111, \ldots\}$$

If we don't find one, it could be because

1. we didn't have the right idea yet, or
2. L_{01} is not regular at all.

It would certainly be helpful if we could prove $L \notin \text{REG}$.

However, we cannot try out all possible DFAs or expressions, we have to be a little more imaginative.

With the Pumping Lemma, we get to know a general method, with the help of which we can show that a given language is not regular.

The Pumping Lemma has the following structure:

"Each regular language L has property A"

If we prove for some language L that it does not have the property A, then L cannot be regular (contraposition).

Before we derive property A, we make a preliminary consideration.

Are there two people in the Oberpfalz with precisely the same number of scalp hairs (people with 0 hairs are excluded)?

Pidgeon Hole Principle

From Wikipedia: [...] the number of hairs varies within certain ranges depending on hair color. For example, blondes have an average of 150,000 hairs, black-haired people 110,000, brunettes 100,000 and redheads 75,000.

The Oberpfalz has $> 10^6$ inhabitants. It seems reasonable to assume that 500,001 inhabitants have at least one and at most 500,000 scalp hairs.

Build 500,000 boxes (pidgeon holes) and put the inhabitants with exactly 1 hair into box 1, the inhabitants with 2 hairs into box 2,...

We've put 500,001 people into 500,000 boxes. So there is at least one box with at least 2 people inside. These have exactly the same number of scalp hairs.

Back to the Pumping Lemma: We obtain the property A (that all reg. languages have) from the following observation:

1. If we consider a DFA with k states when inputting a word of length $\geq k$, then at least one state is run through twice.
2. This loop can also be run through several times, without changing the acceptance behavior of the DFA.

Definition 3.35 (3-Pumpability)

A language L is **3-pumpable**[14] $\Leftrightarrow$ there exists $k \in \mathbb{N}^+$ (called **3-pumping number for L**) such that every word $w \in L$ with $|w| \geq k$ can be split into $w = xyz$ with

1. $y \neq \varepsilon$ (i.e., the pumping part is not empty)
2. $|xy| \leq k$ (i.e., the pumping part is close to the beginning of the word)
3. $\forall_{i \geq 0} xy^iz \in L$ (i.e., when pumping the middle part, the pumping part, up/down, we stay in L)

[14] The choice of the prefix "3-" in "3-pumpable" cannot be understood right now, but will become clear later on.

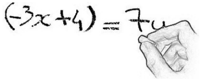
Lemma 3.36 (Pumping Lemma for REG)

Every regular language L is 3-pumpable.

Proof idea:

- We consider a DFA A with k states when inputting a word $w \in L(A)$.
- If w has at least k letters, at least one state is passed several times (pidgeon hole principle).
- I.e. we have gone around in a circle at least once.
- We can run through this circle any number of times without changing the state the automaton is in after reading the word.
- Accordingly, it holds for all regular languages L: In words of sufficient length $w \in L$ we find ranges y, which we can multiply arbitrarily without leaving the language L.

Proof.



□

When applying the Pumping Lemma, we will show that a certain languages is not 3-pumpable (and thus not regular).

Recall: L is 3-pumpable $\Leftrightarrow$

1. there is $k \in \mathbb{N}^+$ such that
2. each $w \in L^{\geq k}$ (notation: $L^{\geq k} \overset{df}{=} \{w \in L \mid |w| \geq k\}$)
3. has a decomposition $w = xyz$ with $|xy| \leq k$ and $y \neq \varepsilon$
4. such that $\forall i \in \mathbb{N}\, xy^i z \in L$

In order to prove $L \notin \text{REG}$, it suffices to show "L not 3-pumpable", i.e.:

1. **For all** $k \in \mathbb{N}^+$
2. **there exists** $w \in L^{\geq k}$ such that
3. **for all decompositions** $w = xyz$ with $|xy| \leq k$ and $y \neq \varepsilon$
4. **there exists** $i \in \mathbb{N}$ with $xy^i z \notin L$.

(note that for all statements B: $\neg\forall_x B \equiv \exists_x \neg B$ and $\neg\exists_x B \equiv \forall_x \neg B$)

Example 3.37

$L_{01} \notin \mathsf{REG}$.

Proof.

Goal: For all $k \in \mathbb{N}^+$ there exists $w \in L^{\geq k}$ such that for all decompositions $w = xyz$ with $|xy| \leq k$ and $y \neq \varepsilon$ there exists $i \in \mathbb{N}$ with $xy^iz \notin L_{01}$.

Let $k \in \mathbb{N}^+$. Choose $w = 0^k1^k \in L^{\geq k}$. Let $w = xyz$ be a decomposition with $|xy| \leq k$ and $y \neq \varepsilon$. Choose $i = 0$. The only letters in xy are 0s (as $|xy| \leq k$). Then $xy^0z = xz = 0^{k-|y|}1^k$ with $k - |y| \neq k$. Thus $xy^0z \notin L_{01}$. □

3.5
Equivalence and Minimization of Automata

Learning Objectives

After this section you should

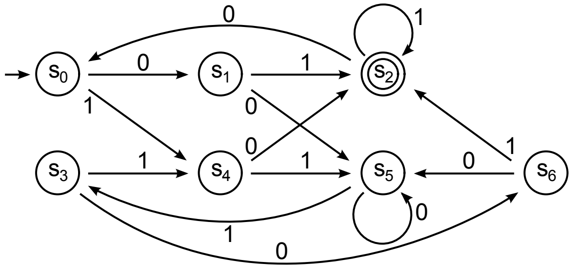
1. be able to construct the minimal automaton of a DFA.
2. be able to perform the equivalence test and the emptiness test for DFAs.
3. know the definitions of equivalence and distinguishability of states.

The essential measure of complexity of finite automata is the number of states. When modeling systems by finite automata, DFAs with many millions of states can arise. We will learn a method with the help of which we can minimize the number of states without changing the accepted language.

Equivalence of DFAs

First, we want to determine whether two DFAs accept the same language. To do this, we define the equivalence of states.

Example 3.38 (Equivalence of states)



For each word $w \in \{0,1\}^*$ it holds:

$$w \text{ is accepted from } s_0 \iff w \text{ is accepted from } s_3.$$

The states s_0 and s_3 are equivalent in this sense.

Definition 3.39 (Equivalence of states)

Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA. Two states $s_1, s_2 \in S$ are called **equivalent** if for all $w \in \Sigma^*$

$$\overline{\delta}(s_1, w) \in F \;\Leftrightarrow\; \overline{\delta}(s_2, w) \in F.$$

For the DFA in Example 3.38 it holds, among others:

- s_1, s_2 are not equivalent, since $\overline{\delta}(s_1, \varepsilon) \notin F$ but $\overline{\delta}(s_2, \varepsilon) \in F$
- s_0, s_5 are not equivalent, since $\overline{\delta}(s_0, 01) \in F$ but $\overline{\delta}(s_5, 01) \notin F$
- s_1, s_6 are equivalent
- s_0, s_3 are equivalent

For the proof of non-equivalence, we simply specify a word as *witness*. But how can we determine equivalence? After all, we can't try out all $w \in \Sigma^*$.

In order to come up with an equivalence test for states, we define the set of distinguishable pairs of states.

Definition 3.40 (Distinguishable pairs of states)

Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA and $s_1, s_2 \in S$.

BC If $s_1 \in F \Leftrightarrow s_2 \notin F$, then s_1 and s_2 are **0-distinguishable**.

IS If $a \in \Sigma$, $\delta(s_1, a) = s_1'$, $\delta(s_2, a) = s_2'$, and s_1' and s_2' are k-distinguishable states for some $k \in \mathbb{N}$, then s_1 and s_2 are **$(k+1)$-distinguishable**.

States s_1 and s_2 are called **distinguishable** if and only if there is some $k \in \mathbb{N}$ so that s_1 and s_2 are k-distinguishable.

We first consider that distinguishable pairs of states are not equivalent.

Property 3.41

Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA. If two states $s_1, s_2 \in S$ are distinguishable, then they are not equivalent.

Proof.

We prove inductively: For all $k \in \mathbb{N}$ and all states s_1 and s_2, if s_1 and s_2 are k-distinguishable, then they are not equivalent.

BC Let s_1 and s_2 be 0-distinguishable, i.e., $s_1 \in F \Leftrightarrow s_2 \notin F$. Then s_1 and s_2 are not equivalent because of witness ε.

IS Let s_1 and s_2 be $(k+1)$-distinguishable and assume that all k-distinguishable pairs of states are not equivalent (induction hypothesis).

As s_1 and s_2 are $(k+1)$-distinguishable, there are $a \in \Sigma$, $s_1' = \delta(s_1, a)$, and $s_2' = \delta(s_2, a)$ so that s_1' and s_2' are k-distinguishable.

By induction hypothesis, s_1' and s_2' are not equivalent.

Let $w \in \Sigma^*$ be a witness for the non-equivalence of s'_1 and s'_2, i.e., $\overline{\delta}(s'_1, w) \in F \Leftrightarrow \overline{\delta}(s'_2, w) \notin F$.

Then $\overline{\delta}(s_1, aw) = \overline{\delta}(s'_1, w)$ and $\overline{\delta}(s_2, aw) = \overline{\delta}(s'_2, w)$.

So $\overline{\delta}(s_1, aw) \in F \Leftrightarrow \overline{\delta}(s_2, aw) \notin F$.

This means that s_1, s_2 are not equivalent because of the witness aw. □

The converse implication of Property 3.41 also holds.

Property 3.42

Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA. If two states $s_1, s_2 \in S$ are not equivalent, then they are distinguishable.

Proof.

We inductively prove: $\forall_{k \in \mathbb{N}} \forall_{s_1, s_2 \in S}$: if there is a word w of length $\leq k$ that witnesses the non-equivalence of s_1 and s_2 (i.e., $\overline{\delta}(s_1, w) \in F \Leftrightarrow \overline{\delta}(s_2, w) \notin F$), then s_1 and s_2 are distinguishable.

BC $k = 0$. Let $s_1, s_2 \in S$. If w with $|w| \leq k$ (i.e., $w = \varepsilon$) witnesses the non-equivalence of s_1 and s_2, then we have $s_1 \in F \Leftrightarrow s_2 \notin F$, so s_1 and s_2 are distinguishable.

IS Assume that all pairs of states that are witnessed to be non-equivalent by a word of length $\leq k$, are distinguishable (induction hypothesis).

Let $s_1, s_2 \in S$ and $w = av$ for some word v with $|v| = k$ be a witness for the non-equivalence of s_1 and s_2.

The states $s_1' \overset{df}{=} \overline{\delta}(s_1, a)$ and $s_2' \overset{df}{=} \overline{\delta}(s_2, a)$ are not equivalent because of the witness v.

By $|v| = k$ and the ind. hypothesis, s_1', s_2' are distinguishable.

By Definition 3.40, s_1 and s_2 are also distinguishable. $\square$

Corollary 3.43

Let $A = (\Sigma, S, \delta, s_0, F)$ *be a DFA and* $s_1, s_2 \in S$*.*

$$s_1, s_2 \text{ are equivalent} \quad \Leftrightarrow \quad s_1, s_2 \text{ are not distinguishable}$$

Proof.

follows from Properties 3.41 and 3.42. □

We can therefore determine the equivalence of states by determining their distinguishability according to Definition 3.40. The following algorithm does this.

Algorithm 3.44

Input: DFA $A = (\Sigma, \{1, \ldots, n\}, \delta, 1, F)$

1. $U =$ *List of all* $\{i, j\}$ *with* $1 \leq i < j \leq n$.
2. *Mark in* U *all* $\{i, j\}$ *with* $(i \in F \Leftrightarrow j \notin F)$.
3. *Mark all* $\{i, j\}$ *in* U*, that are not marked and for which there is an* $a \in \Sigma$ *such that* $\{\delta(i, a), \delta(j, a)\}$ *is marked. If elements have been newly marked, execute step 3 again.*
4. *It holds:* i, j *are not equivalent* $\Leftrightarrow$ $\{i, j\}$ *is marked in* U.

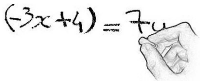
We estimate the runtime of the algorithm.

Assumption: $O(1)$ computation steps for the test $i \in F$,
the computation of $\delta(i, a)$ and
the search for a pair (i, j) in the list U.

Lines 1–2: $O(n^2)$
Line 3: $O(n^2)$ runs, per run $O(|\Sigma| \cdot n^2)$ computation steps

Total runtime: $O(|\Sigma| \cdot n^4)$, where n=number of states of A

Example 3.45 (Computing distinguishable states)



We use the algorithm to compute the distinguishable states to test whether two DFAs accept the same language (in this case we speak of equivalent DFAs).

$$\textbf{EquivalentDFA} \overset{df}{=} \{(A_1, A_2) \mid A_1, A_2 \text{ are DFAs with the same input alphabet and } L(A_1) = L(A_2)\}$$

Algorithm 3.46 (Decision algorithm for EquivalentDFA)

Input: *DFAs A_1, A_2 with the same input alphabet*

1. *We take A_1 and A_2 as* one *DFA, by drawing both automata next to each other and choosing the initial state of A_1.*
2. *Determine the distinguishable states with the help of Algorithm 3.44.*
3. *A_1 and A_2 are equivalent if and only if the two start states are indistinguishable.*

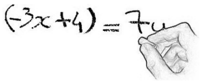
Theorem 3.47

EquivalentDFA *is decidable. The runtime when inputting the DFAs* $A_1 = (\Sigma, S_1, \delta_1, s_{10}, F_1)$ *and* $A_2 = (\Sigma, S_2, \delta_2, s_{20}, F_2)$ *is* $O(|\Sigma| \cdot n^4)$*, where* $n = |S_1| + |S_2|$*.*

Proof.

follows from Corollary 3.43 and Algorithm 3.46. □

Example 3.48 (Equivalence test for DFAs)



Minimization of DFAs

By determining equivalent states, we have already done most of the work to minimize DFAs.

It follows directly from Definition 3.39 that we can summarize equivalent states, without changing the accepted language. We will see that this gives us a minimal DFA.

Property 3.49

Let $(\Sigma, S, \delta, s_0, F)$ be a DFA. The equivalence of states according to Definition 3.39 is an equivalence relation over S, i.e.,

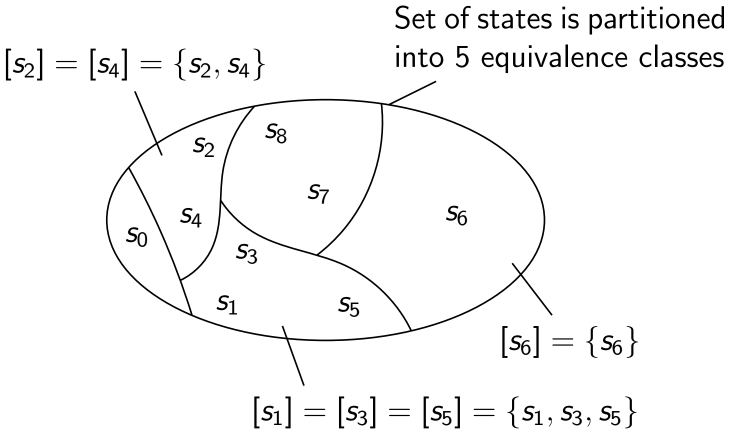
1. *Every state is equivalent to itself (reflexivity).*
2. *If p is equivalent to q, then q is equiv. to p (symmetry).*
3. *If p is equivalent to q and q is equivalent to r, then p is also equivalent to r (transitivity).*

Proof.

Exercise. $\square$

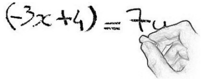
Thus, the set of states is *partitioned* into blocks of equivalent states. These blocks are called *equivalence classes*. The equivalence class containing the state s is defined as

$$[s] \overset{df}{=} \{s' \in S \mid s \text{ and } s' \text{ are equivalent}\}.$$



Set of states is partitioned into 5 equivalence classes
$[s_2] = [s_4] = \{s_2, s_4\}$
s_8
s_2
s_7
s_4
s_6
s_0
s_3
s_1
s_5
$[s_6] = \{s_6\}$
$[s_1] = [s_3] = [s_5] = \{s_1, s_3, s_5\}$

Example 3.50 (Partitioning of the state set from 3.38)



If we take each equivalence class as *one* state, we obtain the following method for minimizing DFAs.

Minimization algorithm for DFAs

Algorithm 3.51

Input: *DFA* $A = (\Sigma, S, \delta, s_0, F)$

Output: *DFA* A' *with min. number of states and* $L(A) = L(A')$

1. *Remove from* S *all states that cannot be reached from* s_0.
2. *Determine the distinguishability of all pairs of states.*
3. *Determine all equivalence classes* $[s]$ *with* $s \in S$.
4. *Define the DFA* $A' = (\Sigma, S', \delta', [s_0], F')$ *with*
   - $S' = \{[s] \mid s \in S\}$
   - $\delta'([s], a) = [\delta(s, a)]$ *for* $s \in S$ *and* $a \in \Sigma$
   - $F' = \{[s] \mid s \in F\}$

Remark 3.52

1. *The set of states reachable from s_0 can be determined using the following algorithm:*

```
$R = \{s_0\}$
while ($\exists s \in R \, \exists a \in \Sigma, \delta(s, a) \notin R$):
    $R = R \cup \{\delta(s, a)\}$
```

2. *We still have to prove the* well-definedness *of δ'. I.e. we have to show that our definition is not ambiguous.*

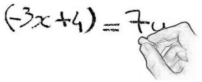
   This is not clear at first because if s and s' belong to the same equivalence class X, then $X = [s] = [s']$.

   Thus, $\delta'(X, a)$ is defined by $\delta'([s], a) \overset{df}{=} [\delta(s, a)]$
   but also by $\delta'([s'], a) \overset{df}{=} [\delta(s', a)]$.

   However, this is not a problem, because the following holds for all equivalent states s, s': $[\delta(s, a)] = [\delta(s', a)]$.

   Consequently, the definition of δ' is independent of the choice of the representative *s and is therefore well-defined.*

Example 3.53 (Minimization of DFA from 3.38)



Algorithm 3.51 provides an equivalent DFA with a minimum number of states.

Theorem 3.54

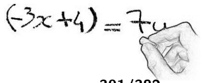
Let $A = (\Sigma, S, \delta, s_0, F)$ be a DFA and let A' be the DFA output by the minimization algorithm (Algorithm 3.51). Then:

1. *$L(A) = L(A')$.*
2. *No DFA A'' with fewer states than A' accepts $L(A)$.*

Proof idea:

- 1. holds, since only equivalent states are summarized.
- 2. Indirect proof: assume there is a smaller DFA A''.
- Choose one word per state from A' that leads to the state.
- Pidgeon hole principle: in A'' two words lead to the same state.
- Contradicts the non-equivalence of the corresponding states in A'

Proof.



□

You can even show that the minimal automaton (except for the renaming of states) is *uniquely* determined. Therefore, we can speak of **the** minimal automaton of a given DFA.

For NFAs, the minimal automaton is not uniquely determined and no efficient algorithm for computing a minimal automaton is known. It is even widely assumed that there exists none.

Emptiness and Finiteness Test for DFAs

It is possible to decide whether a DFA accepts the empty language or accepts a finite language. (cf. undecidability for RAMs, 2.82)

$$\begin{aligned} \textbf{EmptyDFA} &\overset{df}{=} \{A \mid A \text{ is a DFA with } L(A) = \emptyset\} \\ \textbf{FiniteDFA} &\overset{df}{=} \{A \mid A \text{ is a DFA and } L(A) \text{ is finite}\} \end{aligned}$$

Theorem 3.55

EmptyDFA *and* FiniteDFA *are decidable.*

Proof.

For EmptyDFA, use the algorithm from Remark 3.52.1 to test whether an accepting state can be reached.
For FiniteDFA, the proof is an exercise. □

4
Formal Languages

Formal Languages – Outline

- For many applications (e.g. programming or description languages), regular languages are a too restricted formalism. More expressive languages must be used there.
- On the other hand, we are also always interested in *efficient*, automatic processing of the languages used (e.g. fast parsing of a Python program).
- The expressive power of a language often stands in the way of efficient, automatic processing. This means that simple languages can be processed quickly, while the processing of expressive languages requires more effort.

Formal Languages – Outline II

- In this chapter, we will get to know the concept of generating languages using generative grammars. Starting from a start symbol, words are derived by applying substitution rules (also known as production rules).
- Grammars of different expression strength then lead to different language classes. Here we get to know the classes of the Chomsky hierarchy and examine them in more detail later in the chapter.
- The class of context-free languages plays an important role. The essential components of almost all programming and description languages are context-free.
  We will learn about the CYK algorithm, which can be used to efficiently decide context-free languages.

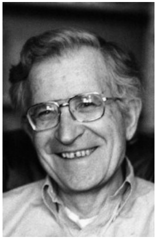
4.1
The Chomsky Hierarchy

Learning Objectives

After this section you should

1. know the terms generative grammar, grammar of type 0/1/2/3, language of type 0/1/2/3, Chomsky hierarchy.
2. understand how languages are generated by generative grammars.
3. be able to notate simple grammars in tuple notation.

Noam Chomsky
(linguist)



Chomsky's idea:

> *Formalization of natural languages such that all syntactically correct sentences of a language are generated by a system of rules (a so-called generative grammar).*

Influence on Linguistics

- Chomsky's idea is regarded as a milestone in linguistics.
- Generative grammars can be divided into a hierarchy according to their expressive power, the so-called *Chomsky hierarchy* (first described in 1956).
- Although Chomsky's concept is widely used in linguistics, few natural languages have so far been described by generative grammars.

  The reason is that human communication contains ambiguities that can hardly be expressed formally and can often only be resolved with the knowledge of native speakers.

Influence on Computer Science

- For computer science, the concept of generative grammar has proven to be very useful due to its mathematical rigor (which was a major disadvantage in the description of natural languages).
- Such grammars allow the simple and clear specification of word sets, that are too complex for finite automata and regular expressions.
- Substitution rules (also named production rules) are used to derive words from a start symbol. This procedure is used in many programming and description languages.
  (Java, Python, HTML, XML,. . .)

Reminder: A formal language over Σ is a subset of Σ^*.

Example 4.1

We consider the following formal language over $\{a, b\}$.

$$\mathrm{PAL} = \{w \in \{a, b\}^* \mid w \text{ is symmetric}\}$$

PAL has a simple inductive description:

BC $\varepsilon, a, b \in \mathrm{PAL}$

IS If $w \in \mathrm{PAL}$, then also awa and bwb.

We can describe this equivalently using the following grammar.

$$\begin{array}{lll} S \to \varepsilon & S \to a & S \to b \\ S \to aSa & S \to bSb & \end{array}$$

Starting with S, the values can be generated from PAL by applying the production rules, e.g.,

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow ab\varepsilon ba = abba \quad \text{or}$$

$$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbabb.$$

The various ways of applying the rules lead to the various derivable words.

The grammar used in the example had the property that on the left hand side of each rule there was exactly one symbol.

This is not necessarily the case for general grammars, here also rules of the form $abRaSSb \to aaSaRRa$ are allowed.

Definition 4.2 (Generative grammar)

A **generative grammar** (grammar for short) is a quadruple $G = (\Sigma, N, S, R)$ with the following properties.

- Σ is a finite, non-empty set (*Terminal symbols*)
- N is a finite, non-empty set with $\Sigma \cap N = \emptyset$ (*non-terminal symbols*)
- $S \in N$ (*start symbol*)
- $R \subseteq (\Sigma \cup N)^+ \times (\Sigma \cup N)^*$ is a finite set (*set of production rules*)

  For $(v, w) \in R$ we also write $v \to w$.

Convention: Terminals = lower case letters
non-terminals = uppercase letters

Generating languages using generative grammars:

Definition 4.3

Let $G = (\Sigma, N, S, R)$ be a grammar, $v, w \in (\Sigma \cup N)^*$, $t \geq 0$.

- $\boldsymbol{v \underset{G}{\Rightarrow} w} \overset{df}{\Longleftrightarrow}$ there exist $u_1, u_2, x, y \in (\Sigma \cup N)^*$ with $(x, y) \in R$, $v = u_1 x u_2$ and $w = u_1 y u_2$ (G generates w from v in *one* step)
- $\boldsymbol{v \overset{t}{\underset{G}{\Rightarrow}} w} \overset{df}{\Longleftrightarrow}$ there exist $w_0, \ldots, w_t \in (\Sigma \cup N)^*$ with $v = w_0 \underset{G}{\Rightarrow} w_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_t = w$ (G generates w from v in t steps)
- $\boldsymbol{v \overset{*}{\underset{G}{\Rightarrow}} w} \overset{df}{\Longleftrightarrow} \exists t' \geq 0$ with $v \overset{t'}{\underset{G}{\Rightarrow}} w$ (G generates w from v)
- $\boldsymbol{L(G)} \overset{df}{=} \{z \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} z\}$ (the language generated by G)

Please note:

- $w \overset{*}{\underset{G}{\Rightarrow}} w$ holds for all $w \in (\Sigma \cup N)^*$.
- $L(G)$ consists only of terminal symbol words.
- Non-terminal symbols are auxiliary characters. They help generating words, but must be eliminated by the end of the generation of a word.

Example 4.4

We look again at the grammar from Example 4.1.

$$\begin{array}{lll} S \to \varepsilon & S \to a & S \to b \\ S \to aSa & S \to bSb & \end{array}$$

We notate this grammar according to Definition 4.2 as $G = (\Sigma, N, S, R)$ with $\Sigma = \{a, b\}$, $N = \{S\}$ and

$$\begin{aligned} R &= \{(S,\varepsilon),(S,a),(S,b),(S,aSa),(S,bSb)\} \\ &= \{S \to \varepsilon,\ S \to a,\ S \to b,\ S \to aSa,\ S \to bSb\}. \end{aligned}$$

Then $S \underset{G}{\Rightarrow} bSb \underset{G}{\Rightarrow} bbSbb \underset{G}{\Rightarrow} bbabb$ and therefore, $S \overset{3}{\underset{G}{\Rightarrow}} bbabb$ and $S \overset{*}{\underset{G}{\Rightarrow}} bbabb$. Furthermore, $L(G) = \mathrm{PAL}$.

Definition 4.5

Two grammars G and G' are **equivalent** $\overset{df}{\iff}$ $L(G) = L(G')$.

Definition 4.6 (Types of grammars)

Let $G = (\Sigma, N, S, R)$ be a grammar.

- G is **grammar of type 0**.
- G is **grammar of type 1** or **context-sensitive grammar** $\overset{df}{\Longleftrightarrow}$ each rule has the form $u_1 A u_2 \to u_1 w u_2$ with $A \in N$, $u_1, u_2, w \in (\Sigma \cup N)^*$ and $w \neq \varepsilon$.
  (replacement of A by w only in the context $u_1 A u_2$)
- G is called **grammar of type 2** or **context-free grammar** $\overset{df}{\Longleftrightarrow}$ each rule has the form $A \to w$ with $A \in N$, $w \in (\Sigma \cup N)^*$ and $w \neq \varepsilon$.
  (replacement of A by w without considering the context)
- G means **grammar of type 3** or **right-regular grammar** $\overset{df}{\Longleftrightarrow}$ each rule has the form $A \to aB$ or $A \to a$ with $A, B \in N$ and $a \in \Sigma$.

Remark 4.7

- *If G is of type i, it is also of type $i - 1$ (for $i = 1, 2, 3$).*
- *Rules of grammars of types 1, 2, and 3 never shorten a word (noncontracting grammars). Therefore, ε cannot be generated from the start symbol. However, in order to be able to describe languages with empty words using grammars, we add ε to the set of generated words if required (see Definition 4.9).*

Example 4.8 (Set of valid bracket words)

The set of correct bracket words such as

$$(),\ ()()(),\ ((((\,))((\,))))()$$

is generated by the following context-free grammar.

$$G = (\{(,)\},\ \{S\},\ S,\ \{S \to (),\ S \to (S),\ S \to SS\})$$

Definition 4.9 (Types of languages)

Let $L \subseteq \Sigma^*$ and $i \in \{0, 1, 2, 3\}$.

- L is called **language of type i** $\overset{df}{\iff}$ there exists a grammar G of type i with $L = L(G)$ or $L = L(G) \cup \{\varepsilon\}$.
- L is called **context-sensitive** $\overset{df}{\iff}$ L is of type 1.
- L is called **context-free** $\overset{df}{\iff}$ L is of type 2.
- $\boldsymbol{L}_i \overset{df}{=} \{L \mid L \text{ is of type } i\}$
- The **Chomsky hierarchy** consists of $\boldsymbol{L}_0$, $\boldsymbol{L}_1$, $\boldsymbol{L}_2$, $\boldsymbol{L}_3$.

Property 4.10

$\boldsymbol{L}_3 \subseteq \boldsymbol{L}_2 \subseteq \boldsymbol{L}_1 \subseteq \boldsymbol{L}_0$.

Proof.

Follows from Definition 4.6. □

We will see later that these inclusions are proper.

4.2
Languages of Type 3

Learning Objectives

After this section you should

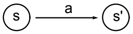
1. know the equivalence between regular languages and languages of type 3 and understand the idea of the proof.
2. be able to prove that the language of symmetric words is in $\boldsymbol{L}_2 - \boldsymbol{L}_3$ with the help of the Pumping Lemma.

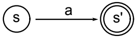
Theorem 4.11

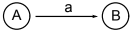
For $L \subseteq \Sigma^*$ the following statements are equivalent:

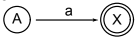
1. *L is regular.*
2. *L is of type 3.*

Proof idea:

- "$1 \Rightarrow 2$": 

is simulated by $s \to as'$


is simulated by $s \to as'$ and $s \to a$
- "$2 \Rightarrow 1$": $A \to aB$ is simulated by 

$A \to a$ is simulated by 

Without proof.

Corollary 4.12

There are non-regular context-free languages, i.e., $\boldsymbol{L}_3 \subsetneq \boldsymbol{L}_2$.

Proof.

The following context-free grammar generates $\mathrm{PAL} - \{\varepsilon\}$.

$$\begin{array}{lll} S \to a & S \to aa & S \to aSa \\ S \to b & S \to bb & S \to bSb \end{array}$$

This shows $\mathrm{PAL} \in \boldsymbol{L}_2$.

We prove $\mathrm{PAL} \notin \mathrm{REG}$ by proving that PAL is not 3-pumpable. So we have to prove: For all $k \in \mathbb{N}^+$ there exists a word $w \in \mathrm{PAL}$ with $|w| \geq k$ such that for every decomposition $w = xyz$ with $|xy| \leq k$ and $y \neq \varepsilon$ there exists some $i \geq 0$ so that $xy^iz \notin \mathrm{PAL}$.

Let $k \in \mathbb{N}^+$ be given. Choose $w \overset{df}{=} a^k b a^k \in \mathrm{PAL}$ (hence $|w| \geq k$). Let $w = xyz$ be a decomposition with $y \neq \varepsilon$ and $|xy| \leq k$.Then it follows from $|xy| \leq k$ that y is to the left of b in the decomposition. Then for $i = 0$ it holds $xy^0z = xz = a^{k-|y|}ba^k \notin \mathrm{PAL}$ (recall $|y| > 0$). □

Remark 4.13

Grammars of type 3 *may by definition not contain* ε*-rules (i.e. rules of the form* $A \to \varepsilon$*).*
However, it can be shown that the addition of ε*-rules does not lead out of the class of languages of type* 3 *(exercise).*

4.3
Context-Free Languages

Learning Objectives

After this section you should

1. be able to convert context-free grammars into Chomsky normal form.
2. be able to explain the CYK algorithm.
3. know the closure properties of the class of context-free languages.
4. know and be able to apply the Pumping Lemma for context-free languages (e.g. to show that $\{a^n b^n c^n \mid n \geq 1\}$ is not context-free).

By definition, context-free grammars must not contain any ε rules (i.e., rules of the form $A \to \varepsilon$).
We show that the addition of ε-rules does not lead out of the class of context-free languages.

Theorem 4.14

For $L \subseteq \Sigma^*$ the following statements are equivalent:

1. *L is context-free.*
2. *L is generated by a grammar $G = (\Sigma, N, S, R)$ with rules of the form $A \to w$, where $A \in N$ and $w \in (\Sigma \cup N)^*$.*

Proof idea:

- "1 ⇒ 2": easy. "2 ⇒ 1": For rules $A \to u_0 B_1 u_1 \cdots B_k u_k$ with $u_i \in (\Sigma \cup N)^*$, $B_i \in N$ and $B_i \underset{G}{\overset{*}{\Rightarrow}} \varepsilon$ we take the rule $A \to u_1 u_2 \cdots u_k$ to the grammar. This means that all ε rules can be removed.

Without proof.

4.3.1
The Chomsky Normal Form

We will now get to know a possibility, to create context-free languages using very simple rules.

This facilitates later proofs, as we can then restrict ourselves to very simple context-free grammars.

Definition 4.15 (Chomsky normal form)

A grammar $G = (\Sigma, N, S, R)$ is in **Chomsky normal form** $\overset{df}{\Longleftrightarrow}$ G has only rules of the form $A \to BC$ or $A \to a$ with $A, B, C \in N$ and $a \in \Sigma$.

Theorem 4.16

Every context-free grammar has an equivalent grammar in Chomsky normal form.

Proof idea:

- Step 1: Eliminate terminals on right-hand sides
- Step 2: Eliminate rules of the form $A \to B$
- Step 3: Eliminate rules of the form $A \to B_1 \cdots B_m$ with $m \geq 3$

Proof sketch.

Let $G = (\Sigma, N, S, R)$ be a context-free grammar. We construct an equivalent grammar in Chomsky normal form: (uppercase letters=non-terminals, lowercase letters=terminals)

Step 1: G_1 arises from G as follows:

- choose new non-terminal D_a for each $a \in \Sigma$
- replace a with D_a in all rules
- add new rule $D_a \to a$

Step 2: G_2 arises from G_1 as follows:

- if $A_1 \underset{G}{\Rightarrow} A_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} A_k \underset{G}{\Rightarrow} a$ with $k \geq 2$, then add $A_1 \to a$
- if $A_1 \underset{G}{\Rightarrow} A_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} A_k \underset{G}{\Rightarrow} B_1 \cdots B_m$ with $m, k \geq 2$, then add $A_1 \to B_1 \cdots B_m$
- remove all rules of the form $A \to B$

Step 3: G_3 arises from G_2 as follows:

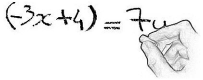
- replace $A \to B_1 \cdots B_m$ with $m \geq 3$ by $A \to B_1 E_2$, $E_2 \to B_2 E_3$, …, $E_{m-1} \to B_{m-1} B_m$ (where the E_i are new non-terminals)

G_3 is in Chomsky normal form and $L(G) = L(G_3)$. $\square$

Example 4.17 (Convert into Chomsky normal form)

We convert the following context-free grammar for PAL $-\{\varepsilon\}$ (cf. Corollary 4.12) into Chomsky normal form.

$$G = (\{a, b\}, \{S\}, S, R) \quad \text{with}$$
$$R = \{S \to aSa, S \to bSb, S \to aa, S \to bb, S \to a, S \to b\}$$



4.3.2
The CYK Algorithm

So far we have found out the following about the Chomsky hierarchy.

$$\mathrm{REG} = \boldsymbol{L}_3 \subsetneq \boldsymbol{L}_2 \subseteq \boldsymbol{L}_1 \subseteq \boldsymbol{L}_0$$

We now examine the complexity of context-free languages and show that every context-free language is decidable in time $O(n^3)$.

This shows that all programming or description languages with context-free syntax have efficient parsers.

In the proof we use the CYK algorithm named after its developers Cocke, Younger, and Kasami. The algorithm is based on *dynamic programming* (systematic solving of all subtasks and storing the results to avoid multiple computations of the same task).

Theorem 4.18 (CYK algorithm)

Every context-free language can be decided by a Python program in time $O(n^3)$.

Proof.

Let L be context-free, i.e., there is a $G = (\Sigma, N, S, R)$ in Chomsky normal form with $L(G) = L - \{\varepsilon\}$.

Let $w = a_0 \cdots a_{n-1}$ with $n \geq 1$ and $a_i \in \Sigma$ be given.
Question: Is $w \in L(G)$?

For this we define for all $A \in N$ and $0 \leq i \leq j < n$ the value

$$A(i,j) \overset{df}{=} \begin{cases} 1, & \text{if } A \underset{G}{\overset{*}{\Rightarrow}} a_i a_{i+1} \cdots a_j \\ 0 & \text{otherwise} \end{cases}$$

Thus: $\quad w \in L \;\Leftrightarrow\; S(0, n-1) = 1.$

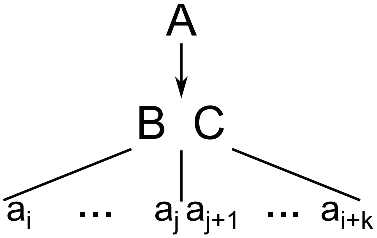
We are therefore interested in the value of $S(0, n-1)$.

In general, $A(i,j)$ can be computed using the following recursion:

BC $A(i,i) = 1 \quad \Leftrightarrow \quad A \rightarrow a_i$ is a rule.

IS For $k \geq 1$,

$A(i, i+k) = 1 \quad \Leftrightarrow \quad$ there is a rule $A \rightarrow BC$ and a j with $i \leq j < i+k$, such that $B(i,j) = 1$ and $C(j+1, i+k) = 1$.



The naive implementation of this recursion leads to a recursion tree of exponential size in n, since many $A(i, j)$ are computed multiple times.

Saving and reusing already computed $A(i, j)$ leads to a significantly faster decision algorithm for the language $L - \{\varepsilon\}$.

On input $w = a_0 \cdots a_{n-1}$, the following algorithm computes all values $A(i, i + k)$ successively for $k = 0, 1, \dots, n - 1$.

```
def CYK(w):                        # input w = [a0, a1, ..., a(n-1)]
  n = len(w)                       # length of w
  if n==0: return 0                # handle empty word

  # add following line for each non-terminal A
  A = [ n*[0] for i in range(0,n) ] # nxn matrix w. entries 0

  for i in range(0,n):
    # BC: A(i,i)=1  <=>  A->ai is a rule
    # add following line for each rule A->a
    if w[i]=='a': A[i][i]=1

  for k in range(1,n):             # k = 1, 2, ..., n-1
    for i in range(0,n-k):
      for j in range(i,i+k): # all i,j w. 1 <= i <= j < i+k <= n
        # IS: A(i,i+k)=1 <=> rule A->BC w. B(i,j)=C(j+1,i+k)=1
        # add following line for each rule A->BC
        if B[i][j]==1 and C[j+1][i+k]==1: A[i][i+k]=1

  return S[0][n-1]
```

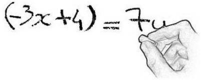
The $n \times n$ matrix A can be replaced by a list A' with the entries $A'[i * n + j] = A[i][j]$. The multiplication $i * n$ can be saved by carrying the values $i' = i \cdot n$ along. It is therefore possible to access elements of A in time $O(1)$.

Runtime: Generate the $n \times n$ matrices: $O(n^2)$
simple for-loop for BC: $O(n)$
triple for-loop for IS: $O(n^3)$

total $O(n^3)$, i.e. L is decidable in time $O(n^3)$ $\square$

Example 4.19 (Applying the CYK Algorithm)

We test whether the grammar $G = (\{a, b\}, \{S, A, B, C\}, S, R)$ with $R = \{S \to CA, A \to CA, B \to AC, C \to BA, A \to a, B \to b, C \to b\}$ generates the word *abbaa*.



4.3.3
The Pumping Lemma for Context-Free Languages

In Section 4.2 we have seen that the class of regular languages is a *proper* subset of the class of context-free languages.

We now want to show that the class of context-free languages is a *proper* subset of the class of context-sensitive languages. To do this, we need to find a language $L \in \boldsymbol{L}_1 - \boldsymbol{L}_2$.

For this purpose, we get to know the Pumping Lemma for context-free languages. It provides a general method with the help of which we can show for a given language that it is not context-free.

2-Pumpability

In the context of finite automata (languages of type 3) we encountered the property "3-pumpability".

Now, in the context of languages of type 2, we deal with a somewhat similar, but a bit more complex property: 2-pumpability.

Definition 4.20 (2-pumpable)

A language L is **2-pumpable** $\Leftrightarrow$ there exists $k \in \mathbb{N}^+$ (called **2-pumping number for L**) such that every word $w \in L$ with $|w| \geq k$ can be split into $w = rstuv$ such that

1. $su \neq \varepsilon$ (i.e., the pumping parts are not both empty)
2. $|stu| \leq k$ (i.e., the pumping parts are close to each other)
3. $\forall_{i \geq 0} rs^i tu^i v \in L$ (i.e., when pumping parts 2 and 4, the pumping parts, up/down, we stay in L)

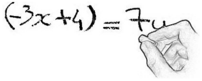
Lemma 4.21 (Pumping Lemma for CFL)

Let L be a language. If L is context-free, then L is 2-pumpable.

Proof idea:

- Let $G = (\Sigma, N, S, R)$ be a grammar in Chomsky normal form with $L = L(G)$
- Let $k = 2^{|N|}$
- Consider a generation tree of a word $w \in L$ with $|w| \geq k$
- It contains a long path on which a non-terminal occurs twice (pidgeon hole principle)
- The section between the duplicate occurrences can be executed multiple times or not at all
- This ensures that pumped words $rs^i tu^i v$ are $\in L$

Proof sketch.



□

Example 4.22 (Applying the Pumping Lemma)

We show that the following language is not context-free.

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

Proof.

Exercise. □

Corollary 4.23

There are context-sensitive languages that are not context-free, i.e., $\boldsymbol{L}_2 \subsetneq \boldsymbol{L}_1$.

Proof.

$L = \{a^n b^n c^n \mid n \geq 1\}$ is generated by

$$\begin{aligned} G &= (\{a, b, c\}, \{S, B\}, S, R) \quad \text{with} \\ R &= \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\}. \end{aligned}$$

We argue for $L \subseteq L(G)$: Consider $a^n b^n c^n$ for an arbitrary $n \in \mathbb{N}^+$.

- Applying $S \to aSBc$ for $n-1$ times yields $a^{n-1}S(Bc)^{n-1}$
- $S \to abc$ removes S and we obtain $a^{n-1}abc(Bc)^{n-1}$
- With $cB \to Bc$ we can move the B's left
- Applying $bB \to bb$ iteratively, we can convert all B's to b's and thus have generated $a^n b^n c^n$.

$$\begin{aligned} G &= (\{a, b, c\}, \{S, B\}, S, R) \quad \text{with} \\ R &= \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\}. \end{aligned}$$

Let's reason for $L \supseteq L(G)$. Let $w \in L(G)$ and consider an arbitrary generation of w in G.

- Let $n \in \mathbb{N}$ be the number of times the rule $S \to aSBc$ is applied
- $S \to abc$ is applied exactly once
- As the other two rules only change the order and replace B's with b's, the number of a's, b's, and c's in w are each $n + 1$
- A B can only be replaced with a terminal if it is moved to the left of all c's, so $w = a^{n+1}b^{n+1}c^{n+1} \in L$

Because of $cB \to Bc$, G is not context-sensitive. However, we can construct an equivalent context-sensitive grammar as follows.

1. Replace c everywhere with C and add $C \to c$
2. Replace $CB \to BC$ with $CB \to DB$, $DB \to DC$, $DC \to BC$

We obtain the following context-sensitive grammar for L.

$$\begin{aligned} G &= (\{a,b,c\},\{S,B,C,D\},S,R) \quad \text{with} \\ R &= \{S \to aSBC, S \to abC, CB \to DB, DB \to DC, \\ &\quad\; DC \to BC, bB \to bb, C \to c\}. \end{aligned}$$

This shows $L \in \boldsymbol{L}_1$. From Example 4.22 we know $L \notin \boldsymbol{L}_2$. □

Similarly as for regular languages: to prove a language L to be non-2-pumpable and thus non-context-free, it suffices to show the contraposition of the property “2-pumpable”, i.e.,

For all $k \in \mathbb{N}^+$ there exists $w \in \Sigma^{\geq n}$ such that for every decomposition $w = rstuv$ with $|stu| \leq$ and $su \neq \varepsilon$ there exists $i \geq 0$ with $rs^i tu^i v \notin L$.

We now consider a second example with an impact in practice.

Goal: For all $k \in \mathbb{N}^+$ there exists $w \in L^{\geq k}$ such that for all decompositions $w = rstuv$ with $|stu| \leq k$ and $su \neq \varepsilon$ there exists $i \in \mathbb{N}$ with $rs^i tu^i v \notin L$.

Example 4.24

$L = \{wcw \mid w \in \{a, b\}^*\} \subseteq \{a, b, c\}^*$ is not context-free.

Proof.

Let $k \in \mathbb{N}^+$. Choose $w = a^k b^k c a^k b^k \in L^{\geq k}$. Let $w = rstuv$ be a decomposition with $|stu| \leq k$ and $su \neq \varepsilon$. Choose $i = 0$.
a) If c is in su, the (pumped down) word rtv contains no c.
b) If c is in rv, the letter c is not in the middle of rtv.
c) If c is in t, we only pump in the red parts of $a^k b^k c a^k b^k$.
Thus $uwy \notin L$. □

Why We Want Context-Free Languages in Practice

As the CYK-algorithm demonstrates, all context-free languages can be decided **efficiently**.

We will later see that this probably does **not** hold for context-sensitive languages.

Hence, language processing tools are often designed for **context-free languages only** (e.g., **parser generators**)

So knowing whether a language is context-free or not is relevant in practice.

Is C Context-Free?

Most general purpose languages (e.g., Java, C, Python,. . .) are mainly context-free (indeed even deterministic context-free).

Thus, they can be dealt with algorithmically in a comfortable way. Handling the non-context-free parts is often done in a preprocessing step.

What about C? Is there a context-free grammar that generates all valid C programs?

Is C Context-Free? – Ctd

C is not context-free because it is not pumpable

Recall: $L = \{wcw \mid w \in \{a,b\}^*\} \subseteq \{a,b,c\}^* \notin \boldsymbol{L}_2$.
Witness: $a^k b^k c a^k b^k$

$$
\begin{array}{l}
\texttt{int main() \{} \\
\quad \texttt{int } \underbrace{\texttt{aa...a}}_{k}\underbrace{\texttt{bb...b}}_{k}\texttt{;} \\
\quad \underbrace{\texttt{aa...a}}_{k}\underbrace{\texttt{bb...b}}_{k}\texttt{++;} \\
\texttt{\}}
\end{array}
$$

$$\texttt{int main()\{int } \underbrace{\texttt{aa...abb...b}}_{w}\texttt{;}\underbrace{\texttt{aa...abb...b}}_{w}\texttt{++;\}}$$

Pumping down such programs does not lead to valid C programs

Trip to the Mountains

It is assumed that most natural languages have a context-free grammar.

However, some fancy languages can be “shown” not to be context-free.

Theorem 4.25

Swiss German is not context-free.

Proof sketch.

The reason is the structure of subclauses.

$$\text{...that we } \overbrace{\text{let}}^{1}\ \overbrace{\text{the children}}^{1}\ \overbrace{\text{help}}^{2}\ \overbrace{\text{Hans}}^{2}\ \overbrace{\text{to paint}}^{3}\ \overbrace{\text{the house}}^{3}.$$

Structure: $L = \{w_1 w_1 w_2 w_2 \dots w_n w_n \mid w_i \in \Sigma, i \in \mathbb{N}\} \in \boldsymbol{L}_3$

German:

$$\text{...dass wir } \underbrace{\text{die Kinder}}_{1}\ \underbrace{\text{dem Hans}}_{2}\ \underbrace{\text{das Haus}}_{3}\ \underbrace{\text{streichen}}_{3}\ \underbrace{\text{helfen}}_{2}\ \underbrace{\text{ließen}}_{1}.$$

Structure: $L = \{w_1 \dots w_n w_n \dots w_1 \mid w_i \in \Sigma, i \in \mathbb{N}\} \in \boldsymbol{L}_2$

Swiss German:

...that we the children Hans the house let help paint.

$$\text{...das mer } \underbrace{\text{d'chind}}_{1}\ \underbrace{\text{em Hans}}_{2}\ \underbrace{\text{es huus}}_{3}\ \underbrace{\text{lönd}}_{1}\ \underbrace{\text{hälfe}}_{2}\ \underbrace{\text{aastriiche}}_{3}.$$

Structure: $L = \{ww \mid w \in \Sigma^*\} \notin \boldsymbol{L}_2$ $\square$

Closure Properties of Context-Free Languages

In contrast to $\boldsymbol{L}_3$, $\boldsymbol{L}_2$ is not closed under intersection and complement.

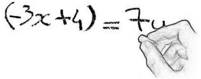
Theorem 4.26 (Closure properties of $\boldsymbol{L}_2$)

1. *$\boldsymbol{L}_2$ is closed under $\cup$, $\cdot$, $*$.*
2. *$\boldsymbol{L}_2$ is not closed under $\cap$ and complement.*

Proof idea:

- Closure under $\cup$, $\cdot$, $*$: with the help of grammars.
- No closure under $\cap$: with $T_1 = \{a^n b^n c^m \mid m, n \geq 1\}$ and $T_2 = \{a^n b^m c^m \mid m, n \geq 1\}$.
- No closure under complement: with De Morgan's rule.

Proof.



$\square$

4.4
Non-Determinism

Learning Objectives

After this section you should

1. understand the concepts of *determinism* and *non-determinism* for algorithms and machines.

We already know the difference between deterministic and non-deterministic finite automata. Both terms can also be applied to general algorithms and machines.

Conventional algorithms work **deterministically**. In contrast, **non-deterministic** algorithms and machines have a special form of parallelism:[15]

- Work can be divided among parallel computation paths
- The input is accepted if at least one computation path accepts (i.e. result 1); otherwise, the input is rejected

[15]The idea behind "nondeterministic" is that computation paths can be split for the purpose of parallelizing work and thus *it is not specified*, which branch of the computation is to be pursued from this point.

Non-Determinism for Turing Machines

Several rules for one situation:

$$\begin{array}{lcl} sa & \rightarrow & s'_1 a'_1 \sigma_1 \\ sa & \rightarrow & s'_2 a'_2 \sigma_2 \\ & \vdots & \\ sa & \rightarrow & s'_k a'_k \sigma_k \end{array}$$

Imagination: The current computation path splits into k parallel computation paths, whereby the instruction $sa \rightarrow s'_i a'_i \sigma_i$ is executed on the ith computation path.

This means that all k possible continuations of the computation are executed simultaneously.

Non-Determinism for RAMs

Multiple RAM instructions with the same number:

$$\begin{array}{ll} 5 & b_1 \\ 5 & b_2 \\ & \vdots \\ 5 & b_k \end{array}$$

Imagination: If [IR] $= 5$, the current computation path splits into k parallel computation paths, whereby the RAM instruction b_i is executed on the ith computation path.

This means that all k possible continuations of the computation are executed simultaneously.

Non-Determinism for While Programs

New type of statement block: If $a_1, \ldots, a_k$ are variables and $b_1, \ldots, b_k$ are expressions, the following is a statement block.

$$a_1 = b_1 \;\mid\; a_2 = b_2 \;\mid\; \cdots \;\mid\; a_k = b_k$$

Imagination: The current computation path splits into k parallel computation paths, whereby the instruction $a_i = b_i$ is executed on the ith computation path.

This means that all k possible continuations of the computation are executed simultaneously.

Example

```
def f(x):
  bl = binlength(x)
  divisor = 0
  p = 1
  for i in range(0, bl):
    divisor = divisor | divisor = (divisor + p)
    p = (p + p)
  if (((divisor >= x) or divisor < 2) or modZ(x, divisor) != 0):
    divisor = 0
  return divisor
```

Each computation path either returns 0 or a non-trivial divisor of x and runs in $O(n)$ (provided that efficient deterministic implementations of `binlength` and `modZ` are used.

Finding a non-trivial divisor of an integer is a complex task. E.g., the RSA cryptosystem becomes unsecure if an “efficient” (polynomial-time) algorithm for this problem is found.

Definition 4.27 (Acceptance of non-det. algorithms)

Let M be a non-deterministic algorithm.

- **M accepts x** $\overset{df}{\Longleftrightarrow}$ there is a computation path of M at input x that computes the value 1
- The **set accepted by M is**

$$\mathbf{L}(\mathbf{M}) \overset{df}{=} \{x \mid \text{M accepts x}\}.$$

Remark 4.28 (Acceptance vs decidability)

Acceptance of languages is a strictly weaker property than decidability. There is a TM M (even a deterministic one) with $L(M) = K_0' \overset{df}{=} \{bin(x) \mid x \in \mathbb{N}$ and M_x halts on input $x\}$: one can e.g. use the TM that on input $x \in \{0,1\}^*$ simulates the RAM $M_{bin^{-1}(x)}$ on input $bin^{-1}(x)$ and returns 1 in case the simulation stops. Nonetheless, K_0' is undecidable.

Languages that are accepted by non-deterministic Turing machines are also called **semi-decidable***.*

Remark 4.29 (Benefit of non-determinism)

- *Of course, there are no machines that can multiply during their work and thus gain more computation power.*
- *Non-deterministic machine models are nevertheless useful, as they can be used to precisely describe certain language classes, which has led to remarkable insights on these classes.*
  Examples: NFAs ($\boldsymbol{L}_3$), non-deterministic pushdown automata ($\boldsymbol{L}_2$, not contained in this lecture), linear bounded automata ($\boldsymbol{L}_1$), non-deterministic TMs ($\boldsymbol{L}_0$), non-deterministic polynomial-time Turing machines (the famous class NP*, studied in the module "Algorithms and data structures")*

4.5
Context-Sensitive Languages

We want to get an overview of different characterizations of the class of context-sensitive languages.
By definition, context-sensitive grammars consist only of rules of the form $u_1Au_2 \to u_1wu_2$ with $A \in N$, $u_1, u_2, w \in (\Sigma \cup N)^*$ and $w \neq \varepsilon$.

Definition 4.30 (Noncontracting grammars)

A grammar of type 0 is called **noncontracting**, if it only has rules of the form $v \to w$ with $|v| \leq |w|$.

Context-sensitive grammars are therefore noncontracting. We now show that for every noncontracting grammar there is an equivalent context-sensitive grammar.

Theorem 4.31

For $L \subseteq \Sigma^*$ the following statements are equivalent:

1. *L can be generated by a context-sensitive grammar.*
2. *L can be generated by a noncontracting grammar.*

Proof.

"$1 \Rightarrow 2$": Holds by the definition of context-sensitive grammars.

"$2 \Rightarrow 1$": Let $G = (\Sigma, N, S, R)$ be a noncontracting grammar. We construct an equivalent context-sensitive grammar G''.

Choose a new non-terminal $E_a \notin N$ for each $a \in \Sigma$.

$R' \overset{df}{=}$ rule set that arises from R as follows:

- in all rules, each $a \in \Sigma$ is replaced by E_a
- for each $a \in \Sigma$ the rule $E_a \to a$ is added

Then $L(G) = L(G')$ for $G' \overset{df}{=} (\Sigma, N', S, R')$ w. $N' \overset{df}{=} N \cup \{E_a \mid a \in \Sigma\}$

Non-context-sensitive rules in R' have the form $B_1 B_2 \cdots B_r \rightarrow C_1 C_2 \cdots C_s$ with $2 \leq r \leq s$ and $B_1, \ldots, B_r, C_1, \ldots, C_s \in N$.

Such a rule can be replaced by context-sensitive rules $v_i \rightarrow v_{i+1}$, where the v_i are defined as follows:

$$\begin{array}{rcl}
v_0 & = & B_1 B_2 B_3 \cdots B_{r-1} B_r \\
v_1 & = & D_1 B_2 B_3 \cdots B_{r-1} B_r \\
v_2 & = & D_1 D_2 B_3 \cdots B_{r-1} B_r \\
 & \vdots & \\
v_r & = & D_1 D_2 D_3 \cdots D_{r-1} D_r \\
v_{r+1} & = & C_1 D_2 D_3 \cdots D_{r-1} D_r \\
v_{r+2} & = & C_1 C_2 D_3 \cdots D_{r-1} D_r \\
 & \vdots & \\
v_{2r-1} & = & C_1 C_2 C_3 \cdots C_{r-1} D_r \\
v_{2r} & = & C_1 C_2 C_3 \cdots C_{r-1} C_r C_{r+1} \cdots C_s
\end{array}$$

It can be shown that the rules are only applicable in the specified order (otherwise non-terminals D_i remain).

This provides a context-sensitive grammar G'' with $L(G'') = L(G)$. $\square$

Example 4.32 (Noncontracting $\rightarrow$ context-sensitive)

Consider the noncontracting grammar $G = (\{a\}, N, S, R)$ with $N = \{S, T, B, M, H, A\}$ and

$$R = \{S \rightarrow MS, S \rightarrow MBT, T \rightarrow BT, T \rightarrow H, MB \rightarrow ABM \\ MH \rightarrow Ha, MA \rightarrow AM, A \rightarrow a, B \rightarrow a, H \rightarrow a\}.$$

Step 1: Replace terminals with non-terminals.

$G' = (\{a\}, N', S, R')$ with $N' = N \cup \{E_a\}$ and

$$R = \{S \rightarrow MS, S \rightarrow MBT, T \rightarrow BT, T \rightarrow H, MB \rightarrow ABM \\ MH \rightarrow HE_a, MA \rightarrow AM, A \rightarrow E_a, B \rightarrow E_a, H \rightarrow E_a, E_a \rightarrow a\}.$$

Step 2: Replace non-context-sensitive rules.

$MB \rightarrow ABM$: $MB \rightarrow D_1 B, D_1 B \rightarrow D_1 D_2, D_1 D_2 \rightarrow AD_2, AD_2 \rightarrow ABM$
$MH \rightarrow HE_a$: $MH \rightarrow D_3 H, D_3 H \rightarrow D_3 D_4, D_3 D_4 \rightarrow HD_4, HD_4 \rightarrow HE_a$
$MA \rightarrow AM$: $MA \rightarrow D_5 A, D_5 A \rightarrow D_5 D_6, D_5 D_6 \rightarrow AD_6, AD_6 \rightarrow AM$

So we obtain $G'' = (\{a\}, \{S, T, B, M, H, A, E_a, D_1, D_2, D_3, D_4, D_5, D_6\}, S, R'')$ with

$$\begin{aligned} R'' = \{ & S \rightarrow MS, S \rightarrow MBT, T \rightarrow BT, T \rightarrow H, \\ & B \rightarrow E_a, H \rightarrow E_a, A \rightarrow E_a, E_a \rightarrow a, \\ & MB \rightarrow D_1 B, D_1 B \rightarrow D_1 D_2, D_1 D_2 \rightarrow AD_2, AD_2 \rightarrow ABM, \\ & MH \rightarrow D_3 H, D_3 H \rightarrow D_3 D_4, D_3 D_4 \rightarrow HD_4, HD_4 \rightarrow HE_a \\ & MA \rightarrow D_5 A, D_5 A \rightarrow D_5 D_6, D_5 D_6 \rightarrow AD_6, AD_6 \rightarrow AM \}. \end{aligned}$$

It holds $L(G) = L(G'')$.

Careful: The allegedly simpler replacement of $MB \rightarrow ABM$ with the rules $MB \rightarrow MD, MD \rightarrow AD, AD \rightarrow ABM$ leads to a non-equivalent grammar that generates $a^5 \notin L(G)$:

$$S \overset{*}{\underset{G}{\Rightarrow}} MMBH \overset{*}{\underset{G}{\Rightarrow}} MADH \overset{*}{\underset{G}{\Rightarrow}} AMDH \overset{*}{\underset{G}{\Rightarrow}} AABMH \Rightarrow AABHa \overset{*}{\underset{G}{\Rightarrow}} a^5$$

The class of context-sensitive languages can be characterized by non-deterministic machines.

Definition 4.33

A **linear bounded automaton** (**LBA**) is a non-deterministic 1-tape TM, which, during its work, only uses the cells occupied by the input word and their neighboring cells.

Theorem 4.34 (Kuroda's theorem)

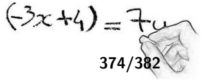
For $L \subseteq \Sigma^*$ the following statements are equivalent:

1. *L is context-sensitive.*
2. *There is an LBA that accepts L.*

Proof idea:

- "$1 \rightarrow 2$": LBA guesses on input w the production $S \overset{*}{\underset{G}{\Rightarrow}} w$ backwards
- "$1 \Leftarrow 2$": Construct the following noncontracting grammar:
- Non-terminals with 2 components: Component 1 simulates computation of the LBA, Component 2 stores input w
- Start symbol generates a word w in component 2 and in component 1 the initial situation of the LBA with input w
- Grammar rules simulate the computation of the LBA in component 1. If the LBA accepts, the input word w stored in component 2 is generated
- The grammar thus generates all words accepted by the LBA

Proof.



□

Corollary 4.35

Every context-sensitive language is decidable in time $2^{O(n)}$.

Proof idea:

- On input x, the LBA has a maximum of $2^{O(|x|)}$ configurations (due to restriction to input and its neighboring cells)
- Iteratively determine the set of configurations reachable from the start configuration and test whether there is an accepting configuration among them.

Without proof.

Theorem 4.36 (Closure properties of the class $\boldsymbol{L}_1$)

$\boldsymbol{L}_1$ is closed under $\cup$, $\cap$, $\cdot$, $*$, and complement.

Proof idea:

- $\cup$, $\cap$, $\cdot$, $*$: simple simulations of LBAs
- The closure under complement was a long-standing open problem with a surprising solution.[16]

Without proof.

[16] See this blog post for some interesting background on the problem and its solution.

4.6
Languages of Type 0

Type 0 languages can be characterized by non-deterministic Turing machines (without time and space constraints).

Theorem 4.37 (Kuroda's theorem)

For $L \subseteq \Sigma^*$ and $k \geq 1$ are equivalent to the following statements:

1. *L is of type 0.*
2. *L is accepted by a non-deterministic k-TM.*

Proof sketch.

Can be shown as in the proof of Theorem 4.34 $\square$

Theorem 4.38 (Closure properties of the class $\boldsymbol{L_0}$)

- $\boldsymbol{L}_0$ *is closed under* $\cup$, $\cap$, $\cdot$ *and* $*$.
- $\boldsymbol{L}_0$ *is not closed under complement.*

Without proof.

Remark 4.39

As explained in Remark 4.28, the set $K_0' = \{bin(x) \mid x \in \mathbb{N}$ *and* M_x *halts on input* $x\}$ *is accepted by a TM. However, it can be shown that there does not exist a non-deterministic TM that accepts* $\overline{K_0'}$.
Moreover, as K_0' *is undecidable, Theorem 4.35 shows that* $\overline{K_0'}$ *is not in* $\boldsymbol{L}_1$.

Thus, due to K_0' *the class* $\boldsymbol{L}_0$ *is not closed under complement and* $\boldsymbol{L}_1$ *is a proper subset of* $\boldsymbol{L}_0$.

4.7 Summary

Summary

L_0 = semi-decidable languages over Σ
(NTM = (non-)deterministic TM)

proper due to K'_0

L_1 = context-sensitive languages
(LBA = linear bounded automaton)

proper due to $\{a^n b^n c^n\}$

L_2 = context-free languages
(pushdown automaton)

proper due to $\{a^n b^n\}$

L_3 = FA = REG = regular languages
(DFA, NFA)

Closure properties

	$\cup$	$\cap$	$\overline{}$	$\cdot$	$*$
L_0	$+$	$+$	$-$	$+$	$+$
L_1	$+$	$+$	$+$	$+$	$+$
L_2	$+$	$-$	$-$	$+$	$+$
L_3	$+$	$+$	$+$	$+$	$+$

:wq