A **computable function** is a function

\[ f : A \rightarrow B \]

such that there is a *mechanical procedure* for computing for every \( a \in A \) the result \( f(a) \in B \).

In this part (IV) we study the **limits of computation**.

The area studying computable functions and their limits is called **computability theory**.
Examples

Define \( \exp : \mathbb{N} \to \mathbb{N} \), \( \exp(n) := 2^n \),
where \( \mathbb{N} = \{0, 1, 2, \ldots\} \).
\( \exp \) is computable.
However, can we really compute \( \exp(1000000000000000000000000000000000) \)?

Let \( \text{String} \) be the set of strings of ASCII symbols.
Define a function \( \text{check} : \text{String} \to \{\text{true}, \text{false}\} \) by

\[
\text{check}(p) := \begin{cases} 
\text{true} & \text{if } p \text{ is a syntactically correct Java program}, \\
\text{false} & \text{otherwise}.
\end{cases}
\]

Is \( \text{check} \) computable or not?
Define a function \texttt{terminate} : String \rightarrow \{\text{true, false}\},

\[
\text{terminate}(p) := \begin{cases} 
\text{true} & \text{if } p \text{ is a syntactically correct} \\
& \text{Java program with no input and outputs, which terminates;} \\
\text{false} & \text{otherwise.}
\end{cases}
\]

Is \text{terminate} \texttt{computable}?
Answer

__(To be filled in during the lecture)__
Define a function \( \text{is\_sorting\_fun} : \text{String} \to \{\text{true, false}\} \),

\[
\text{is\_sorting\_fun}(p) := \begin{cases} 
\text{true} & \text{if } p \text{ is a syntactically correct} \\
\text{false} & \text{otherwise.}
\end{cases}
\]

Is is\_sorting\_fun \emph{computable}?
Assume is\_sorting\_fun were computable.

Then we can construct (compute) a program which computes terminate as follows:

- Assume as input a string $p$.
- Check whether it is a syntactically correct Java program with no input and outputs.
- If no, $\text{terminate}(p) = \text{false}$, so return false.
- Otherwise, create from $p$ a program $q(p)$ which is a potential sorting function as follows:
  - $q(p)$ takes as input a list $l$.
  - Then it runs $p$.
  - If $p$ has terminated, then it runs a known sorting function on $l$, and returns the result.
Explanation

- If $p$ terminates, then $q(p)$ will be a sorting function, so
  \(\text{is\_sorting\_fun}(q(p)) = \text{true} = \text{terminate}(p)\).
- If $p$ does not terminate, then $q(p)$ does not terminate on any input, so
  \(\text{is\_sorting\_fun}(q(p)) = \text{false} = \text{terminate}(p)\).
- Our program returns now \(\text{is\_sorting\_fun}(q(p))\) which is the result of
  \(\text{terminate}(p)\).

- So we have obtained by using a program for \(\text{is\_sorting\_fun}\) a program
  which computes \(\text{terminate}\).

- But \(\text{terminate}\) is non-computable, therefore \(\text{is\_sorting\_fun}\) cannot be
  computable.
Problems in Computability

In order to understand and answer the questions we have to

- Give a precise definition of what *computable* means.
  - That will be a *mathematical definition*.
  - Such a notion is particularly important for showing that certain functions are *non-computable*.
- Then provide evidence that the definition of “*computable*” is the correct one.
  - That will be a *philosophical argument*.
- Develop methods for proving that certain functions are *computable or non-computable*.
Three areas are involved in computability theory.

- **Mathematics.**
  - Precise definition of computability.
  - Analysis of the concept.

- **Philosophy.**
  - Validation that notions found are the correct ones.

- **Computer science.**
  - Study of relationship between these concepts and computing in the real world.
Questions Related to The Above

- Given a function $f : A \rightarrow B$, which can be computed, can it be done \textit{effectively}?

  (Complexity theory.)

- Can the task of deciding a given problem $P_1$ be reduced to deciding another problem $P_2$?

  (Reducibility theory.)
More Advanced Questions

The following is beyond the scope of this module.

- Can the notion of *computability* be extended to computations on *infinite objects* (e.g. streams of data, real numbers, higher type operations)? *(Higher and abstract computability theory).*

- What is the relationship between *computing* (producing actions, data etc.) and *proving*? *(Research by U. Berger and M. Seisenberger).*
Idealisation

In computability theory, one usually abstracts from limitations on

- time and
- space.

A problem will be computable, if it can be solved on an *idealised computer*, even if it the computation would take longer than the life time of the universe.
Remark on Variables

- In this lecture I will often use $i, j, k, l, m, n$ for variables denoting natural numbers.
- I will often use $p, q$ and some others for variables denoting programs.
- I will use $z$ for integers.
- Other letters might be used as well for variables.
- These conventions are not treated very strictly.
  - Especially when running out of letters.
Gottfried Wilhelm von Leibnitz (1646 – 1716)

- Built a first *mechanical calculator*.
- Was thinking about a machine for manipulating symbols in order to determine truth values of mathematical statements.
- Noticed that this requires the definition of a precise *formal language*. 
David Hilbert (1862 – 1943)

- Poses 1900 in his famous list “Mathematical Problems” as 10th problem to decide Diophantine equations.
Hilbert (1928)

- Poses the *Entscheidungsproblem* (German for decision problem).
- The *decision problem* is the question, whether we can decide whether a formula in predicate logic is provable or not.
  - *Predicate logic* is the standard formalisation of logic with connectives $\land, \lor, \rightarrow, \neg$ and quantifiers $\forall, \exists$.
  - Predicate logic is “*sound and complete*”.
    - This means that a formula is provable if and only if it is valid (in all models).
So the decidability of predicate logic is the question whether we can decide whether a formula is valid (in all models) or not.

If predicate logic were decidable, provability in mathematics would become trivial.

“Entscheidungsproblem” became one of the few German words which have entered the English language.
History of Computability Theory

- Gödel, Kleene, Post, Turing (1930s)
  Introduce different *models of computation* and prove that they all define the same class of computable functions.
Kurt Gödel (1906 – 1978)
Introduced the (Herbrand-Gödel-)
recursive functions
in his 1933 - 34 Princeton lectures.
History of Computability Theory

Stephen Cole Kleene
(1909 – 1994)
Probably the most influential
computability theorist up to now.
Introduced the partial recursive
functions.

CS_236
Chapt. IV.1
21/ 40
History of Computability Theory

Emil Post
(1897 – 1954)
Introduced the Post problems.
History of Computability Theory

Alan Mathison Turing (1912 – 1954)

Introduced the Turing machine.
Proved the undecidability of the Turing-Halting problem.
Gödel’s Incompleteness Theorem

- Gödel (1931) proves in his first incompleteness theorem:
  - Every reasonable primitive-recursive theory is incomplete, i.e. there is a formula s.t. neither the formula nor its negation is provable.
  - The theorem generalises to recursive i.e. computable theories.
  - The notions “primitive-recursive” and “recursive” are introduced in an advanced module on computability theory.
    For the moment it suffices to understand “recursive” informally as intuitively computable.
Therefore no computable theory proves all true formulae.
Therefore, it is undecidable whether a formula is true or not.

Otherwise, the theory consisting of all true formulae would be a complete computable theory.
Church and Turing (1936) postulate that the models of computation established above define exactly the set of all computable functions (Church-Turing thesis, sometimes called Church’s thesis).

Both established independently undecidable problems and proved that the decision problem is undecidable, i.e. unsolvable.

Even for a class of very simple formulae we cannot decide the decision problem.
Undecidability of the Decision Problem

- Church shows the undecidability of equality in the $\lambda$-calculus.
- Turing shows the unsolvability of the **halting problem**.
  - It is undecidable whether a Turing machine (and by the Church-Turing thesis equivalently any non-interactive computer program) eventually stops.
  - That problem turns out to be the most important undecidable problem.
The undecidability of the Turing Halting Problem has similarities with limits in physics such as
- Speed limited by speed of light.
- Heisenberg’s uncertainty principle.

It limits what human beings can do
- Even with the most advanced computers we cannot decide the Turing halting problem.
History of Computability Theory

Alonzo Church (1903 - 1995)
History of Computability Theory

- Post (1944) studies degrees of unsolvability. This is the birth of degree theory.
- In degree theory one divides problems into groups ("degrees") of problems, which are reducible to each other.
  - Reducible means essentially “relative computable”.
- Degrees can be ordered by using reducibility as ordering.
- The question in degree theory is: what is the structure of degrees?
Yuri Vladimirovich Matiyasevich (∗ 1947)

- Solves 1970 Hilbert’s 10th problem negatively: The solvability of Diophantine equations is undecidable.
History of Computability Theory

Stephen Cook (Toronto)

- Cook (1971) introduces the complexity classes $P$ and $NP$ and formulates the problem, whether $P \neq NP$. 
Current State

- The problem $P \neq NP$ is still open. Complexity theory has become a big research area.
- Intensive study of computability on infinite objects (e.g. real numbers, higher type functionals) is carried out (e.g. U. Berger, J. Blanck and J. Tucker in Swansea).
- Computability on inductive and co-inductive data types is studied (e.g. A. Setzer, U. Berger).
- Research on program synthesis from formal proofs (e.g. U. Berger and M. Seisenberger in Swansea).
Current State

- Concurrent and game-theoretic models of computation are developed (e.g. Prof. Moller in Swansea).
- Automata theory further developed.
- Alternative models of computation are studied (quantum computing, genetic algorithms).
- ...
The original name was *recursion theory*, since the mathematical concept claimed to cover exactly the computable functions is called “recursive function”.

This name was changed to *computability theory* during the last 10 years.

Many books still have the title “recursion theory”.
Overview over Part IV

1. Introduction.
2. The Unlimited Register Machine (URM)
3. Turing machines and. the undecidability of the halting problem.
Literature

  - Main text book.
Literature

  - Criticized in Amazon Reviews. But several editions.

  - Excellent book, mainly on automata theory context free grammars.
  - But covers Turing machines, decidability questions as well.
Literature

  - Book on basic mathematics.
  - Useful if you need to fresh up your mathematical knowledge.
  - Expensive. Postgraduate level.