Mathematical Foundations of Computer Science

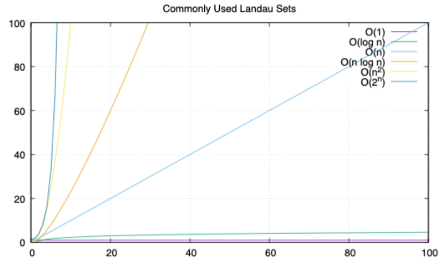
Lecture Notes

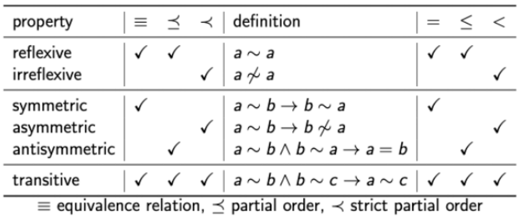
Jürgen Schönwälder

August 12, 2025

Abstract

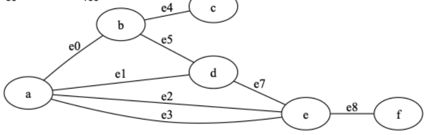
These lecture notes provide a basic summary of the module "Mathematical Foundations of Computer Science" offered at Constructor University (formerly Jacobs University Bremen). Topics addressed are algorithms, properties of algorithms, foundations of discrete mathematics, data representation, boolean algebra, propositional and predicate logic, basics of abstract algebra, foundations of graph theory, and software correctness. The course assumes that students have a general understanding of mathematical concepts at the level provided by high school education. The material does not require any programming competence but some programming knowledge might be helpful to understand how certain mathematical concepts relate to programming.





property	≡	⪯	≺	definition	=	≤	<
reflexive	✓	✓		$a \sim a$	✓	✓	
irreflexive			✓	$a \not\sim a$			✓
symmetric	✓			$a \sim b \to b \sim a$	✓		
asymmetric			✓	$a \sim b \to b \not\sim a$			✓
antisymmetric		✓		$a \sim b \land b \sim a \to a = b$		✓	
transitive	✓	✓	✓	$a \sim b \land b \sim c \to a \sim c$	✓	✓	✓

≡ equivalence relation, ⪯ partial order, ≺ strict partial order



$$\{P\}\ C\ \{Q\}$$

```
function COLLATZ($X$)
    while $X > 1$ do
        if $(X\%2) \neq 0$ then
            $X \leftarrow (3 \cdot X) + 1$
        else
            $X \leftarrow X/2$
        end if
    end while
    return $X$
end function
```

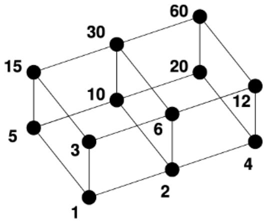
P1 $0 \in \mathbb{N}$

P2 $\forall n \in \mathbb{N}.s(n) \in \mathbb{N} \land n \neq s(n)$

P3 $\neg(\exists n \in \mathbb{N}.0 = s(n))$

P4 $\forall n \in \mathbb{N}.\forall m \in \mathbb{N}.s(n) = s(m) \Rightarrow n = m$

P5 $\forall P.(P(0) \land (\forall n \in \mathbb{N}.P(n) \Rightarrow P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}.P$



https://cnds.constructor.university/courses/mfcs-2025

Table of Contents

Part I

Algorithms, Complexity, Correctness

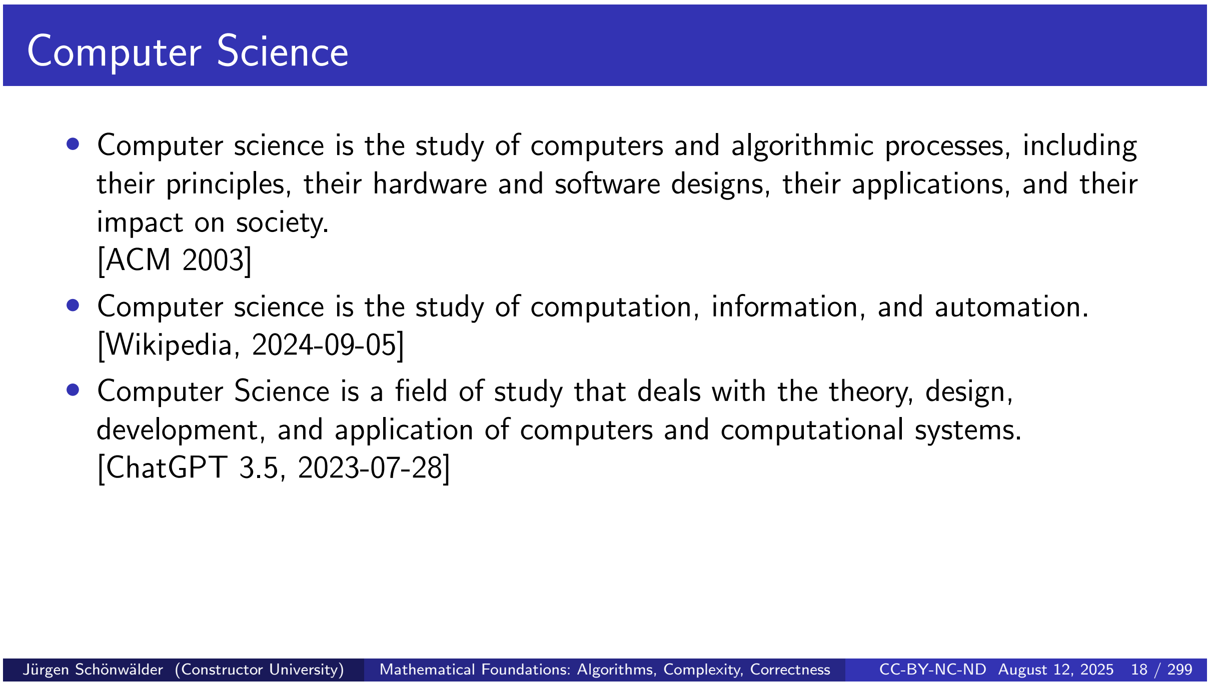
In this part we explore what computer science is all about, namely the notion of algorithms. After introducing the terms *algorithm* and *algorithmic thinking*, we will study two typical problems, namely the creation of mazes and the search of a pattern in a string. We will demonstrate that it is useful to look at a problem from different perspectives in order to find efficient algorithms to solve the problem.

By the end of this part, students should be able to

- explain what an algorithm and algorithmic thinking is;
- understand the importance of precise problem formalization;
- use mathematical notations for describing graphs;
- understand the difference between an abstract object and its representations;
- execute Kruskal's algorithm to calculate (minimum) spanning trees;
- describe why the Boyer-Moore algorithm outperforms a naive string search algorithm;
- execute the Boyer-Moore algorithm to find strings in a text;
- explain time and space complexity of an algorithm;
- understand Landau sets and the big O notation;
- define partial and total correctness of an algorithm.

Section 1: Computer Science and Algorithms

1. Computer Science and Algorithms
2. Maze Generation Algorithms
3. String Search Algorithms
4. Complexity and Correctness



Computer Science

- Computer science is the study of computers and algorithmic processes, including their principles, their hardware and software designs, their applications, and their impact on society.
  [ACM 2003]
- Computer science is the study of computation, information, and automation.
  [Wikipedia, 2024-09-05]
- Computer Science is a field of study that deals with the theory, design, development, and application of computers and computational systems.
  [ChatGPT 3.5, 2023-07-28]

James W. McGuffee presents different definitions of Computer Science, and he investigates how they are perceived by students [17]. Peter J. Denning et al. discuss misconceptions about Computer Science [4]. Some typical misconceptions are:

1. *Computer Science equals programming.*

   While computing professionals are expected to be able to program, they are also expected to do many things that are not programming.

2. *Once you master variables, sequencing, conditionals, loops, abstraction, modularization, and decomposition, you are a computing professional.*

   This again reduces computer science to programming concepts.

3. *Programming is easy to learn.*

   Programming is a skill set and programmers progress from beginners to experts over a long period of time.

4. *Computer Science is basically science and math.*

   While science and math are key enablers, computer science also is driven by engineering.

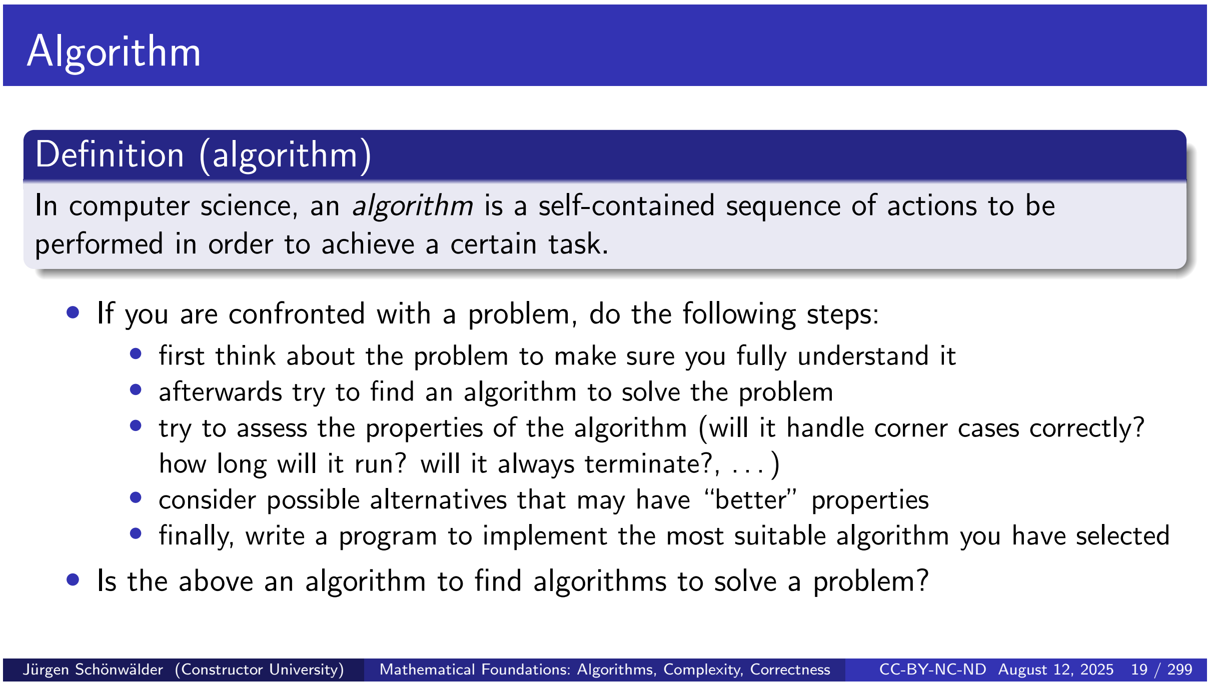
5. *Old Computer Science is obsolete. The important developments in Computer Science are Artificial Intelligence.*

   Artificial Intelligence is an old idea of computer science, recent advances in machine learning are enabled by the availability of huge datasets and large amounts of processing power, which are the results of decades of continuous improvements of “old” computer science.

Perhaps the most classic and most comprehensive book series presenting core algorithms and concepts of Computer Science is the series “The Art of Computer Programming’ by Donald E. Knuth [11, 14, 12, 13].

Further online information:

- **Wikipedia**: Computer Science
- **YouTube**: Map of Computer Science



Algorithm

Definition (algorithm)

In computer science, an *algorithm* is a self-contained sequence of actions to be performed in order to achieve a certain task.

- If you are confronted with a problem, do the following steps:
  - first think about the problem to make sure you fully understand it
  - afterwards try to find an algorithm to solve the problem
  - try to assess the properties of the algorithm (will it handle corner cases correctly? how long will it run? will it always terminate?, . . .)
  - consider possible alternatives that may have "better" properties
  - finally, write a program to implement the most suitable algorithm you have selected
- Is the above an algorithm to find algorithms to solve a problem?

The notion of an algorithm is central to computer science. Computer science is all about algorithms. A program is an implementation of an algorithm. While programs are practically important (since you can execute them), we usually focus on the algorithms and their properties and less on the concrete implementations of algorithms.

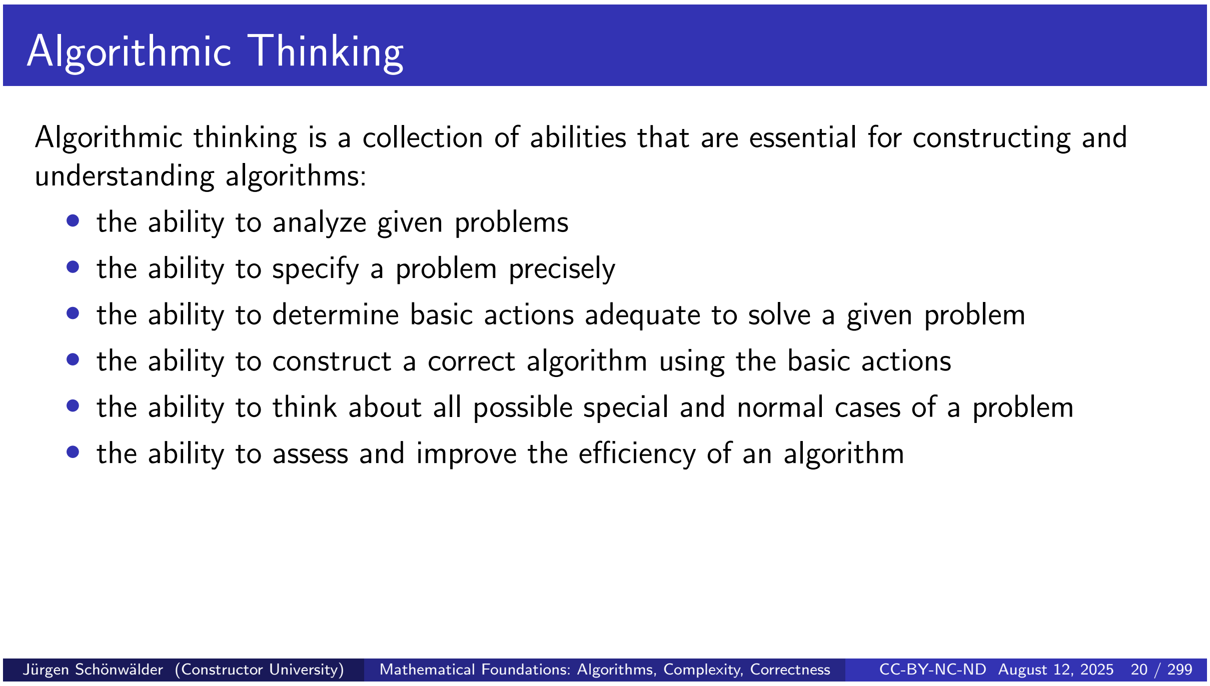
Another important aspect of computer science is the definition of abstractions that allow us to describe and implement algorithms efficiently. A good education in computer science will (i) strengthen your abstract thinking skills and (ii) train you in algorithmic thinking.

Some algorithms are extremely old. Algorithms were used in ancient Greek, for example the Euclidean algorithm to find the greatest common divisor of two numbers. Marks on sticks were used before Roman numerals were invented. Later in the 11th century, Hindu–Arabic numerals were introduced into Europe that we still use today.

The word *algorithm* goes back to Muhammad ibn Musa al-Khwarizmi, a Persian mathematician, who wrote a document in Arabic language that got translated into Latin as "Algoritmi de numero Indorum". The Latin word was later altered to algorithmus, leading to the corresponding English term 'algorithm'.

Further online information:

- **Wikipedia**: Algorithm



Algorithmic Thinking

Algorithmic thinking is a collection of abilities that are essential for constructing and understanding algorithms:

- the ability to analyze given problems
- the ability to specify a problem precisely
- the ability to determine basic actions adequate to solve a given problem
- the ability to construct a correct algorithm using the basic actions
- the ability to think about all possible special and normal cases of a problem
- the ability to assess and improve the efficiency of an algorithm

We will train you in algorithmic thinking [6]. This is going to change how you look at the world. You will start to enjoy (hopefully) the beauty of well designed abstract theories and elegant algorithms. You will start to appreciate systems that have a clean and pure logical structure.

But beware that the real world is to a large extend not based on pure concepts. Human natural language is very imprecise, sentences often have different interpretations in different contexts, and the real meaning of a statement often requires to know who made the statement and in which context. Making computers comprehend natural language is still a hard problem to be solved.

Example: Consider the following problem: Implement a function that returns the square root of a number (on a system that does not have a math library). At first sight, this looks like a reasonably clear definition of the problem. However, on second thought, we discover a number of questions that need further clarification.

- What is the input domain of the function? Is the function defined for natural numbers, integer numbers, real numbers (or an approximate representation of real numbers), complex numbers?
- Are we expected to always return the principal square root, i.e., the positive square root?
- What happens if the function is called with a negative number? Shall we return a complex number or indicate a runtime exception? In the later case, how exactly is the runtime exception signaled?
- In general, square roots can not be calculated and represented precisely (recall that $\sqrt{2}$ is irrational). Hence, what is the precision that needs to be achieved?

While thinking about a problem, it is generally useful to go through a number of examples. The examples should cover regular cases and corner cases. It is useful to write the examples down since they may serve later as test cases for an implementation of an algorithm that has been selected to solve the problem.

Further online information:

- **YouTube**: The Essentials of Problem Solving

Section 2: Maze Generation Algorithms

1. Computer Science and Algorithms

2. Maze Generation Algorithms

3. String Search Algorithms

4. Complexity and Correctness

Maze (33 x 11)

```
[]    [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
[]    []                            []                []                            []          []
[]    []    [] [] [] [] [] [] []    []    [] [] []    [] [] [] [] []    [] [] []    []    []    []
[]          []                []    []    []    []                []          []          []    []
[] [] [] [] [] [] []    []    []    []    []    [] [] [] [] []    []    []    [] [] [] [] []    []
[]          []          []    []          []    []          []    []    []    []    []          []
[]    []    []    [] [] []    [] [] [] [] []    []    []    []    [] [] []    []    []    [] [] []
[]    []          []                            []    []          []          []    []    []    []
[]    [] [] [] [] [] [] [] [] [] [] [] [] [] [] []    [] [] [] [] []    [] [] []    []    []    []
[]                                                                      []                      []
[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []    []
```

This is a simple 33x11 maze. How do you find a path through the maze from the entry (left top) to the exit (bottom right)?

We are not going to explore maze solving algorithms here. Instead we look at the problem of generating mazes.

Further online information:

- **Wikipedia**: Maze Solving Algorithms
- **YouTube**: Maze Solving - Computerphile

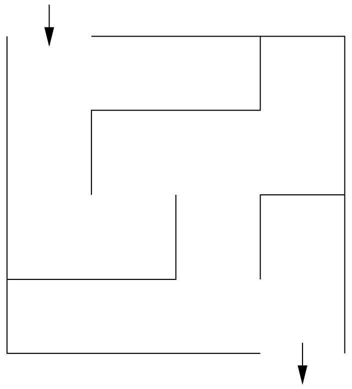
Problem Statement

Problem:

- Write a program to generate mazes.
- Every maze should be solvable, i.e., it should have a path from the entrance to the exit.
- We want maze solutions to be unique.
- We want every “room” to be reachable.

Questions:

- How do we approach this problem?
- Are there other properties that make a maze a “good” or a “challenging” maze?



It is quite common that problem statements are not very precise. A customer might ask for “good” mazes or “challenging” mazes or mazes with a certain “difficulty level” without being able to say precisely what this means. As a computer scientist, we appreciate well defined requirements but what we usually get is imprecise and leaves room for interpretation.

What still too often happens is that the computing professional discovers that the problem is underspecified and then decides to go ahead to produce a program that, according to his understanding of the problem, seems to close the gaps in a reasonable way. The customer then later sees the result and is often disappointed by the result. To avoid such negative surprises, it is crucial to reach out to the customer if the problem definition is not precise enough.

Hacking. . .

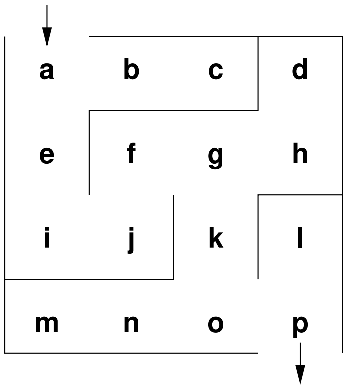
While hacking can be fun, it is often far more effective to think first before opening the editor and starting to write code. It often also helps to discuss the problem with others. Even coding together in pairs of two (pair programming) has been found to lead to better programs. Despite common believes, computer science in practice usually means a lot of team work and requires a great deal of communication.

Further online information:

- **Wikipedia**: Pair Programming

Problem Formalization (1/3)

- Think of a maze as a (two-dimensional) grid of rooms separated by walls.
- Each room can be given a name.
- Initially, every room is surrounded by four walls
- General idea:
  - Randomly knock out walls until we get a good maze.
  - How do we ensure there is a solution?
  - How do we ensure there is a unique solution?
  - How do we ensure every room is reachable?



Thinking of a maze as a (two-dimensional) grid seems natural, probably because we are used to two-dimensional mazes in the real world since childhood.

But what about one-dimensional mazes? Are they useful?

What about higher-dimensional mazes? Should we generalize the problem to consider arbitrary n-dimensional mazes? Quite surprisingly, generalizations sometimes lead to simpler solutions. As we will see later, the dimensionality of the maze does not really matter much.

And finally, what about mazes that are not rectangular?

Problem Formalization (2/3)

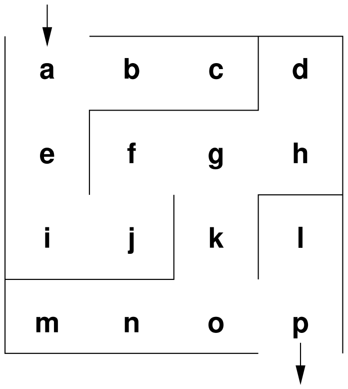
Lets try to formalize the problem in mathematical terms:

- We have a set V of rooms.
- We have a set E of pairs $\{x, y\}$ with $x \in V$ and $y \in V$ of adjacent rooms that have an open wall between them.

In the example, we have

- $V = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
- $\{a, b\} \in E$ and $\{g, k\} \in E$ and $\{a, c\} \notin E$ and $\{e, f\} \notin E$

Abstractly speaking, this is a mathematical structure called a graph consisting of a set of vertices (also called nodes) and a set of edges (also called links).

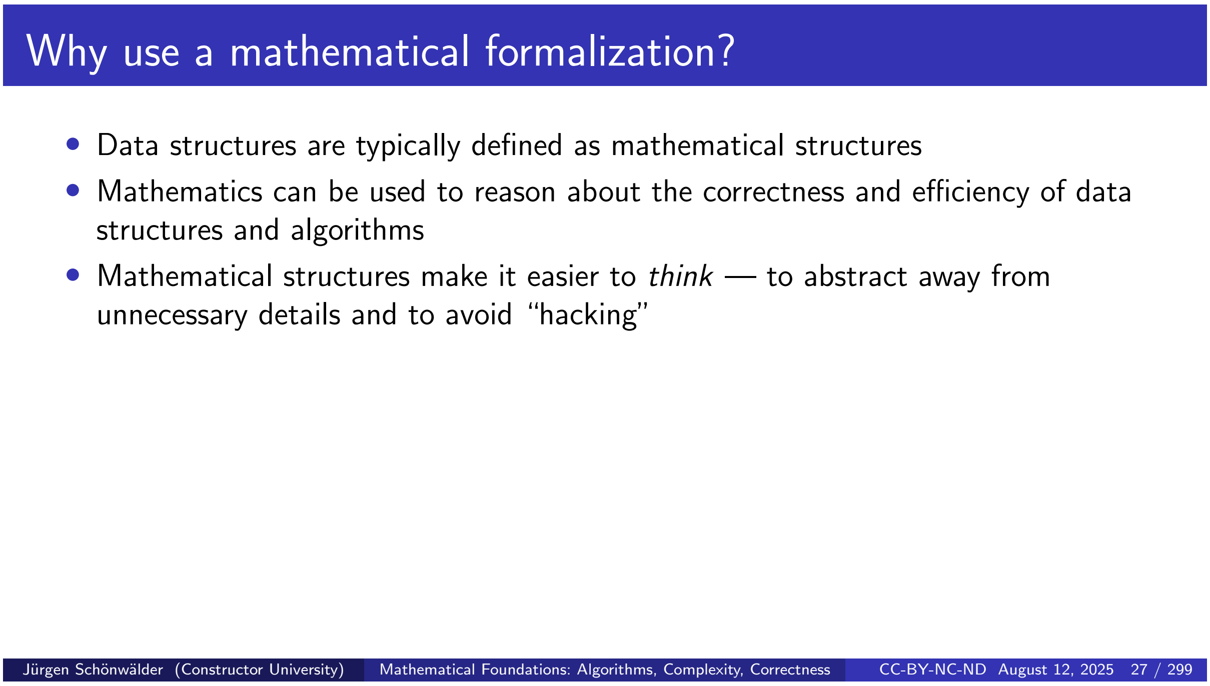


Graphs are very fundamental in computer science. Many real-world structures and problems have a graph representation. Relatively obvious are graphs representing the structure of relationships in social networks or graphs representing the structure of communication networks. Perhaps less obvious is that compilers internally often represent source code as graphs.

Note: What is missing in this problem formalization is how we represent (or determine) that two rooms are adjacent. (This is where the dimensions of the space come in.)

Further online information:

- **Wikipedia**: Graph



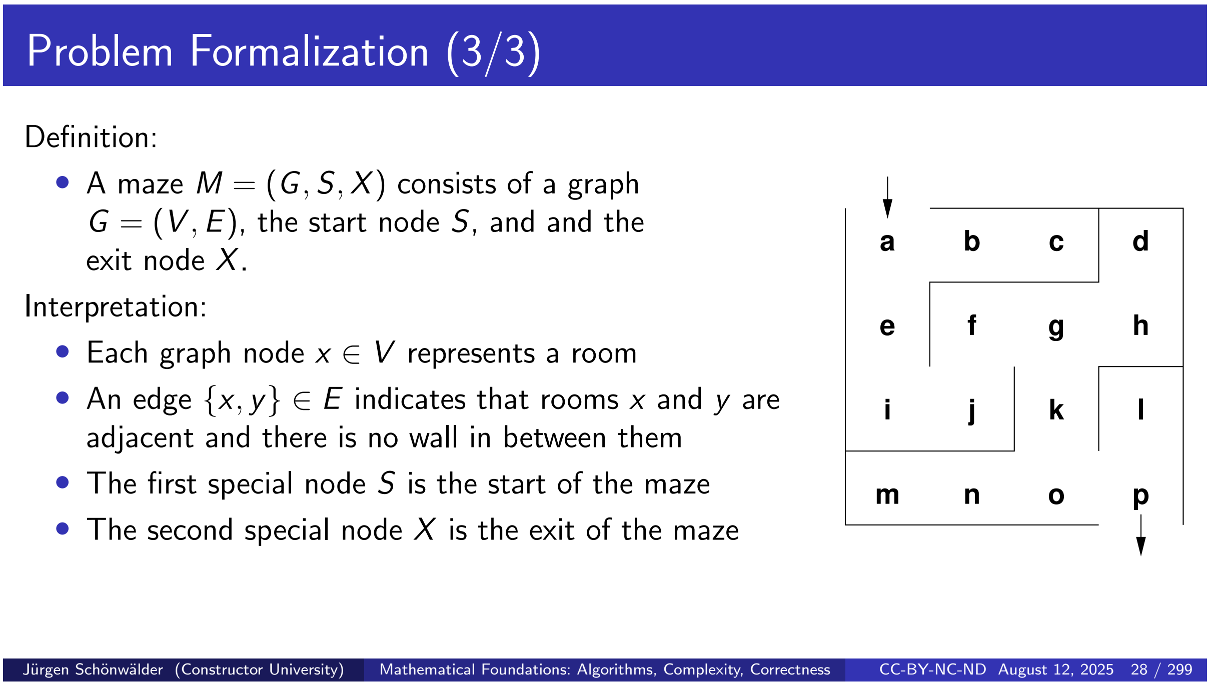
Why use a mathematical formalization?

- Data structures are typically defined as mathematical structures
- Mathematics can be used to reason about the correctness and efficiency of data structures and algorithms
- Mathematical structures make it easier to *think* — to abstract away from unnecessary details and to avoid "hacking"

Formalizing a problem requires us to think abstractly about what needs to be done. It requires us to identify clearly

- what our input is,
- what our output is, and
- what the task is that needs to be achieved.

Formalization also leads to a well-defined terminology that can be used to talk about the problem. Having a well-defined terminology is crucial for any teamwork. Without it, a lot of time is wasted because people talk past each other, often without discovering it. Keep in mind that larger programs are almost always the result of teamwork.



Problem Formalization (3/3)

Definition:

- A maze $M = (G, S, X)$ consists of a graph $G = (V, E)$, the start node S, and and the exit node X.

Interpretation:

- Each graph node $x \in V$ represents a room
- An edge $\{x, y\} \in E$ indicates that rooms x and y are adjacent and there is no wall in between them
- The first special node S is the start of the maze
- The second special node X is the exit of the maze

Another way of formulating this is to say that a maze M is described by a structure $M = (G, S, X)$ where $G = (V, E)$ is a graph with the vertices V and the edges E and $S \in V$ is the start node and $X \in V$ is the exit node.

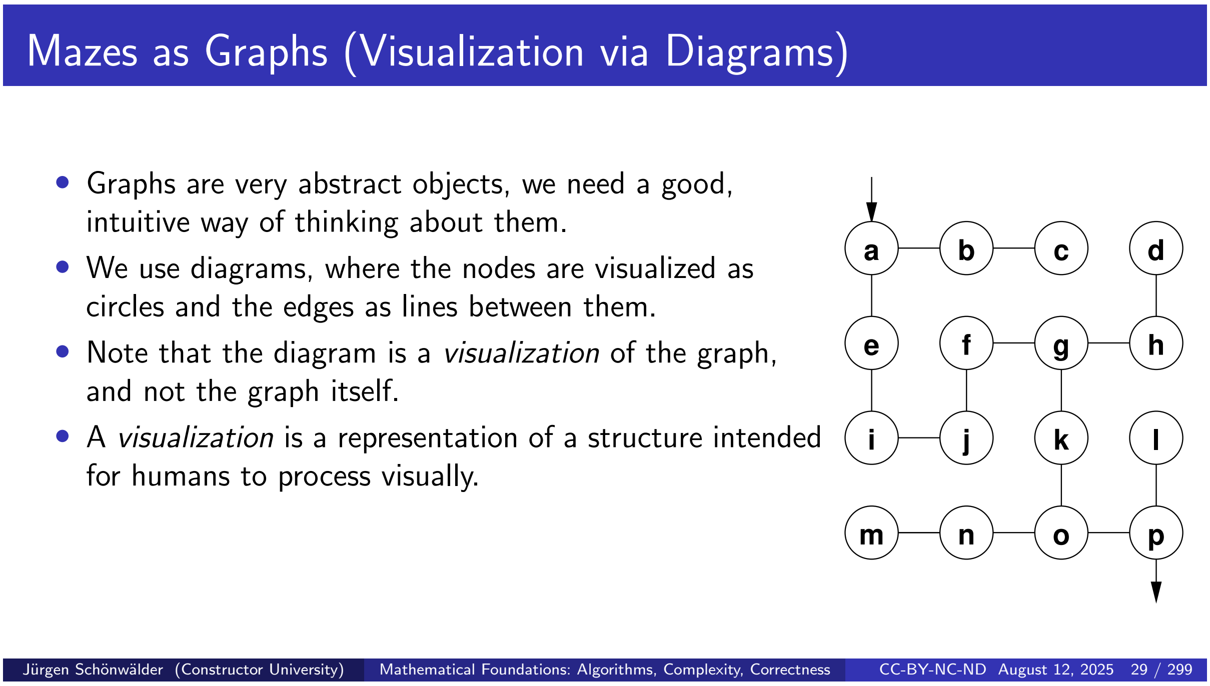
Given this formalization, the example maze M is represented as follows:

$$\begin{aligned}
M &= (G, S, X) \\
G &= (V, E) \\
V &= \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\} \\
E &= \{\{a, b\}, \{a, e\}, \{b, c\}, \{d, h\}, \{e, i\}, \{f, g\}, \{f, j\}, \{g, h\}, \\
&\quad \{g, k\}, \{i, j\}, \{k, o\}, \{l, p\}, \{m, n\}, \{n, o\}, \{o, p\}\} \\
S &= a \\
X &= p
\end{aligned}$$

The maze M is constructed as a subgraph of a supergraph $G_T = (V_T, E_T)$ representing the adjacency (topology) of the rooms:

$$\begin{aligned}
G_T &= (V_T, E_T) \\
V_T &= \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\} \\
E_T &= \{\{a, b\}, \{b, c\}, \{c, d\}, \{e, f\}, \{f, g\}, \{g, h\}, \{i, j\}, \{j, k\}, \{k, l\}, \{m, n\}, \{n, o\}, \{o, p\}, \\
&\quad \{a, e\}, \{b, f\}, \{c, g\}, \{d, h\}, \{e, i\}, \{f, j\}, \{g, k\}, \{h, i\}, \{i, m\}, \{j, n\}, \{k, o\}, \{l, p\}\}
\end{aligned}$$

Note that this is just one out of many different possible representations of a maze. Finding a good representation of a given problem requires experience and knowledge of many different possible representation approaches.



Mazes as Graphs (Visualization via Diagrams)

- Graphs are very abstract objects, we need a good, intuitive way of thinking about them.
- We use diagrams, where the nodes are visualized as circles and the edges as lines between them.
- Note that the diagram is a *visualization* of the graph, and not the graph itself.
- A *visualization* is a representation of a structure intended for humans to process visually.

Visualizations help humans to think about a problem. The human brain is very good in recognizing structures visually. But note that a visualization is just another representation and it might not be the best representation for a computer program. Also be aware that bad visualizations may actually hide structures.

Graphs like the one discussed here can also be represented using a graph notation. Here is how the graph looks like in the dot notation (used by the graphviz tools):

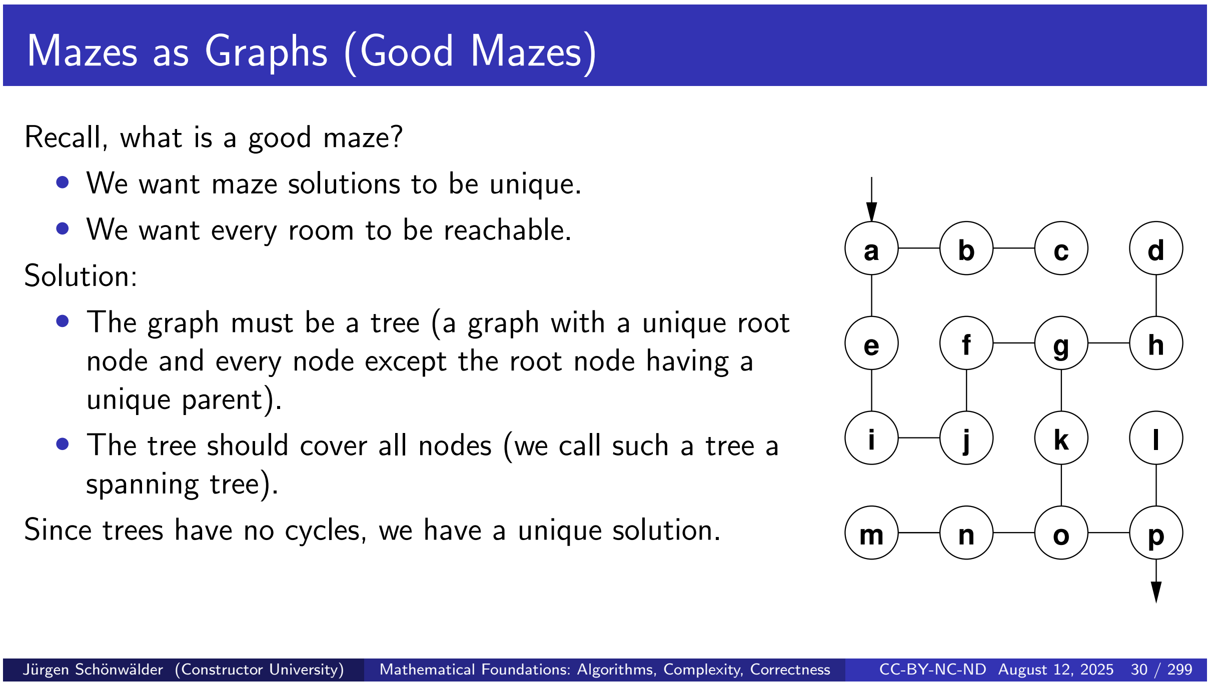
```
graph maze {
        a -- b -- c;
        a -- e -- i -- j -- f -- g;
        g -- h -- d;
        g -- k -- o;
        o -- m -- n;
        o -- p -- l;

        // _S an _X are additional invisible nodes with an edge
        // to the start and exit nodes.
        _S [style=invis]
        _S -- a
        _X [style=invis]
        p -- _X
}
```

Several graph drawing tools can read graph representations in dot notation and render graphs. Producing “good” drawings for a given graph is a non-trivial problem. You can look at different drawings of the graph by saving the graph definition in a file (say `maze.dot`) and then running the following commands to produce .pdf files (assuming you have the graphviz software package installed).

```
  $ neato -T pdf -o maze-neato.pdf maze.dot
  $ dot -T pdf -o maze-dot.pdf maze.dot
```

Further online information:

- **Wikipedia**: Graph Drawing
- **Web**: Graphviz



Mazes as Graphs (Good Mazes)

Recall, what is a good maze?

- We want maze solutions to be unique.
- We want every room to be reachable.

Solution:

- The graph must be a tree (a graph with a unique root node and every node except the root node having a unique parent).
- The tree should cover all nodes (we call such a tree a spanning tree).

Since trees have no cycles, we have a unique solution.

Apparently, we are not interested in arbitrary graphs but instead in spanning trees. So we need to solve the problem to construct a spanning tree rooted at the start node. This turns out to be a fairly general problem which is not specific to the construction of mazes.

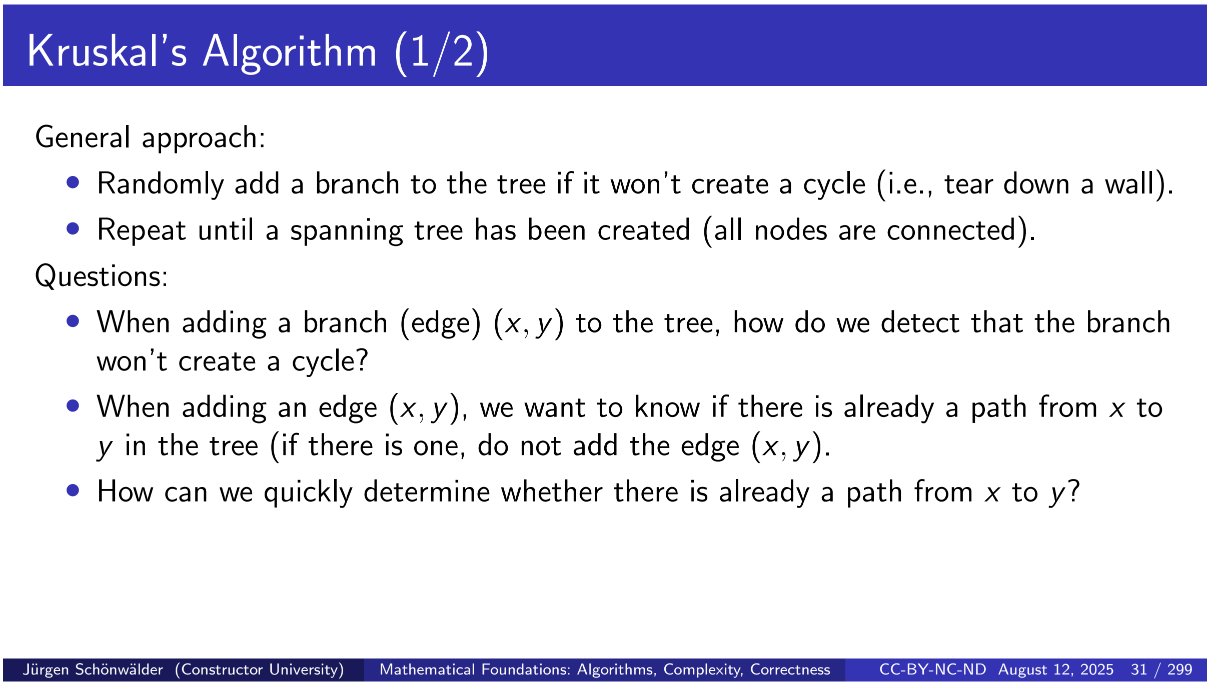
Note that in graph theory, a spanning tree $T = (V, E')$ of an simple connected graph $G = (V, E)$ is a connected subgraph of G with a minimum number of edges over all vertices of G. In general, a graph may have several spanning trees.

Spanning trees are important for communication networks in order to avoid loops. Spanning trees are also often used as building blocks in more complex algorithms.

Computer scientists draw trees in a somewhat unconventional fashion: The root is usually at the top and the tree grows towards the bottom.

Further online information:

- **Wikipedia**: Spanning Tree



Kruskal's Algorithm (1/2)

General approach:

- Randomly add a branch to the tree if it won't create a cycle (i.e., tear down a wall).
- Repeat until a spanning tree has been created (all nodes are connected).

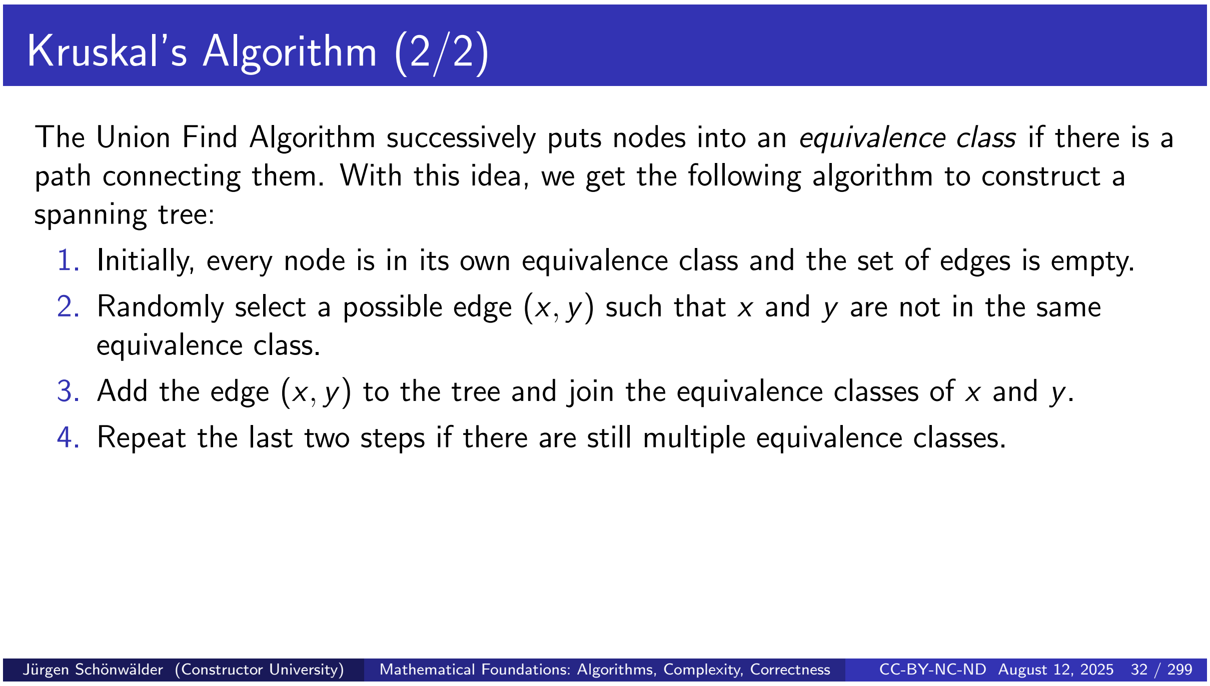
Questions:

- When adding a branch (edge) (x, y) to the tree, how do we detect that the branch won't create a cycle?
- When adding an edge (x, y), we want to know if there is already a path from x to y in the tree (if there is one, do not add the edge (x, y).
- How can we quickly determine whether there is already a path from x to y?

Kruskal's algorithm was published in 1956 [15] and has since then become one of the standard textbook algorithms for generating spanning trees. Kruskal's algorithm can be used to create minimum spanning trees if the edges of the graph are weighted and in each step an edge with the smallest weight is selected and added to the graph.

Further online information:

- **Wikipedia**: Kruskal's Algorithm



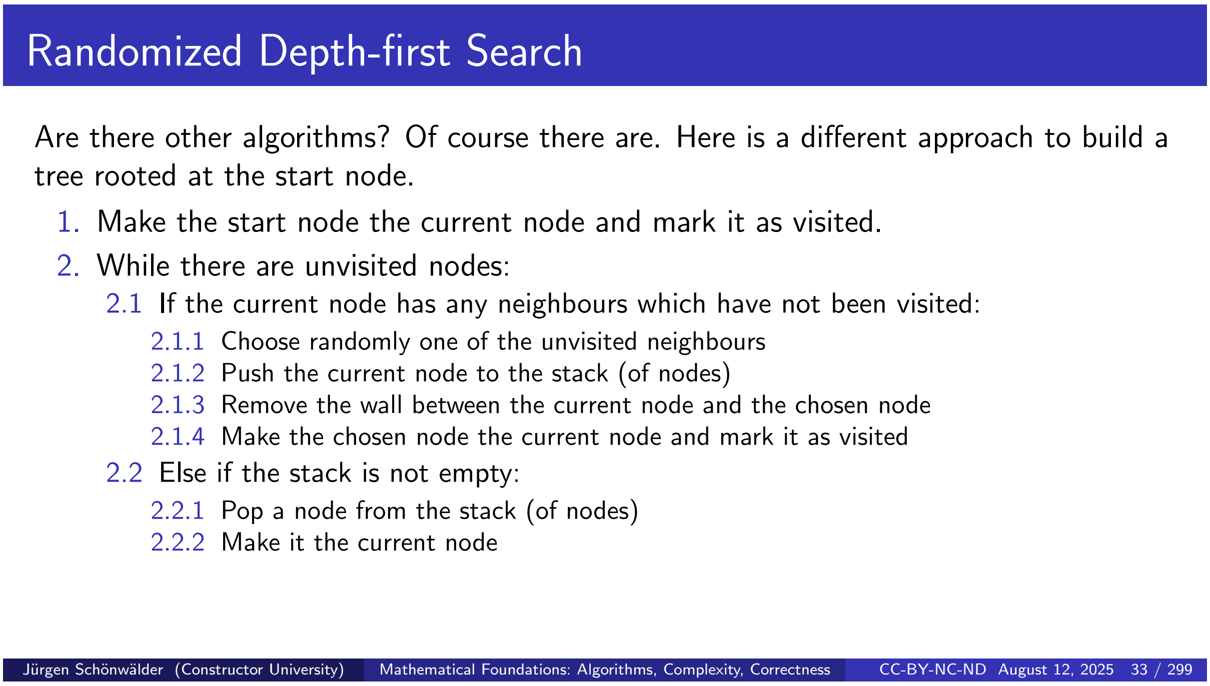
Kruskal's Algorithm (2/2)

The Union Find Algorithm successively puts nodes into an *equivalence class* if there is a path connecting them. With this idea, we get the following algorithm to construct a spanning tree:

1. Initially, every node is in its own equivalence class and the set of edges is empty.
2. Randomly select a possible edge (x, y) such that x and y are not in the same equivalence class.
3. Add the edge (x, y) to the tree and join the equivalence classes of x and y.
4. Repeat the last two steps if there are still multiple equivalence classes.

Further online information:

- **Wikipedia**: Equivalence Class
- **Wikipedia**: Disjoint Set Data Structure



Randomized Depth-first Search

Are there other algorithms? Of course there are. Here is a different approach to build a tree rooted at the start node.

1. Make the start node the current node and mark it as visited.
2. While there are unvisited nodes:
   - 2.1 If the current node has any neighbours which have not been visited:
     - 2.1.1 Choose randomly one of the unvisited neighbours
     - 2.1.2 Push the current node to the stack (of nodes)
     - 2.1.3 Remove the wall between the current node and the chosen node
     - 2.1.4 Make the chosen node the current node and mark it as visited
   - 2.2 Else if the stack is not empty:
     - 2.2.1 Pop a node from the stack (of nodes)
     - 2.2.2 Make it the current node

There are quite a few different algorithms to create mazes. Selecting a “good” one may not be easy. The different algorithms tend to create slightly different mazes in terms of the number of branching points, the number of dead ends, the path length distribution and other relevant properties.

Further online information:

- **Wikipedia**: Maze Generation Algorithm

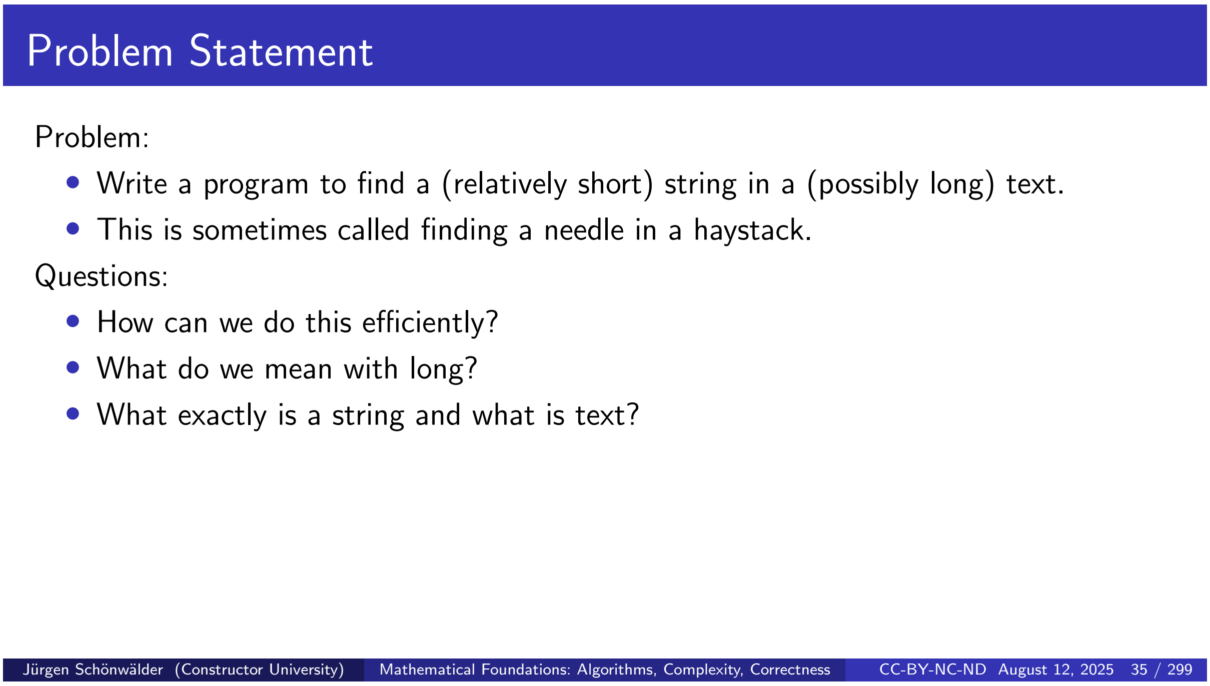
Section 3: String Search Algorithms

1 Computer Science and Algorithms

2 Maze Generation Algorithms

3 String Search Algorithms

4 Complexity and Correctness



Problem Statement

Problem:

- Write a program to find a (relatively short) string in a (possibly long) text.
- This is sometimes called finding a needle in a haystack.

Questions:

- How can we do this efficiently?
- What do we mean with long?
- What exactly is a string and what is text?

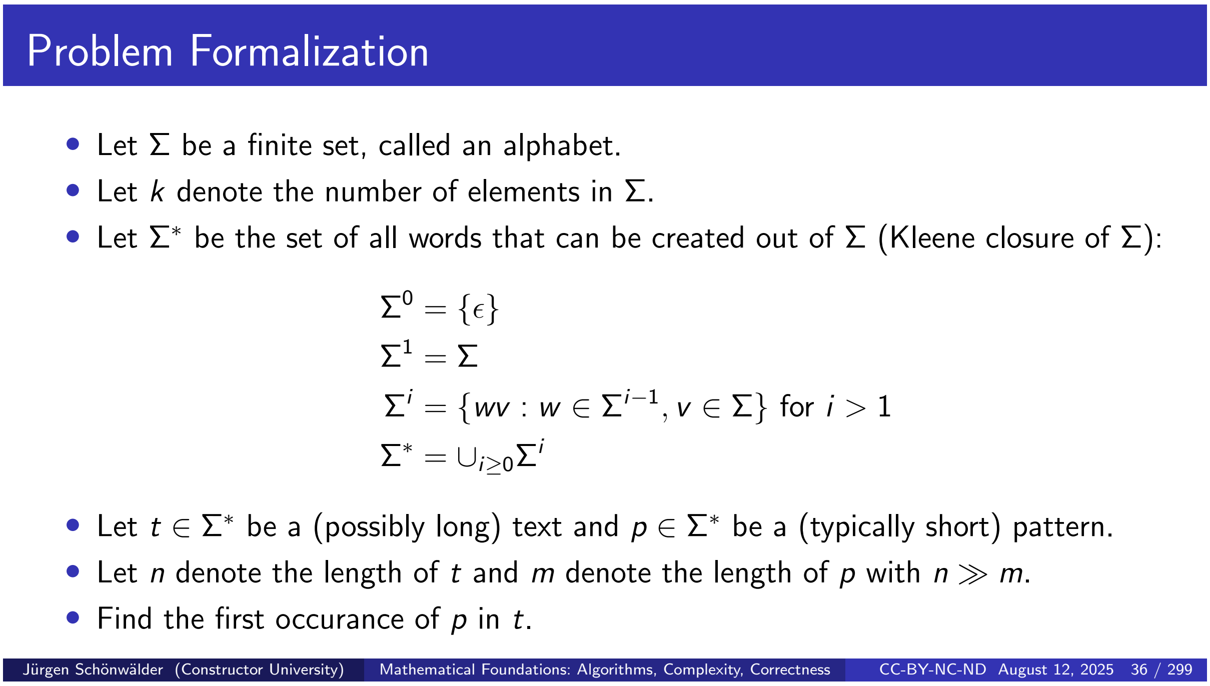
Searching is one of the main tasks computers do for us.

- Searching in the Internet for web pages.
- Searching within a web page for a given pattern.
- Searching for a pattern in network traffic.
- Searching for a pattern in a DNA.
- Searching for a malware pattern in executable files.

The search we are considering here is more precisely called a substring search. There are more expressive search techniques that we do not consider here, e.g., searching for regular expressions or fuzzy search techniques that support non-exact matches.

Further online information:

- **Wikipedia**: String Searching Algorithm



Problem Formalization

- Let Σ be a finite set, called an alphabet.
- Let k denote the number of elements in Σ.
- Let Σ^* be the set of all words that can be created out of Σ (Kleene closure of Σ):

$$
\begin{aligned}
\Sigma^0 &= \{\epsilon\} \\
\Sigma^1 &= \Sigma \\
\Sigma^i &= \{wv : w \in \Sigma^{i-1}, v \in \Sigma\} \text{ for } i > 1 \\
\Sigma^* &= \cup_{i \geq 0} \Sigma^i
\end{aligned}
$$

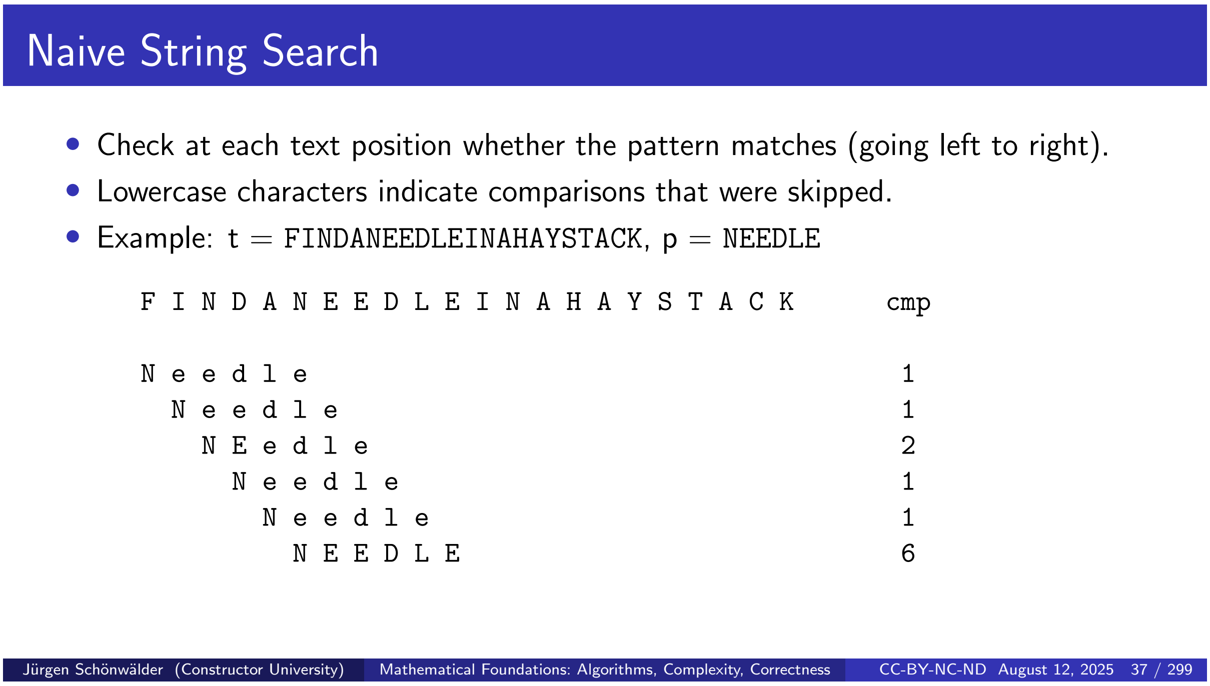
- Let $t \in \Sigma^*$ be a (possibly long) text and $p \in \Sigma^*$ be a (typically short) pattern.
- Let n denote the length of t and m denote the length of p with $n \gg m$.
- Find the first occurance of p in t.

The formalization introduces common terms that make it easier to discuss the problem. Furthermore, we introduce the abstract notion of an alphabet and we do not care anymore about the details how such an alphabet looks like. Some examples for alphabets:

- The characters of the Latin alphabet.
- The characters of the Universal Coded Character Set (Unicode)
- The binary alphabet $\Sigma = \{0, 1\}$.
- The DNA alphabet $\Sigma = \{A, C, G, T\}$ used in bioinformatics.
- The values 0...255 of a byte.

The Kleene closure Σ^* of the alphabet Σ is the (infinite) set of all words that can be formed with elements out of Σ. This includes the empty word of length 0, typically denoted by ϵ. Note that words here are any concatenation of elements of the alphabet, it does not matter whether the word is meaningful or not.

Note that the problem formalization details that we are searching for the first occurrence of p in t. We could have defined the problem differently, e.g., searching for the last occurrence of p in t, or searching for all occurrences of p in t, or searching for any occurrence of p in t.



Naive String Search

- Check at each text position whether the pattern matches (going left to right).
- Lowercase characters indicate comparisons that were skipped.
- Example: t = `FINDANEEDLEINAHAYSTACK`, p = `NEEDLE`

```
F I N D A N E E D L E I N A H A Y S T A C K      cmp

N e e d l e                                        1
  N e e d l e                                      1
    N E e d l e                                    2
      N e e d l e                                  1
        N e e d l e                                1
          N E E D L E                              6
```

An implementation of naive string search in an imperative language like C is straight forward (see Listing 1). You need two nested for-loops, the outer loop iterates over all possible alignments and the inner loop iterates over the pattern to test whether the pattern matches the current part of the text.

```c
/*
 * naive-search/search.c --
 *
 * Implementation of naive string search in C.
 */

#include <stdlib.h>
#include "search.h"

const char *
search(const char *haystack, const char *needle)
{
    const char *t, *p, *r;

    for (t = haystack; *t; t++) {
        for (p = needle, r = t; *r && *p && *r == *p; p++, r++) ;
        if (! *p) {
            return t;
        }
    }

    return NULL;
}
```

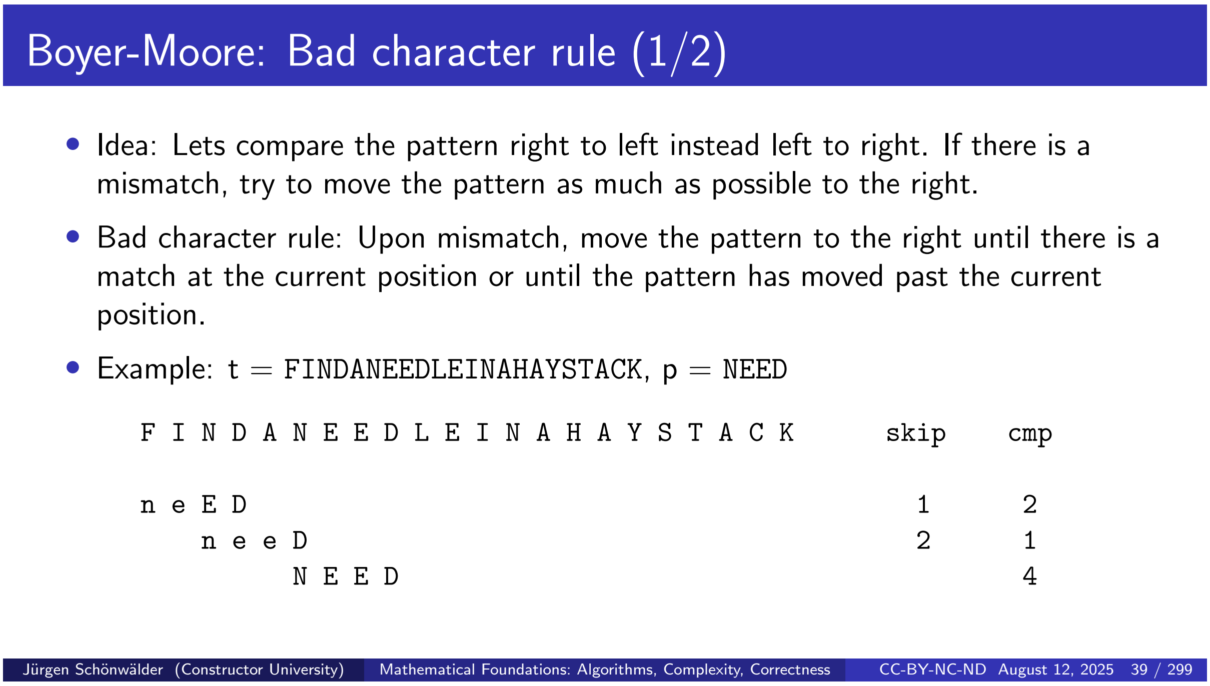
Listing 1: Naive string search implemented in C

Naive String Search Performance

- How “fast” is naive string search?
- Idea: Lets count the number of comparisons.
- Problem: The number of comparisons depends on the strings.
- Idea: Consider the worst case possible.
- What is the worst case possible?
  - Consider a haystack of length n using only a single symbol of the alphabet (e.g., “aaaaaaaaaa” with $n = 10$).
  - Consider a needle of length m which consists of $m - 1$ times the same symbol followed by a single symbol that is different (e.g., “aax” with $m = 3$).
  - With $n \gg m$, the number of comparisons needed will be roughly $n \cdot m$.

When talking about the performance of an algorithm, it is often useful to consider performance in the best case, performance in the worst case, and performance in average cases. Furthermore, performance is typically discussed in terms of processing steps (time) and in terms of memory required (space). There is often an interdependency between time and space and it is often possible to trade space against time or vice versa.

The naive string search is very space efficient but not very time efficient. Alternative search algorithms can be faster, but they require some extra space.



Boyer-Moore: Bad character rule (1/2)

- Idea: Lets compare the pattern right to left instead left to right. If there is a mismatch, try to move the pattern as much as possible to the right.
- Bad character rule: Upon mismatch, move the pattern to the right until there is a match at the current position or until the pattern has moved past the current position.
- Example: t = FINDANEEDLEINAHAYSTACK, p = NEED

```
F I N D A N E E D L E I N A H A Y S T A C K      skip    cmp

n e E D                                             1      2
    n e e D                                         2      1
          N E E D                                          4
```

The bad character rule allows us to skip alignments:

1. In the initial alignment, we find that D is matching but E is not matching the N. So we check whether there is an N in the part of the pattern not tested yet that we can align with the N. In this case there is an N in the pattern and we can skip 1 alignment.
2. In the second alignment, we find that D is not matching N and so we check whether there is an N in the part of the pattern not tested yet. In this case, we can skip 2 alignments.
3. In the third alignment, we find a match.

In this case, we have skipped 3 alignments and we used 3 alignments to find a match. With the naive algorithm we would have used 6 alignments. We have performed 7 comparisons in this case while the naive algorithm used 12 comparisons.

Boyer-Moore: Bad character rule (2/2)

- Example: t = `FINDANEEDLEINAHAYSTACK`, p = `HAY`

```
F I N D A N E E D L E I N A H A Y S T A C K       skip    cmp

h a Y                                               2       1
      h a Y                                         2       1
            h a Y                                   2       1
                  h a Y                             2       1
                        h a Y                       1       1
                            H A Y                           3
```

- How do we decide efficiently how far we can move the pattern to the right?

In this example, we test four times against a character in the text that is not present in the pattern. Hence we can skip 2 alignments each time. In the fifth alignment, we compare Y against H and since H is in the pattern, we skip 1 alignment. So overall, we have skipped 9 alignments. (With naive string search, we would check 15 alignments, Boyer-Moore only requires 6 alignments.)

In order to determine how many alignments we can skip, we need a function that takes the current position in the pattern and the character of the text that does not match and returns the number of alignments that can be skipped. A naive implementation of this function requires again several comparisons. In order to make this more efficient, we can pre-compute all possible skips and store the skips in a two-dimensional table. This way, we can simply lookup the number of alignments that can be skipped by indexing into the table. The table can be seen as a function that maps the unmatched character and the position in the pattern to the number of alignments that can be skipped.

Example: Lets assume $p = NEED$ and an alphabet consisting of the characters A-Z. Then the table looks as shown below on the left (counting character positions in the pattern starting with 0). Since all rows for characters not appearing in the pattern are similar, we might also represent them using a wildcard row as shown below on the right.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ \vdots \\ N \\ \vdots \\ Z \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & - \\ 0 & - & - & 0 \\ \vdots & \vdots & \vdots & \vdots \\ - & 0 & 1 & 2 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 2 & 3 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ D \\ E \\ N \\ * \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & - \\ 0 & - & - & 0 \\ - & 0 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix} \end{array}$$

The pre-computation of a lookup table is key to the performance of the Boyer-Moore bad character rule. The size of the lookup table and the time to compute it depends only on the length of the pattern and the size of the alphabet. The effort is independent of the length of the text. Since we assume that the text is significantly longer than the pattern, the effort to calculate the lookup table becomes irrelevant for very long texts.

Boyer-Moore: Good suffix rule (1/3)

- Idea: If we already matched a suffix and the suffix appears again in the pattern, skip the alignment such that we keep the good suffix.
- Good suffix rule: Let s be a non-empty suffix already matched in the inner loop. If there is a mismatch, skip alignments until (i) there is another match of the suffix (which may include the mismatching character), or (ii) a prefix of p matches a suffix of s or (iii) skip until the end of the pattern if neither (i) or (ii) apply to the non-empty suffix s.
- Example: t = FINDANEEDLEINAHAYSTACK, p = NEEDUNEED

```
F I N D A N E E D L E I N A H A Y S T A C K     skip    cmp

n e e d U N E E D                                 4       5
          n e e d u n e e D                               1
```

This example demonstrates case (i) of the good suffix rule. We have matched the suffix NEED and we have a mismatch of U against A. Since the pattern contains this suffix again left to the current comparison position, we move the pattern right to align with this suffix.

Boyer-Moore: Good suffix rule (2/3)

- Example: t = FINDANEEDLEINAHAYSTACK, p = EDISUNEED

```
F I N D A N E E D L E I N A H A Y S T A C K      skip    cmp

e d i s U N E E D                                  6       5
              e d i s u n e e D                            1
```

This example demonstrates case (ii) of the good suffix rule. We have matched the suffix NEED and we have a mismatch of U against A. The prefix of the pattern contains the suffix ED of our suffix NEED and hence we move to align this prefix with the matched suffix of our suffix.

Boyer-Moore: Good suffix rule (3/3)

- Example: t = FINDANEEDLEINAHAYSTACK, p = FOODINEED

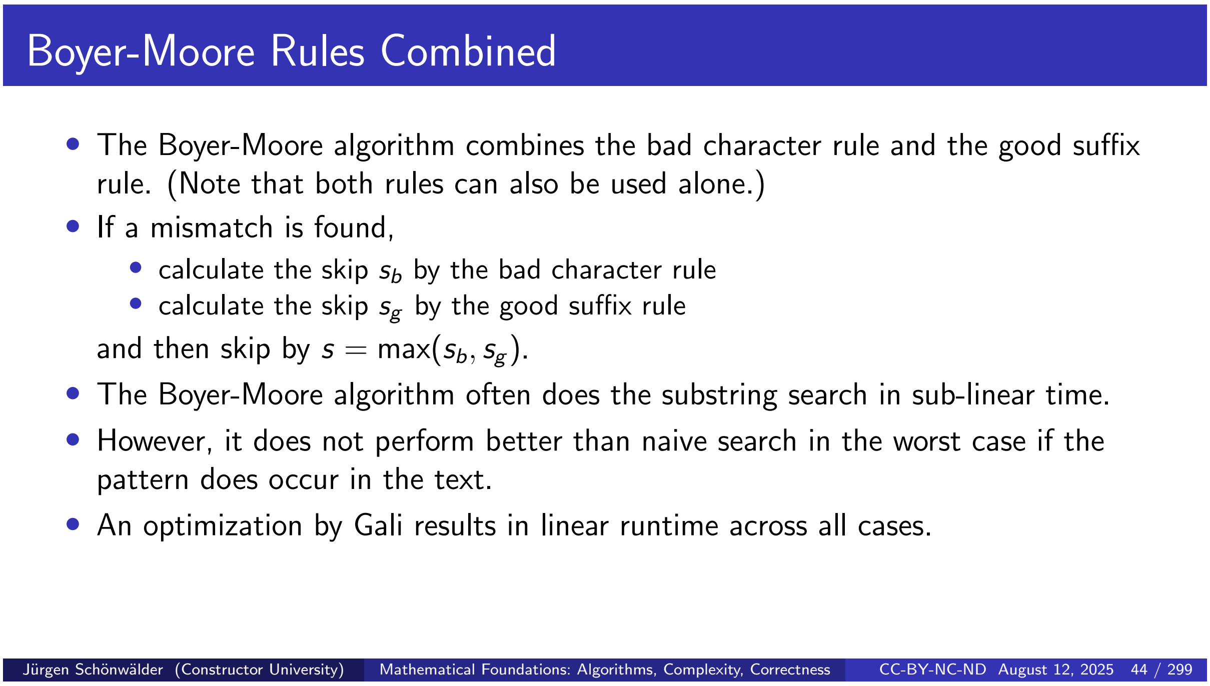
```
F I N D A N E E D L E I N A H A Y S T A C K        skip    cmp

f o o d I N E E D                                    8      5
                  f o o d i n e e D                         1
```

- How do we decide efficiently how far we can move the pattern to the right?

This example demonstrates case (iii) of the good suffix rule. We neither have a matching suffix nor does the prefix match a suffix of the suffix. Hence we skip alignments until the end of the current alignment.

The good suffix rule is actually a bit complex. Consult wikipedia or a textbook on algorithms or the original publication for a complete description of the good suffix rule.

To implement the good suffix rule efficiently, lookup tables are again needed to quickly lookup how many alignments can be skipped. Given a pattern p, the lookup tables can be calculated, that is, the lookup tables do not depend on the text being searched.



Boyer-Moore Rules Combined

- The Boyer-Moore algorithm combines the bad character rule and the good suffix rule. (Note that both rules can also be used alone.)
- If a mismatch is found,
  - calculate the skip s_b by the bad character rule
  - calculate the skip s_g by the good suffix rule

  and then skip by $s = \max(s_b, s_g)$.
- The Boyer-Moore algorithm often does the substring search in sub-linear time.
- However, it does not perform better than naive search in the worst case if the pattern does occur in the text.
- An optimization by Gali results in linear runtime across all cases.

The Boyer-Moore algorithm demonstrates that in computer science we sometimes trade space against time. The lookup table reduces the time needed to perform the search but it requires additional space to store the lookup table.

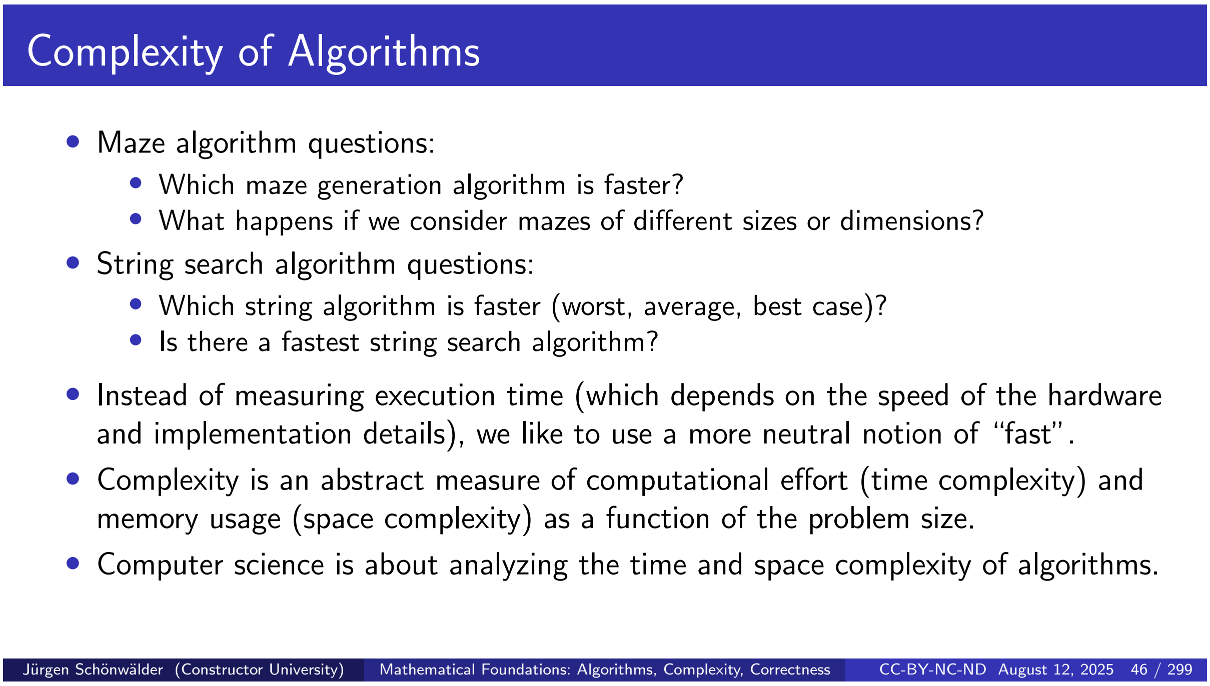
The Boyer-Moore algorithm was introduced in 1977 [3] and the improvement by Gali for worst cases in 1979 in [7].

Further online information:

- **YouTube**: Boyer-Moore: basics
- **YouTube**: Boyer-Moore: putting it all together

Section 4: Complexity and Correctness

1. Computer Science and Algorithms
2. Maze Generation Algorithms
3. String Search Algorithms
4. Complexity and Correctness



Complexity of Algorithms

- Maze algorithm questions:
    - Which maze generation algorithm is faster?
    - What happens if we consider mazes of different sizes or dimensions?
- String search algorithm questions:
    - Which string algorithm is faster (worst, average, best case)?
    - Is there a fastest string search algorithm?
- Instead of measuring execution time (which depends on the speed of the hardware and implementation details), we like to use a more neutral notion of “fast”.
- Complexity is an abstract measure of computational effort (time complexity) and memory usage (space complexity) as a function of the problem size.
- Computer science is about analyzing the time and space complexity of algorithms.

The performance analysis of algorithms is a very important part of computer science. Since we are generally not so much interested in execution times that depend on the hardware components of a computer system, we like to have a more abstract way of talking about the “performance” of an algorithm. We call this abstract measure of “performance” the complexity of an algorithm and we usually distinguish the time complexity (the computational effort) and the space complexity (how much storage is required).

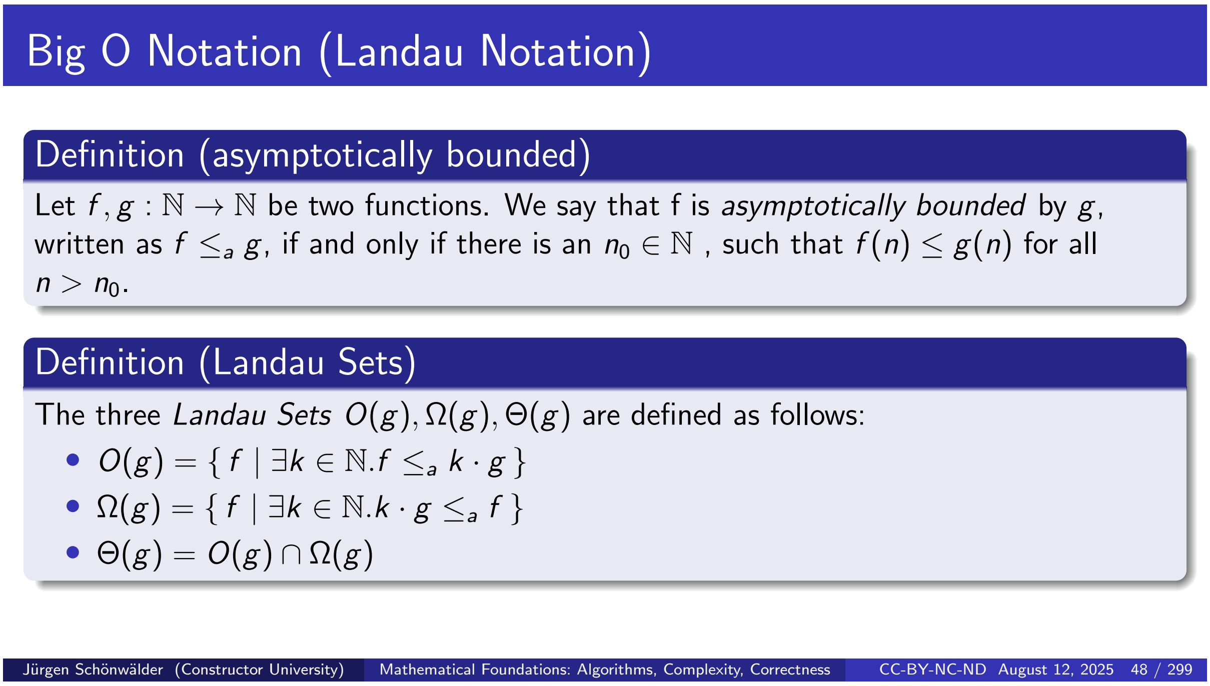
We will discuss this further later in the course and we will introduce a framework that allows us to define classes of complexity.

Performance and Scaling

size n	$t(n) = 100n$ µs	$t(n) = 7n^2$ µs	$t(n) = 2^n$ µs
1	100 µs	7 µs	2 µs
5	500 µs	175 µs	32 µs
10	1 ms	700 µs	1024 µs
50	5 ms	17.5 ms	13 031.25 d
100	10 ms	70 ms	
1000	100 ms	7 s	
10 000	1 s	700 s	
100 000	10 s	70 000 s	

- Suppose we have three algorithms to choose from (linear, quadratic, exponential).
- With $n = 50$, the exponential algorithm runs for more than 35 years.
- For $n \geq 1000$, the exponential algorithm runs longer than the age of the universe!

Something that scales exponentially with the problem size quickly becomes intractable. In theoretical computer science, we will look at the question whether there are problems that are inherently exponential. We will also investigate whether we can show the best possible solution for a given problem in terms of complexity. Once you prove for a given problem that the best possible solution is lets say quadratic, you can stop searching for a linear solution (this can literally save you endless nights of work).



Big O Notation (Landau Notation)

Definition (asymptotically bounded)

Let $f, g : \mathbb{N} \to \mathbb{N}$ be two functions. We say that f is *asymptotically bounded* by g, written as $f \leq_a g$, if and only if there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.

Definition (Landau Sets)

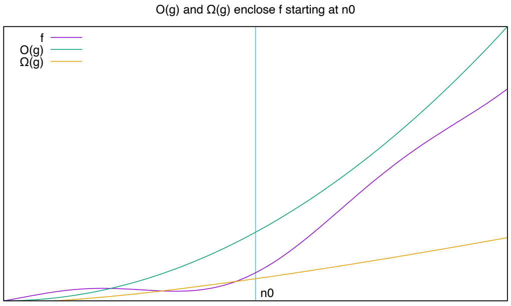
The three *Landau Sets* $O(g), \Omega(g), \Theta(g)$ are defined as follows:

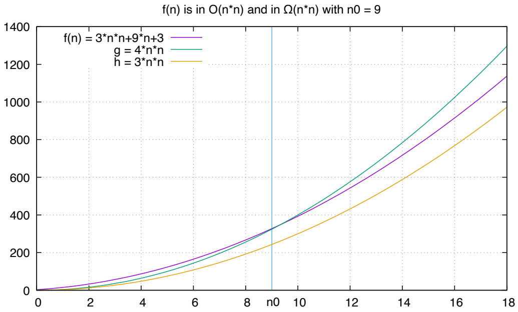
- $O(g) = \{ f \mid \exists k \in \mathbb{N}. f \leq_a k \cdot g \}$
- $\Omega(g) = \{ f \mid \exists k \in \mathbb{N}. k \cdot g \leq_a f \}$
- $\Theta(g) = O(g) \cap \Omega(g)$

Interpretation of the three Landau Sets:

- $f \in O(g)$: f does not grow significantly faster than g
- $f \in \Omega(g)$: f grows not significantly slower than g
- $f \in \Theta(g)$: f grows as fast as g

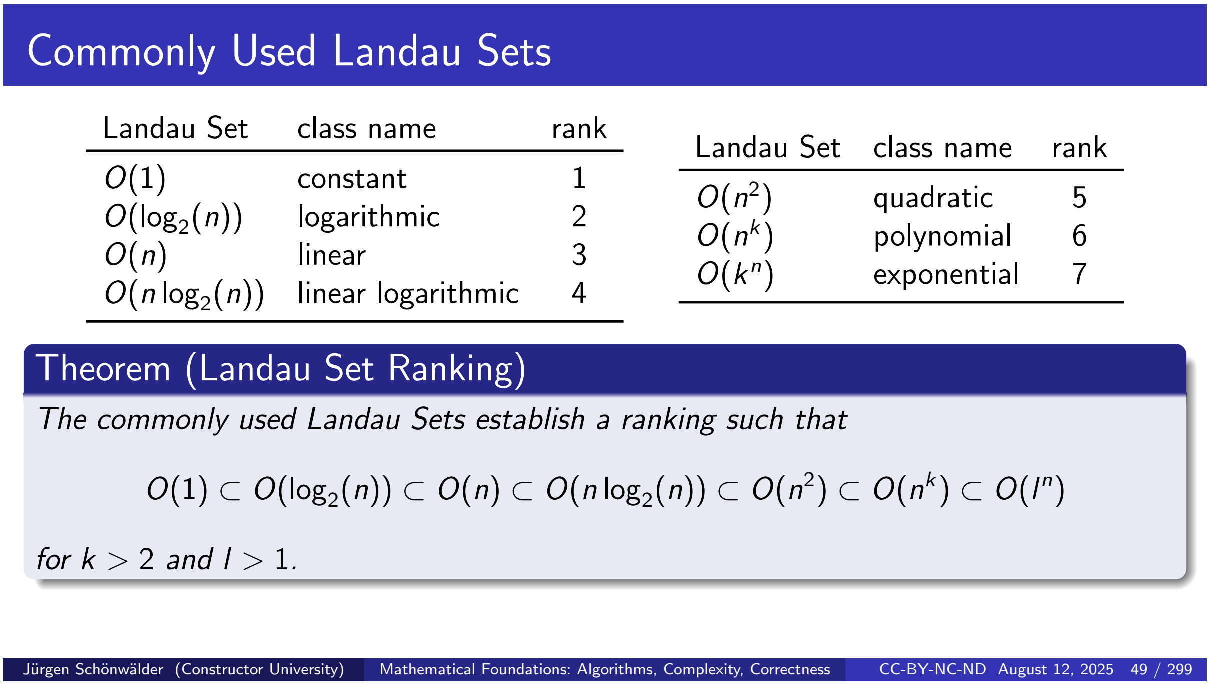
For a given function f, we are usually interested in finding the “smallest” upper bound $O(g)$ and the “largest” lower bound $\Omega(g)$. The “smallest upper bound” is interesting when we consider the worst case behaviour of an algorithm, the “largest lower bound” is interesting when we consider the best case behaviour.





As an example, lets consider the function $f(n) = 3n^2 + 9n + 3$. We try to prove that $f \in \Theta(n^2)$:

- Lets first show that f is in $O(n^2)$. We have to find a $k \in \mathbb{N}$ and an $n_0 \in \mathbb{N}$ such that $f(n) \leq kn^2$ for $n > n_0$. Lets pick $k = 4$. Apparently, $f(n) = 3n^2 + 9n + 3 \geq 4n^2$ for $n \in \{0, \ldots, 9\}$, but $f(n) \leq 4n^2$ for $n > 9 = n_0$. Hence, f is in $O(n^2)$.
- We now show that f is in $\Omega(n^2)$. We have to find a $k \in \mathbb{N}$ and an $n_0 \in \mathbb{N}$ such that $f(n) \geq kn^2$ for $n > n_0$. Lets pick $k = 3$. Apparently, $f(n) = 3n^2 + 9n + 3 \geq 3n^2$ since we add a positive value to $3n^2$. Hence, f is in $\Omega(n^2)$.
- Since $f \in O(n^2)$ and $f \in \Omega(n^2)$, it follows that $f \in \Theta(n^2)$.



Commonly Used Landau Sets

Landau Set	class name	rank
$O(1)$	constant	1
$O(\log_2(n))$	logarithmic	2
$O(n)$	linear	3
$O(n\log_2(n))$	linear logarithmic	4

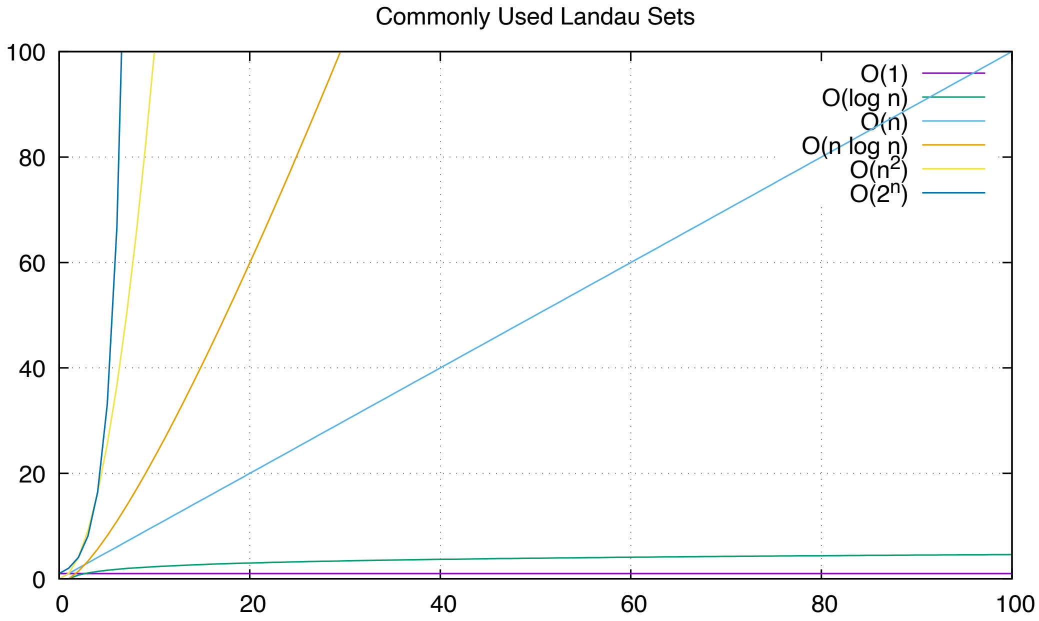
Landau Set	class name	rank
$O(n^2)$	quadratic	5
$O(n^k)$	polynomial	6
$O(k^n)$	exponential	7

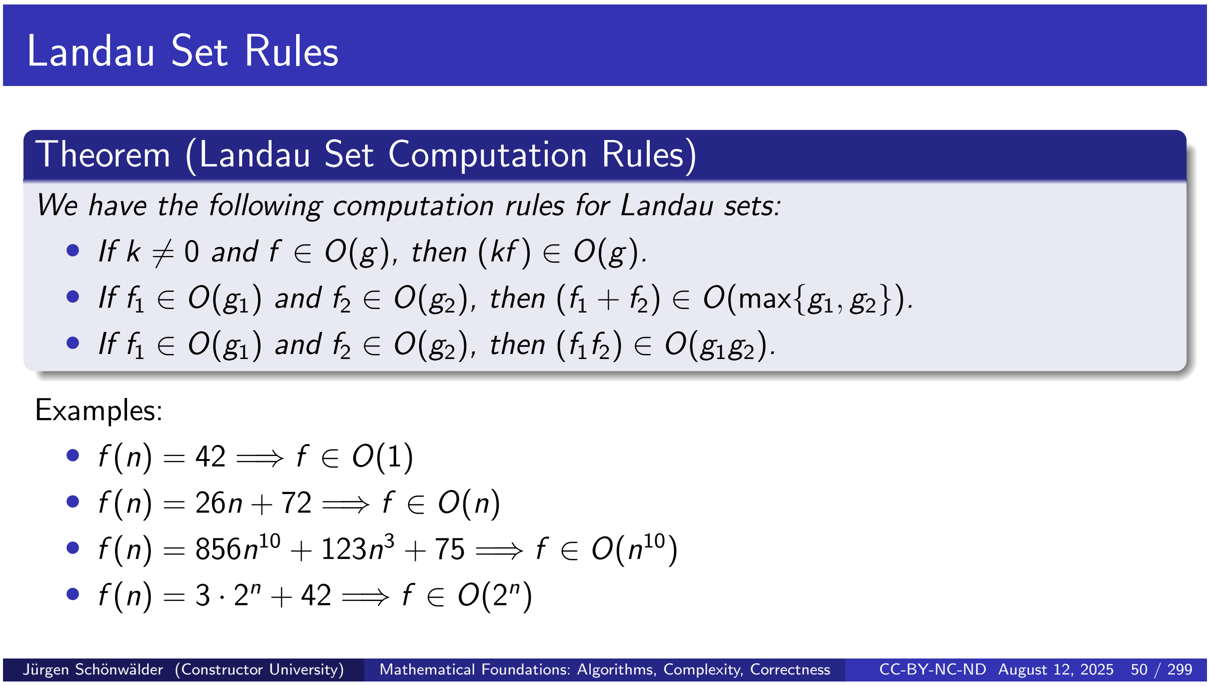
Theorem (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

$$O(1) \subset O(\log_2(n)) \subset O(n) \subset O(n\log_2(n)) \subset O(n^2) \subset O(n^k) \subset O(l^n)$$

for $k > 2$ and $l > 1$.





Landau Set Rules

Theorem (Landau Set Computation Rules)

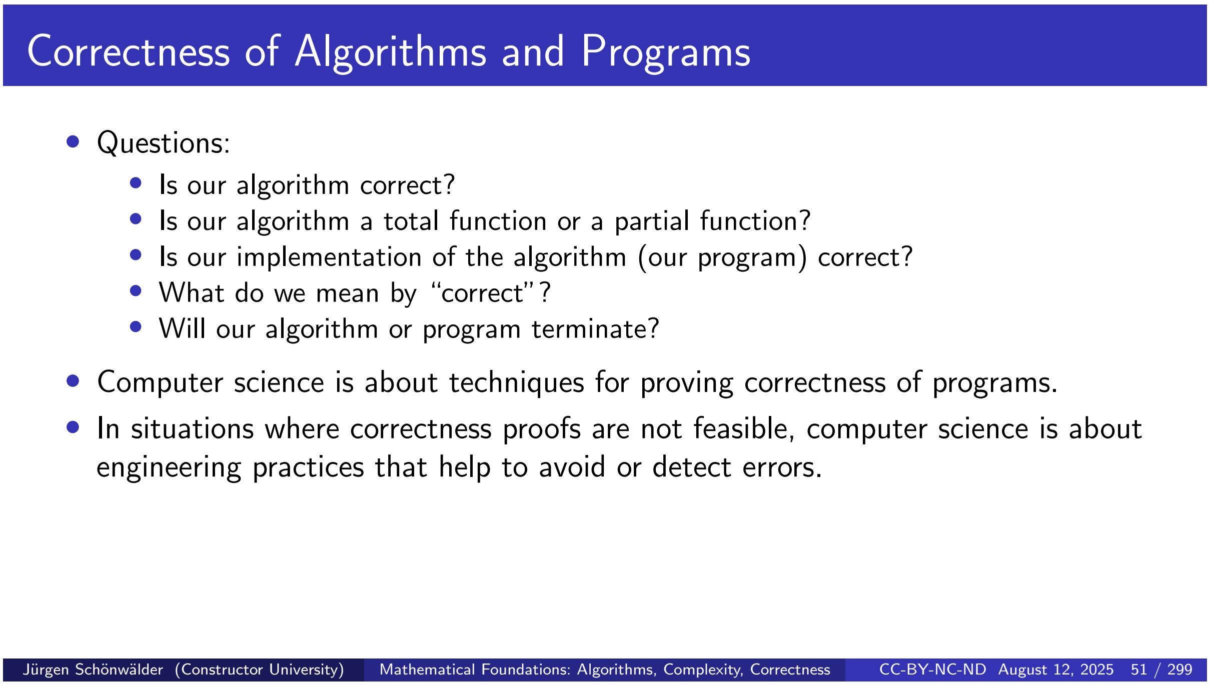
We have the following computation rules for Landau sets:

- *If $k \neq 0$ and $f \in O(g)$, then $(kf) \in O(g)$.*
- *If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $(f_1 + f_2) \in O(\max\{g_1, g_2\})$.*
- *If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $(f_1 f_2) \in O(g_1 g_2)$.*

Examples:

- $f(n) = 42 \Longrightarrow f \in O(1)$
- $f(n) = 26n + 72 \Longrightarrow f \in O(n)$
- $f(n) = 856n^{10} + 123n^3 + 75 \Longrightarrow f \in O(n^{10})$
- $f(n) = 3 \cdot 2^n + 42 \Longrightarrow f \in O(2^n)$

- The Big O Notation describes the limiting behavior of a function when the argument tends towards a particular value of infinity.
- We classify a function describing the (time or space) complexity of an algorithm by determining the closest Landau Set it belongs to.
- Good sequential sorting algorithms achieve a time complexity $O(n \log n)$, simpler algorithms often belong to $O(n^2)$. Some sorting algorithms do not need any extra memory, so they achieve $O(1)$ in terms of space complexity.
- Use O classes for worst case complexity, use Ω classes for best case complexity.

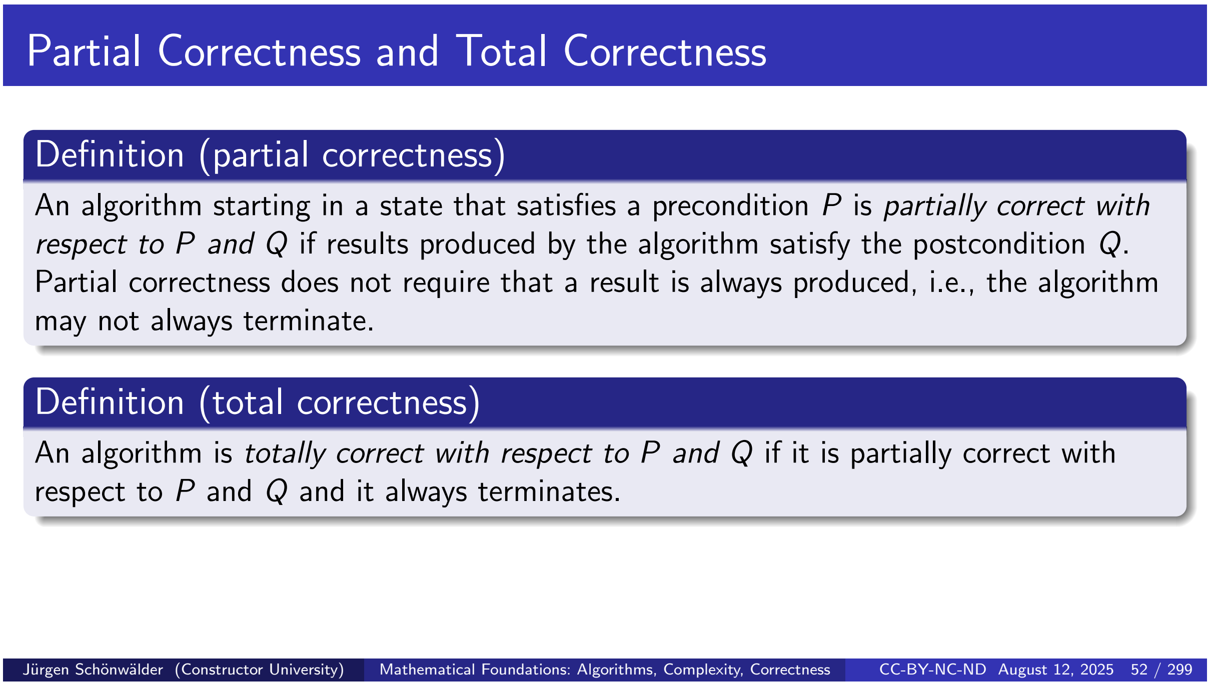


Correctness of Algorithms and Programs

- Questions:
  - Is our algorithm correct?
  - Is our algorithm a total function or a partial function?
  - Is our implementation of the algorithm (our program) correct?
  - What do we mean by “correct”?
  - Will our algorithm or program terminate?
- Computer science is about techniques for proving correctness of programs.
- In situations where correctness proofs are not feasible, computer science is about engineering practices that help to avoid or detect errors.

Note the difference between the correctness of an algorithm and the correctness of a program implementing an algorithm. While correctness proofs are feasible, they are difficult and thus expensive. As a consequence, they are done mostly in situations where a potential failure of an algorithm or its implementation can cause damages that are far more costly than the correctness proof.

Hence, for many software systems, we tend to rely on testing techniques in the hope that good test coverage will reduce errors and the likelihood of bad failures. But testing never can proof the absence of errors (unless all possible inputs and outputs can be tested - which usually is infeasible for anything more complicated than a hello world program).



Partial Correctness and Total Correctness

Definition (partial correctness)

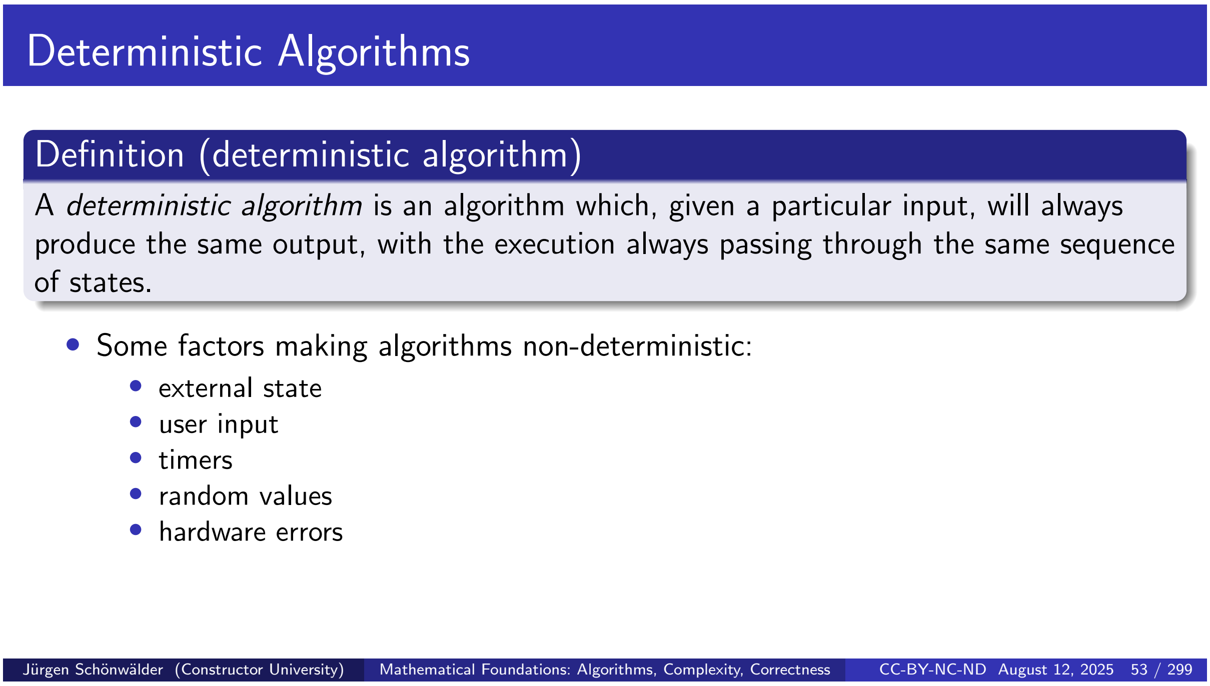
An algorithm starting in a state that satisfies a precondition P is *partially correct with respect to P and Q* if results produced by the algorithm satisfy the postcondition Q. Partial correctness does not require that a result is always produced, i.e., the algorithm may not always terminate.

Definition (total correctness)

An algorithm is *totally correct with respect to P and Q* if it is partially correct with respect to P and Q and it always terminates.

In order to talk about the correctness of an algorithm, we need a specification that clearly states the precondition P and the postcondition Q. In other words, an algorithm can only be correct regarding a concrete specification, ideally a problem formalization. Without a precise specification, it is impossible to say whether an algorithm is correct or not.

The distinction between partial correctness and total correctness is important. Total correctness requires a termination proof and unfortunately an automated termination proof is impossible for arbitrary algorithms.



Deterministic Algorithms

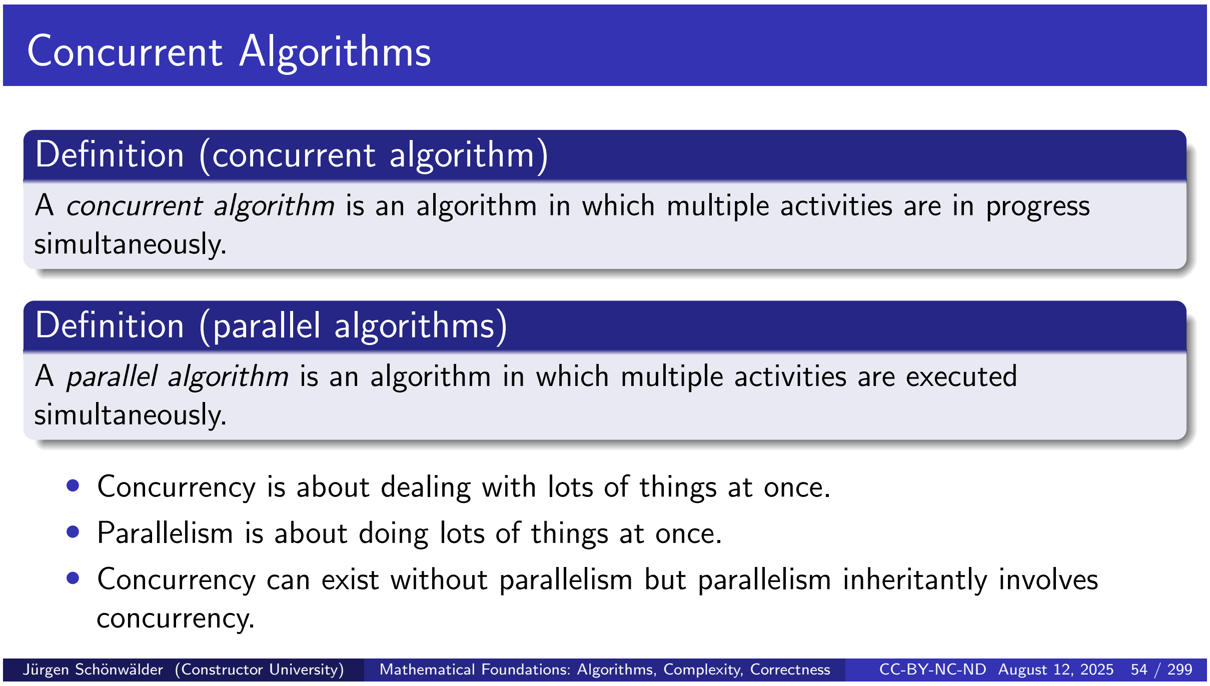
Definition (deterministic algorithm)

A *deterministic algorithm* is an algorithm which, given a particular input, will always produce the same output, with the execution always passing through the same sequence of states.

- Some factors making algorithms non-deterministic:
  - external state
  - user input
  - timers
  - random values
  - hardware errors

Deterministic algorithms are often easier to understand and analyze. Real software systems, however, are rarely fully deterministic since they interact with a world that is largely non-deterministic.

Computer science is spending a lot of effort trying to make the execution of algorithms deterministic. Operating systems, for example, deal with a large amount of nondeterminism originating from computing hardware and they try to provide an execution environment for programs that is less nondeterministic than the hardware components.



Concurrent Algorithms

Definition (concurrent algorithm)

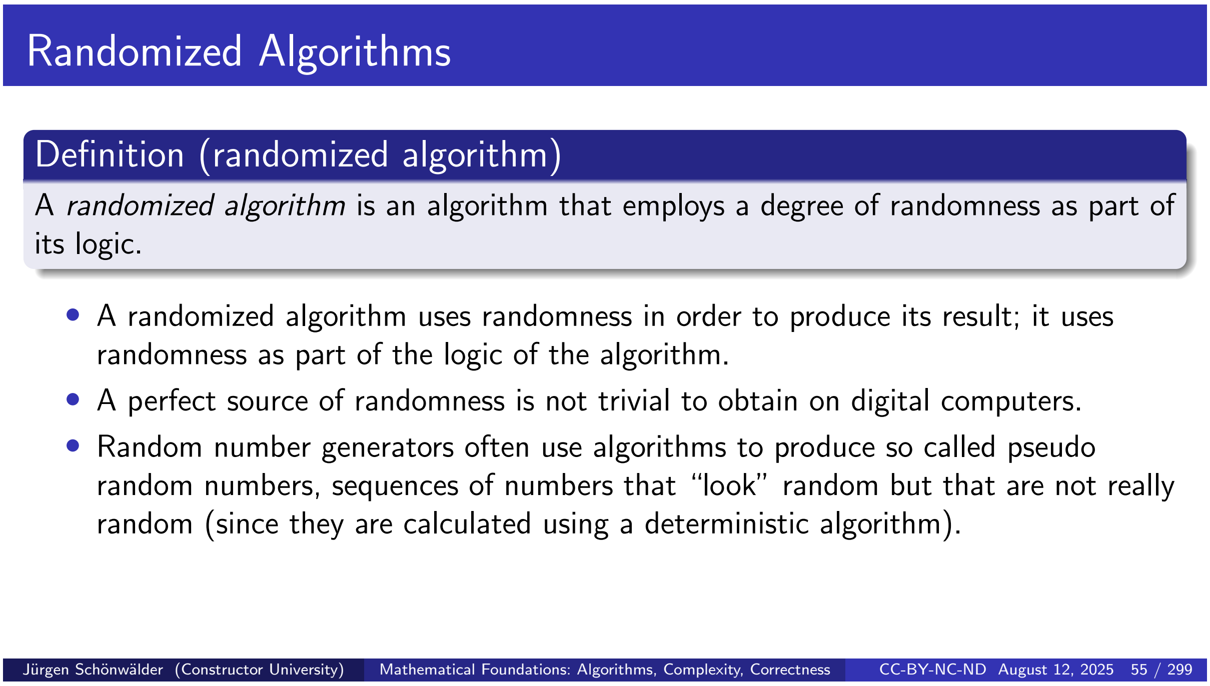
A *concurrent algorithm* is an algorithm in which multiple activities are in progress simultaneously.

Definition (parallel algorithms)

A *parallel algorithm* is an algorithm in which multiple activities are executed simultaneously.

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.
- Concurrency can exist without parallelism but parallelism inheritantly involves concurrency.

Concurrent algorithms are a great opportunity to explore modern hardware, which is often capable to perform multiple computations simultaneously. Modern computers are getting faster by increasing the number of CPU cores and to make effective use of this, it is desirable to design concurrent algorithms that can benefit from execution parallelism.



Randomized Algorithms

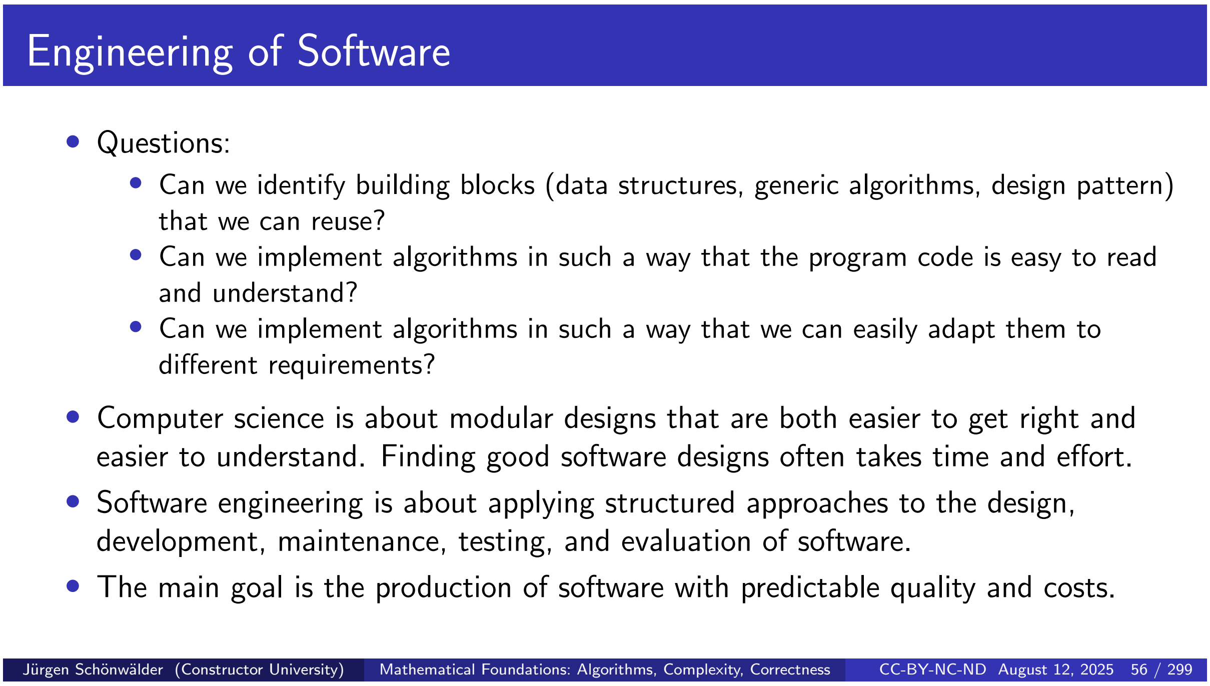
Definition (randomized algorithm)

A *randomized algorithm* is an algorithm that employs a degree of randomness as part of its logic.

- A randomized algorithm uses randomness in order to produce its result; it uses randomness as part of the logic of the algorithm.
- A perfect source of randomness is not trivial to obtain on digital computers.
- Random number generators often use algorithms to produce so called pseudo random numbers, sequences of numbers that "look" random but that are not really random (since they are calculated using a deterministic algorithm).

Randomized algorithms are sometimes desirable, for example to create cryptographic keys or to drive computer games. For some problems, some randomized algorithms provide solutions faster than deterministic solutions. The question for which classes of problems randomized algorithms provide an advantage is an import question investigated in theoretical computer science.

Pseudo random numbers are commonly provided as library functions (e.g., the `long random(void)` function of the C library). You have to be very careful with the usage of these pseudo random numbers. A common problem is that not all bits of a random number have the same degree of "randomness".



Engineering of Software

- Questions:
  - Can we identify building blocks (data structures, generic algorithms, design pattern) that we can reuse?
  - Can we implement algorithms in such a way that the program code is easy to read and understand?
  - Can we implement algorithms in such a way that we can easily adapt them to different requirements?
- Computer science is about modular designs that are both easier to get right and easier to understand. Finding good software designs often takes time and effort.
- Software engineering is about applying structured approaches to the design, development, maintenance, testing, and evaluation of software.
- The main goal is the production of software with predictable quality and costs.

Software engineering is the application of engineering to the development of software in a systematic method. Another definition says that software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Further online information:

- **Wikipedia**: Software Engineering

Part II

Proofs, Sets, Relations, Functions

This part introduces basic elements of discrete mathematics that are essential for many courses in computer science. We start introducing basic terminology and then discuss basic methods to construct proofs. Afterwards, we discuss sets, relations and finally functions.

Recommended background reading are the first five chapters of the book “Mathematics for Computer Science” by Eric Lehmann, F. Thomson Leighton, and Albert R. Meyer [16]. Students interested to dive deeper into proof methods may want to study Richard Hammack’s “Book of Proof” [8]. Both books are available online and in print.

By the end of this part, students should be able to

- understand the role of axioms, theorems, lemmata, and corollaries;
- recall the Peano axioms defining natural numbers;
- familiar with mathematical notation and the notion of predicates and quantifiers;
- use different techniques to construct mathematical proofs;
- construct induction proofs over inductively defined sets;
- use sets and basic set operations to model scenarios;
- understand the concept and properties of relations;
- recognize equivalence and (strict) partial order relations;
- interpret functions as special relations;
- describe basic operations on functions;
- decide whether functions are injective, surjective, or bijective;
- explain Lambda notation of functions and currying.

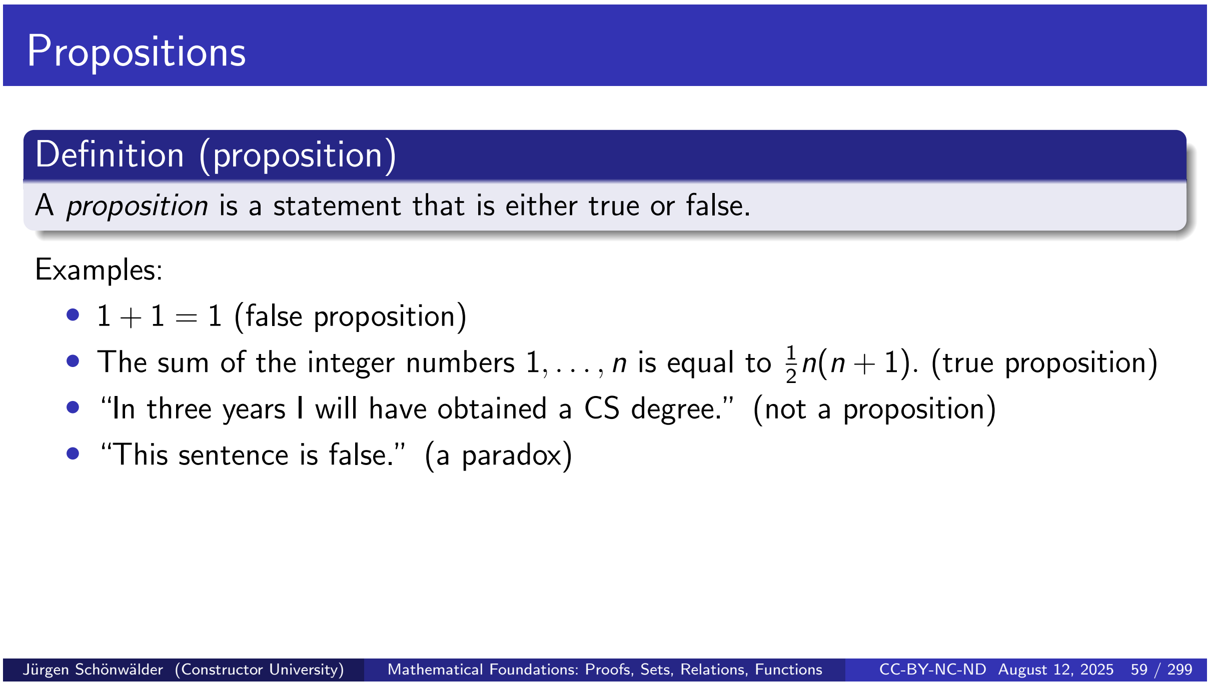
Section 5: Proofs

5 Proofs

6 Sets

7 Relations

8 Functions



Propositions

Definition (proposition)

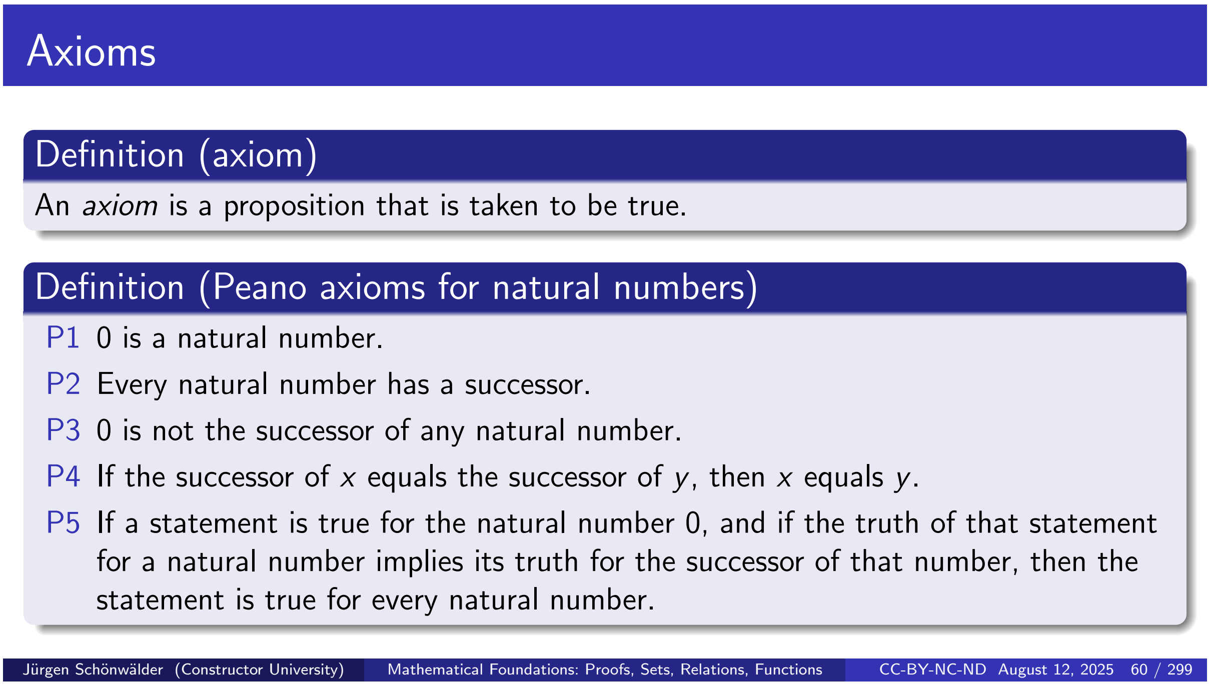
A *proposition* is a statement that is either true or false.

Examples:

- $1 + 1 = 1$ (false proposition)
- The sum of the integer numbers $1, \ldots, n$ is equal to $\frac{1}{2}n(n+1)$. (true proposition)
- “In three years I will have obtained a CS degree.” (not a proposition)
- “This sentence is false.” (a paradox)

A key property is that a proposition is either true or false. Every statement that is only true or false in a certain context is not a proper proposition. In addition, any statement that depends on something undefined (e.g., something happening in the future) is not a proposition.

Beware that there can be logically self-contradictory statements. This has bothered mathematicians for a long time but for now we keep things simple by ignoring the fact that there can be paradoxes.



Axioms

Definition (axiom)

An *axiom* is a proposition that is taken to be true.

Definition (Peano axioms for natural numbers)

P1 0 is a natural number.

P2 Every natural number has a successor.

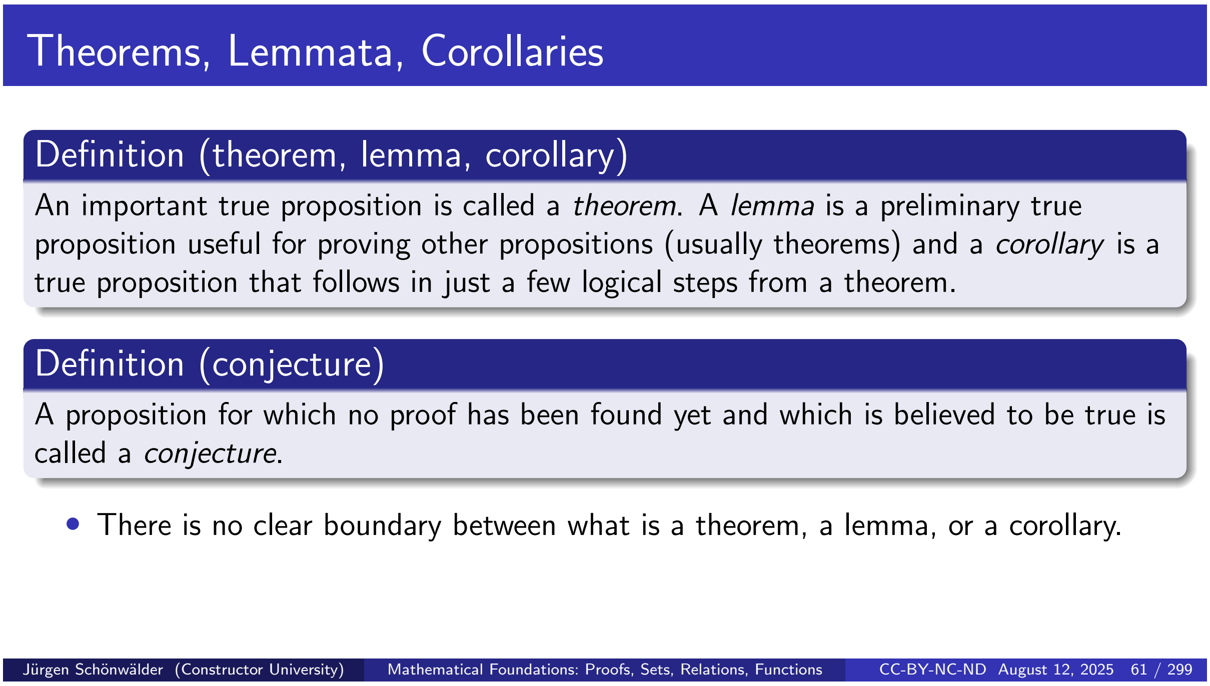
P3 0 is not the successor of any natural number.

P4 If the successor of x equals the successor of y, then x equals y.

P5 If a statement is true for the natural number 0, and if the truth of that statement for a natural number implies its truth for the successor of that number, then the statement is true for every natural number.

When developing a theory and using formal proofs, it is important to be clear about the underlying axioms that are used. Ideally, a small number of well defined axioms are sufficient to develop and proof a complex theory. Finding a minimal set of axioms that are sufficient to derive all knowledge of a certain theory is an important part of research.

The five Peano axioms, defined in 1889 by Giuseppe Peano, are sufficient to derive everything we know about natural numbers. The fifths Peano axiom is interesting since it allows us to prove a statement for all natural numbers even though there are infinitely many natural numbers. We will make use of this technique, called induction, frequently.



Theorems, Lemmata, Corollaries

Definition (theorem, lemma, corollary)

An important true proposition is called a *theorem*. A *lemma* is a preliminary true proposition useful for proving other propositions (usually theorems) and a *corollary* is a true proposition that follows in just a few logical steps from a theorem.

Definition (conjecture)

A proposition for which no proof has been found yet and which is believed to be true is called a *conjecture*.

- There is no clear boundary between what is a theorem, a lemma, or a corollary.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Proofs, Sets, Relations, Functions CC-BY-NC-ND August 12, 2025 61 / 299

Theorem 1 (Fermat's last theorem)**.** *There are no positive integers x, y, and z such that*

$$x^n + y^n = z^n$$

for some integer $n > 2$.

Fermat claimed to have a proof for this conjecture in 1630 but he had not enough space on the margin of the book he was reading to write it down. Fermat's last theorem was finally proven to be true by Andrew Wiles in 1995. Sometimes it takes time to fully work out a proof.

Predicates

- A predicate is a statement that may be true or false depending on the values of its variables. It can be thought of as a function that returns a value that is either true or false.
- Variables appearing in a predicate are often quantified:
  - A predicate is true for all values of a given set of values.
  - A predicate is true for at least one value of a given set of values. (There exists a value such that the predicate is true.)
- There may be multiple quantifiers and they may be combined (but note that the order of the quantifiers matters).
- Example: (Goldbach's conjecture) For every even integer n greater than 2, there exists primes p and q such that $n = p + q$.

Human language is often ambiguous. The statement "Every American has a dream." can be interpreted in two different ways:

a) There exists a dream d out of the set of all dreams D and forall persons a out of the set of Americans A, person a has dream d.

b) Forall persons a out of the set of Americans A, there exists a dream d out of the set of all dreams D such that persons a has dream d.

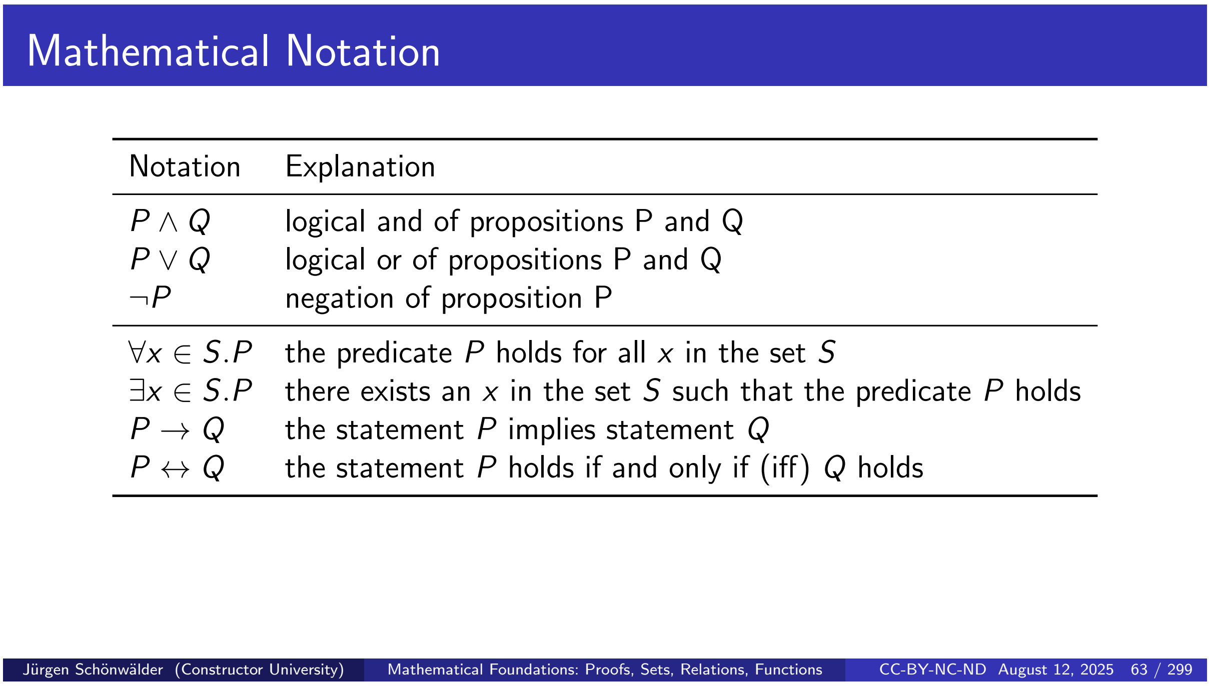
Our common sense says that b) is the more likely interpretation but for machines, which lack a notion of common sense, such ambiguities are difficult to work with. (And this makes natural language processing difficult for computers. Machine learning helps here as it provides a statistical basis to determine the *likely meaning* of a natural language expression.) In mathematics and computer science, we try hard to avoid ambiguities.

Note that predicates can be simple or more complex and as a consequence we have different logics:

- Propositional logic (or zeroth-order logic) deals with simple propositions like "Socrates is a man". Boolean algebra provides the algebraic rules for handling statements in propositional logic.
- Predicate logic (or first-order logic) extends propositional logic with quantified variables like for example "there exists x such that x is Socrates and x is a man".
- Second order logic is even more expressive than first-order logic by quantifying over relations.

For computer scientists, a fundamental question is whether logics are decidable (Entscheidungsproblem): Is there an algorithm that takes a logic statement as input and decides whether it is true or false? For some logics, such an algorithm exists, for others there are proofs that they cannot exist. For those logics where such an algorithm exists, the second question typically concerns the complexity of such an algorithm.

There are special logics such as Horn clauses that have nice bounds on the computational complexity and that have been used as the foundation of logic programming languages like Prolog.



Mathematical Notation

Notation	Explanation
$P \wedge Q$	logical and of propositions P and Q
$P \vee Q$	logical or of propositions P and Q
$\neg P$	negation of proposition P
$\forall x \in S.P$	the predicate P holds for all x in the set S
$\exists x \in S.P$	there exists an x in the set S such that the predicate P holds
$P \rightarrow Q$	the statement P implies statement Q
$P \leftrightarrow Q$	the statement P holds if and only if (iff) Q holds

Some examples:

- Lets formalize the two interpretations of the statement “every American has a dream”. Let A be the set of Americans and D be the set of dreams:

  $$\forall a \in A. \exists d \in D. dream(a, d)$$

  $$\exists d \in D. \forall a \in A. dream(a, d)$$

- Goldbach’s conjecture is stated in mathematical notation as follows:

  Let E be the set all even integers larger than two and P the set of prime numbers. Then the following holds:

  $$\forall n \in E. \exists p \in P. \exists q \in P. n = p + q$$

  Note that $n = p+q$ is a predicate over the variables n, p, and q. Also recall that changing the order of the quantifiers may alter the statement.

- We can write the five Peano axioms in mathematical notation. Lets assume that the function $s : \mathbb{N} \rightarrow \mathbb{N}$ returns the successor of its argument.

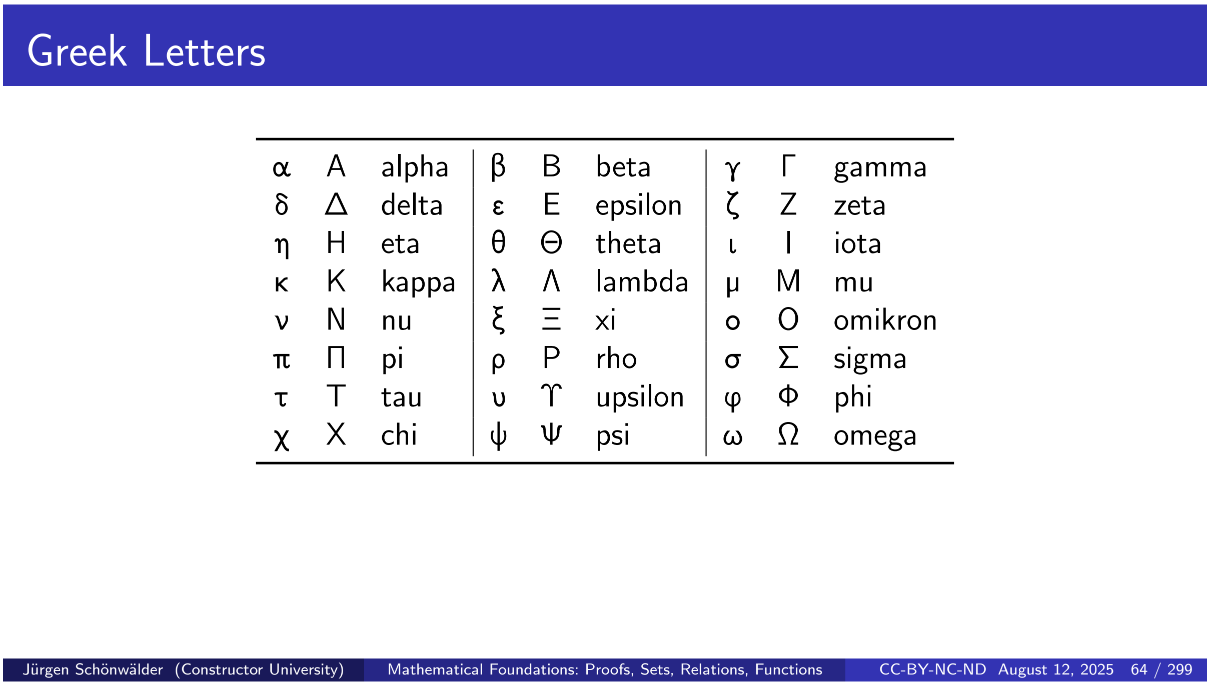
  P1 $0 \in \mathbb{N}$ (zero is a natural number)

  P2 $\forall n \in \mathbb{N}. s(n) \in \mathbb{N} \wedge n \neq s(n)$ (closed under successor, distinct)

  P3 $\neg(\exists n \in \mathbb{N}. 0 = s(n))$ (zero is not a successor)

  P4 $\forall n \in \mathbb{N}. \forall m \in \mathbb{N}. s(n) = s(m) \rightarrow n = m$ (different successors)

  P5 $\forall P.(P(0) \wedge (\forall n \in \mathbb{N}. P(n) \rightarrow P(s(n)))) \rightarrow (\forall m \in \mathbb{N}. P(m))$ (induction)



Greek Letters

α	A	alpha	β	B	beta	γ	Γ	gamma
δ	Δ	delta	ε	E	epsilon	ζ	Z	zeta
η	H	eta	θ	Θ	theta	ι	I	iota
κ	K	kappa	λ	Λ	lambda	μ	M	mu
ν	N	nu	ξ	Ξ	xi	o	O	omikron
π	Π	pi	ρ	P	rho	σ	Σ	sigma
τ	T	tau	υ	Υ	upsilon	φ	Φ	phi
χ	X	chi	ψ	Ψ	psi	ω	Ω	omega

Mathematicians love to use greek letters. And if they run out of greek letters, they love to use roman letters in different writing styles. And to keep reading math fun, every author is free to choose the letters he/she likes best.

We observe the same behavior with young programmers until they realize that reading code written by someone else is way more common than writing code and that things get much simpler if all people within a project follow common conventions, the so called coding styles.

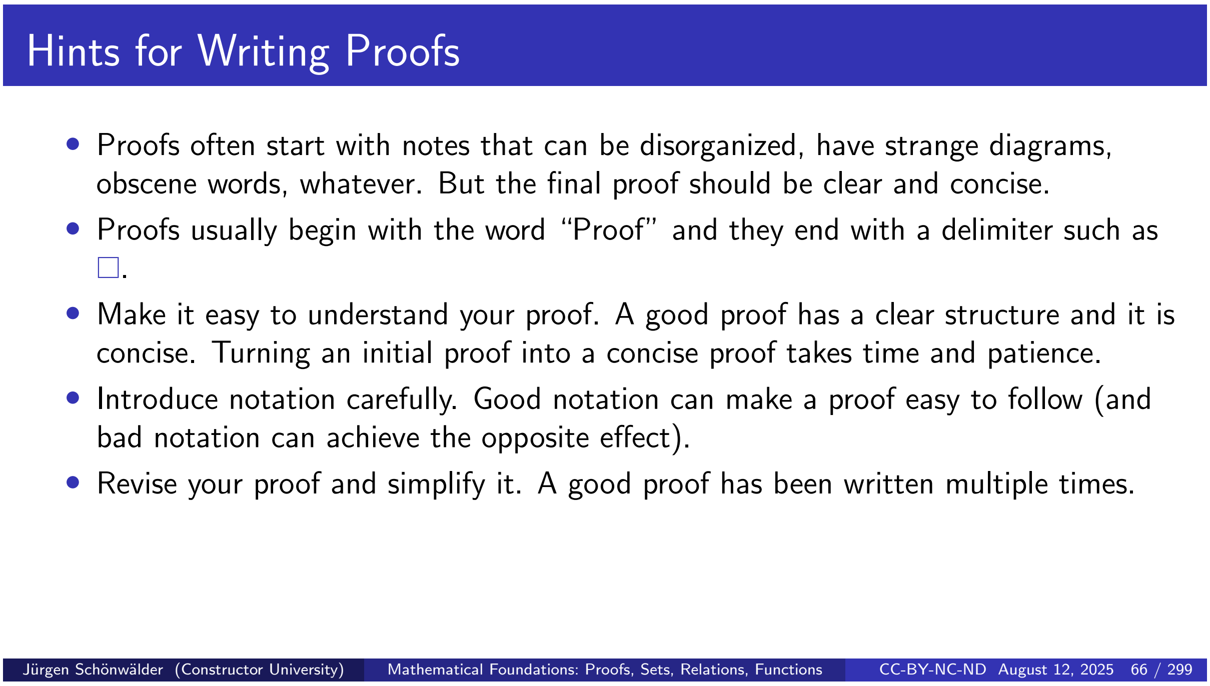
The same applies to mathematicians to some extend. Different areas of math tend to prefer certain notations and writing styles. As a novice mathematician or programmer, the best advice one can give is to follow the conventions that are used by the seniors around you.

Mathematical Proof

Definition (mathematical proof)

A *mathematical proof* of a proposition is a chain of logical deductions from a base set of axioms (or other previously proven propositions) that concludes with the proposition in question.

- Informally, a proof is a method of establishing truth. There are very different ways to establish truth. In computer science, we usually adopt the mathematical notion of a proof.
- There are a certain number of templates for constructing proofs. It is good style to indicate at the beginning of the proof which template is used.



Hints for Writing Proofs

- Proofs often start with notes that can be disorganized, have strange diagrams, obscene words, whatever. But the final proof should be clear and concise.
- Proofs usually begin with the word "Proof" and they end with a delimiter such as □.
- Make it easy to understand your proof. A good proof has a clear structure and it is concise. Turning an initial proof into a concise proof takes time and patience.
- Introduce notation carefully. Good notation can make a proof easy to follow (and bad notation can achieve the opposite effect).
- Revise your proof and simplify it. A good proof has been written multiple times.

Writing good source code is a bit like writing a good proof. Good source code is clear and concise, it has a well-defined structure, it uses a carefully chosen notation, it is easy to read, and it likely has been revised a couple of times. Learning how to write good source code (and good proofs) requires practice. The earlier you start, the faster you get excellent at it. Start right now. Stop producing source code and proofs that are just good enough, instead challenge yourself to produce source code and proofs that are elegant and a little piece of "art" you can be proud of. If you do not know how to distinguish beautiful source code and proofs from just average stuff, start reading other people's source code and proofs. Learn from how they are doing things, ask yourself what you like about what you read and what you find perhaps irritating or difficult. Think how things could have been done differently.

With compiled programming languages, the compiler verifies that our source code is well formed and complete. In traditional mathematics, proofs are checked by humans. Obviously, this manual approach does not scale well and opens the floor to oversights. Modern mathematicians therefore often fully formalize proofs so that tools can verify proofs for correctness and completeness. An example tool for this purpose is the Lean programming language and theorem prover.

Further online information:

- **Web**: Programming Language and Theorem Prover — Lean

Prove an Implication by Derivation

- An implication is a proposition of the form “If P, then Q”, or $P \to Q$.
- One way to prove such an implication is by a derivation where you start with P and stepwise derive Q from it.
- In each step, you apply theorems (or lemmas or corollaries) that have already been proven to be true.
- Template:

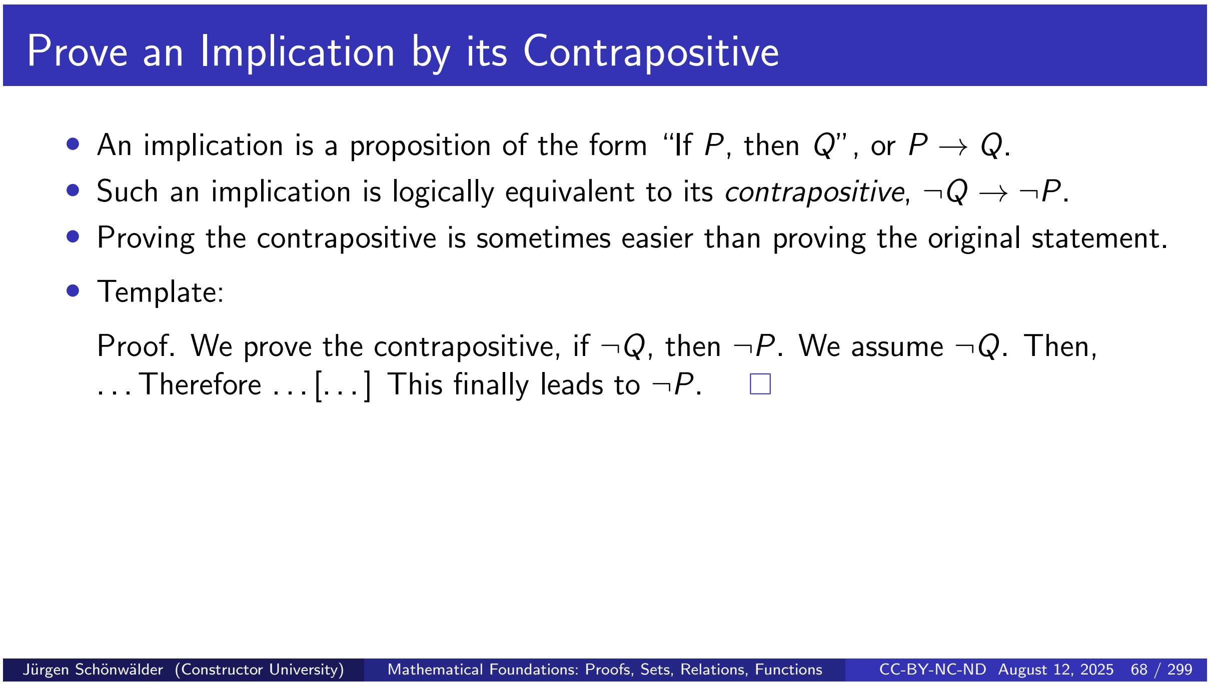
  Assume P. Then, ... Therefore ... [...] This finally leads to Q. □

Theorem 2. *Let x and y be two integers. If x and y are both odd, then the product xy is odd.*

Proof. Assume x and y are two odd integers. We can write x as $x = 2a + 1$ and y as $y = 2b + 1$ with suitable integers a and b. With this, we can write the product of x and y as follows:

$$\begin{aligned} xy &= (2a+1)(2b+1) \\ &= 4ab + 2a + 2b + 1 \\ &= 2(2ab + a + b) + 1 \end{aligned}$$

Since $2(2ab + a + b)$ is even and we add 1 to it, it follows that the product of x and y is odd. □



Prove an Implication by its Contrapositive

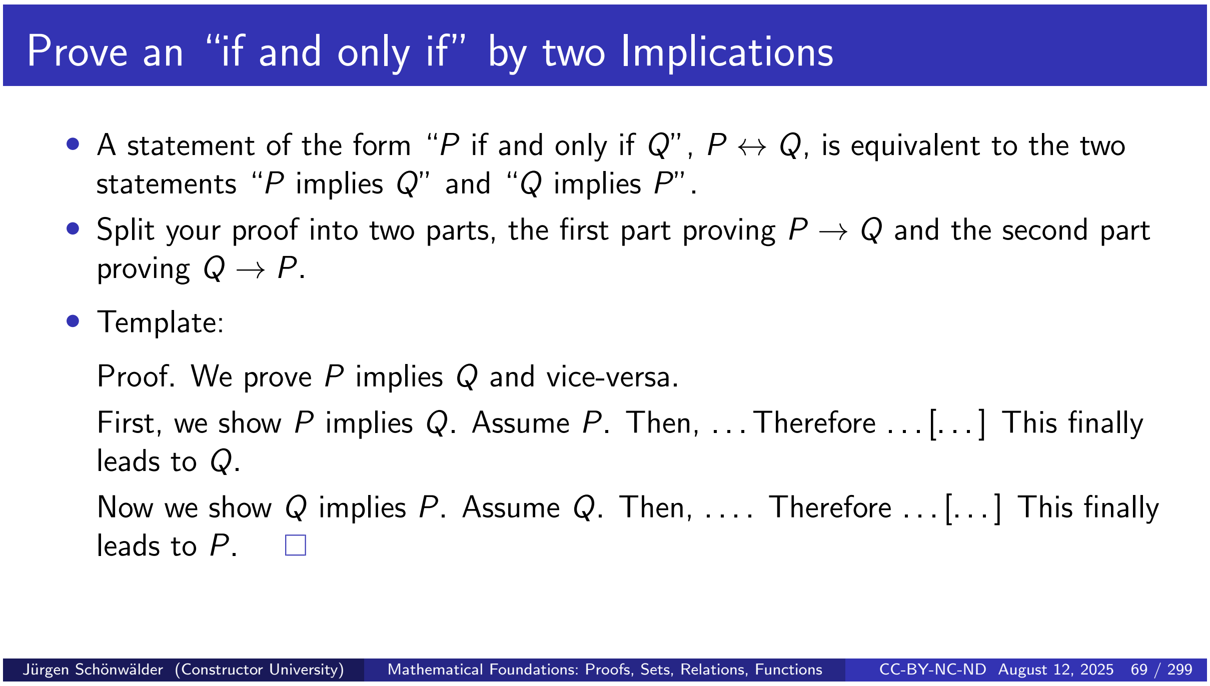
- An implication is a proposition of the form “If P, then Q”, or $P \rightarrow Q$.
- Such an implication is logically equivalent to its *contrapositive*, $\neg Q \rightarrow \neg P$.
- Proving the contrapositive is sometimes easier than proving the original statement.
- Template:

  Proof. We prove the contrapositive, if $\neg Q$, then $\neg P$. We assume $\neg Q$. Then, . . . Therefore . . . [. . .] This finally leads to $\neg P$. □

Theorem 3. *Let x be an integer. If x^2 is even, then x is even.*

Proof. We prove the contrapositive, if x is not even, then x^2 is odd. Assume x is not even. Since the product of two odd numbers results in an odd number (see Theorem 2), it follows that $x^2 = x \cdot x$ is odd. □

Note that the proof above relies on Theorem 2 to be true.



Prove an "if and only if" by two Implications

- A statement of the form "P if and only if Q", $P \leftrightarrow Q$, is equivalent to the two statements "P implies Q" and "Q implies P".
- Split your proof into two parts, the first part proving $P \rightarrow Q$ and the second part proving $Q \rightarrow P$.
- Template:

  Proof. We prove P implies Q and vice-versa.

  First, we show P implies Q. Assume P. Then, ... Therefore ... [...] This finally leads to Q.

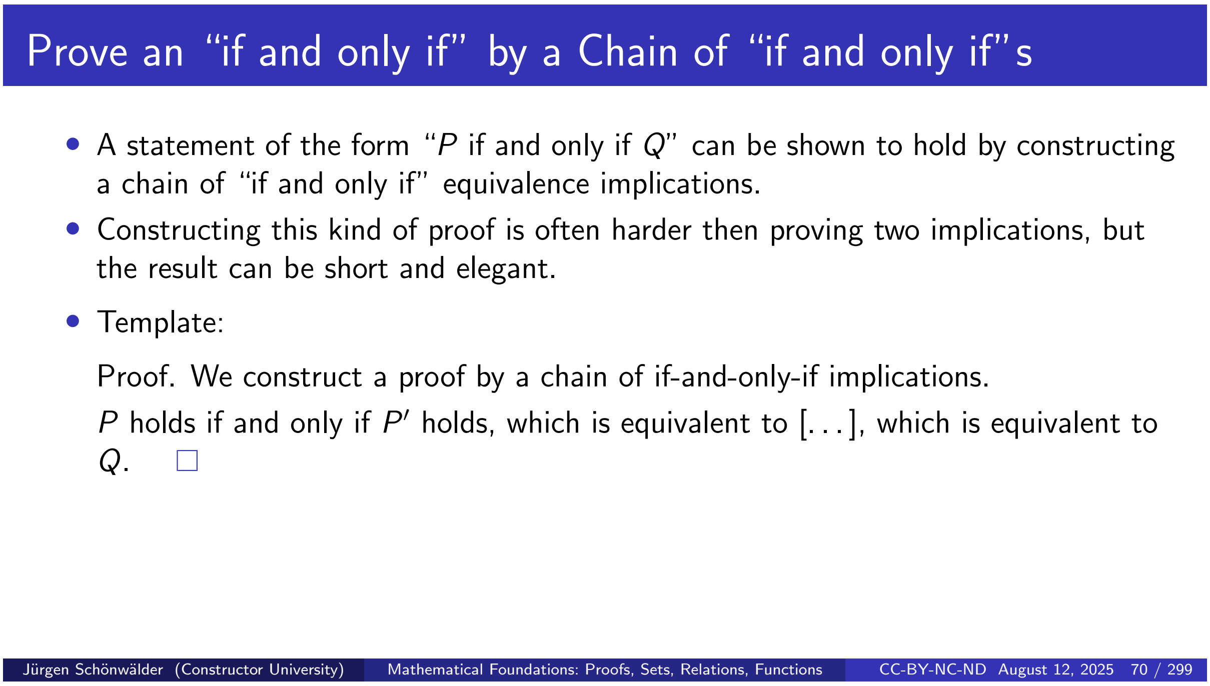
  Now we show Q implies P. Assume Q. Then, ... Therefore ... [...] This finally leads to P. □

Theorem 4. *An integer x is even if and only if its square x^2 is even.*

Proof. We prove x is even implies x^2 is even and vice versa. First, we show that if x is even, then x^2 is even. Since x is even, it can be written as $x = 2k$ for some suitable number k. With this, we obtain $x^2 = (2k)^2 = 2(2k^2)$. Hence, x^2 is even.

Next, we show that if x^2 is even, then x is even. Since x^2 is even, we can write it as $x^2 = 2k$ for some suitable number k, i.e., two divides x^2. This means that two divides either x or x, which implies that x is even. □

This approach to prove an equivalence can be extended. Suppose that you have to show that $A \leftrightarrow B$, $B \leftrightarrow C$ and $C \leftrightarrow A$, then it is sufficient to prove a single chain of implications, namely that $A \rightarrow B$ and $B \rightarrow C$ and $C \rightarrow A$. It is not necessary to prove $B \rightarrow A$ since this follows from $B \rightarrow C \rightarrow A$. Similarly, it is not necessary to prove $C \rightarrow B$ since this follows form $C \rightarrow A \rightarrow B$.



Prove an “if and only if” by a Chain of “if and only if”s

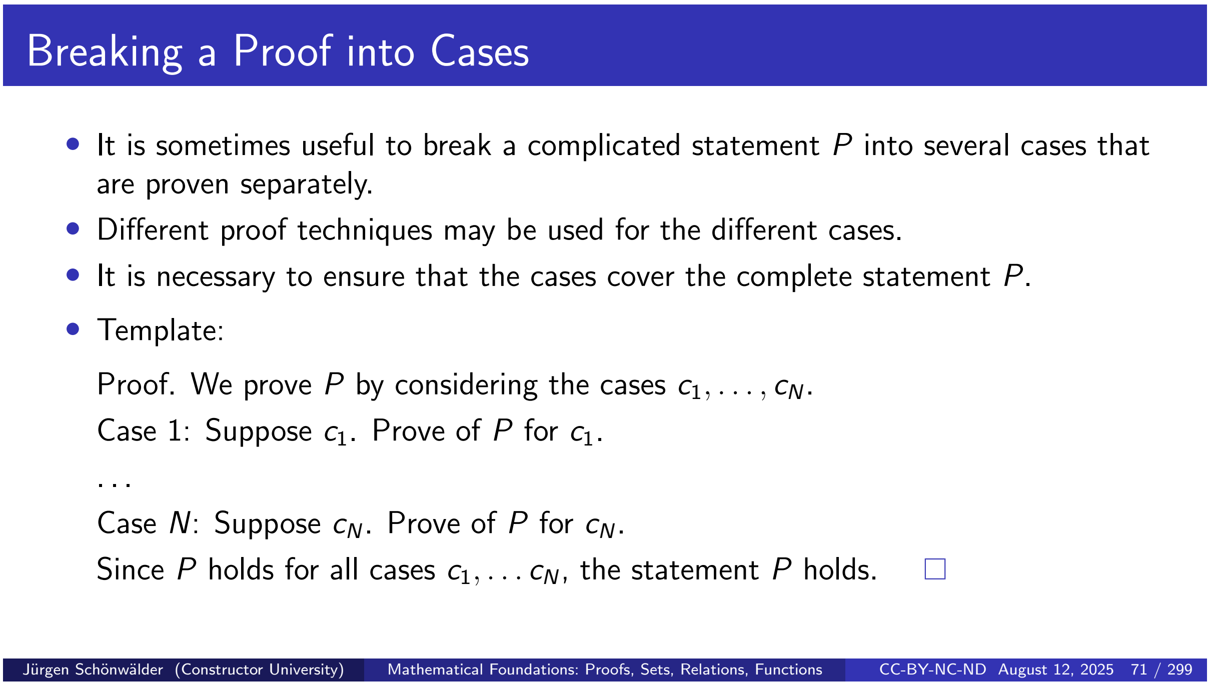
- A statement of the form “P if and only if Q” can be shown to hold by constructing a chain of “if and only if” equivalence implications.
- Constructing this kind of proof is often harder then proving two implications, but the result can be short and elegant.
- Template:

  Proof. We construct a proof by a chain of if-and-only-if implications.

  P holds if and only if P' holds, which is equivalent to $[\ldots]$, which is equivalent to Q. □

Phrases commonly used as alternatives to P “if and only if” Q include:

- Q is necessary and sufficient for P
- P is equivalent (or materially equivalent) to Q
- P precisely if Q
- P exactly when Q
- P just in case Q



Breaking a Proof into Cases

- It is sometimes useful to break a complicated statement P into several cases that are proven separately.
- Different proof techniques may be used for the different cases.
- It is necessary to ensure that the cases cover the complete statement P.
- Template:

  Proof. We prove P by considering the cases $c_1, \ldots, c_N$.

  Case 1: Suppose c_1. Prove of P for c_1.

  ...

  Case N: Suppose c_N. Prove of P for c_N.

  Since P holds for all cases $c_1, \ldots c_N$, the statement P holds. □

Theorem 5. *For every integer $n \in \mathbb{Z}$, $n^2 + n$ is even.*

Proof. We prove $n^2 + n$ is even for all $n \in \mathbb{Z}$ by considering the case where n is even and the case where n is odd.

- Case 1: Suppose n is even:

  n can be written as $2k$ with $k \in \mathbb{Z}$. This gives us:

  $$n^2 + n = (2k)^2 + (2k) = 4k^2 + 2k = 2(2k^2 + k)$$

  Since the result is a multiple of two, it is even.

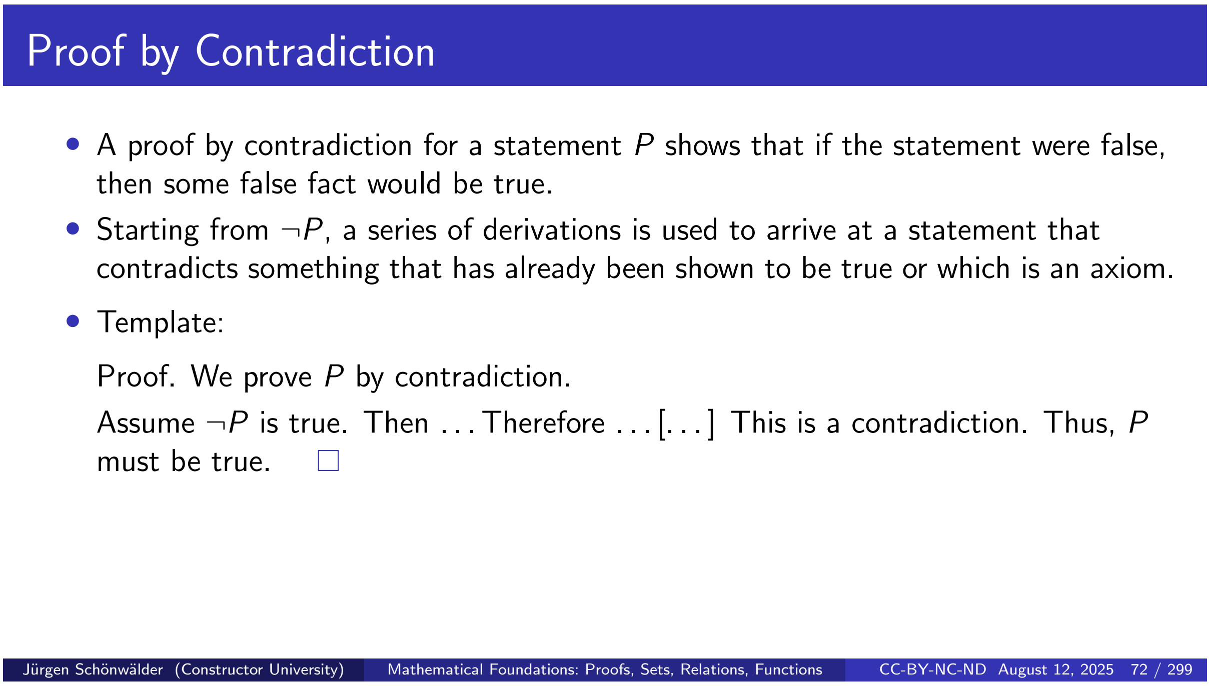
- Case 2: Suppose n is odd:

  n can be written as $2k + 1$ with $k \in \mathbb{Z}$. This gives us:

  $$n^2 + n = (2k+1)^2 + (2k+1) = (4k^2 + 4k + 1) + (2k + 1) = 4k^2 + 6k + 2 = 2(2k^2 + 3k + 1)$$

  Since the result is a multiple of two, it is even.

Since $n^2 + n$ is even holds for the two cases n is even and n is odd, it holds for all $n \in \mathbb{Z}$. □



Proof by Contradiction

- A proof by contradiction for a statement P shows that if the statement were false, then some false fact would be true.
- Starting from $\neg P$, a series of derivations is used to arrive at a statement that contradicts something that has already been shown to be true or which is an axiom.
- Template:

  Proof. We prove P by contradiction.

  Assume $\neg P$ is true. Then …Therefore …[…] This is a contradiction. Thus, P must be true. □

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Proofs, Sets, Relations, Functions CC-BY-NC-ND August 12, 2025 72 / 299

Theorem 6. *$\sqrt{2}$ is irrational.*

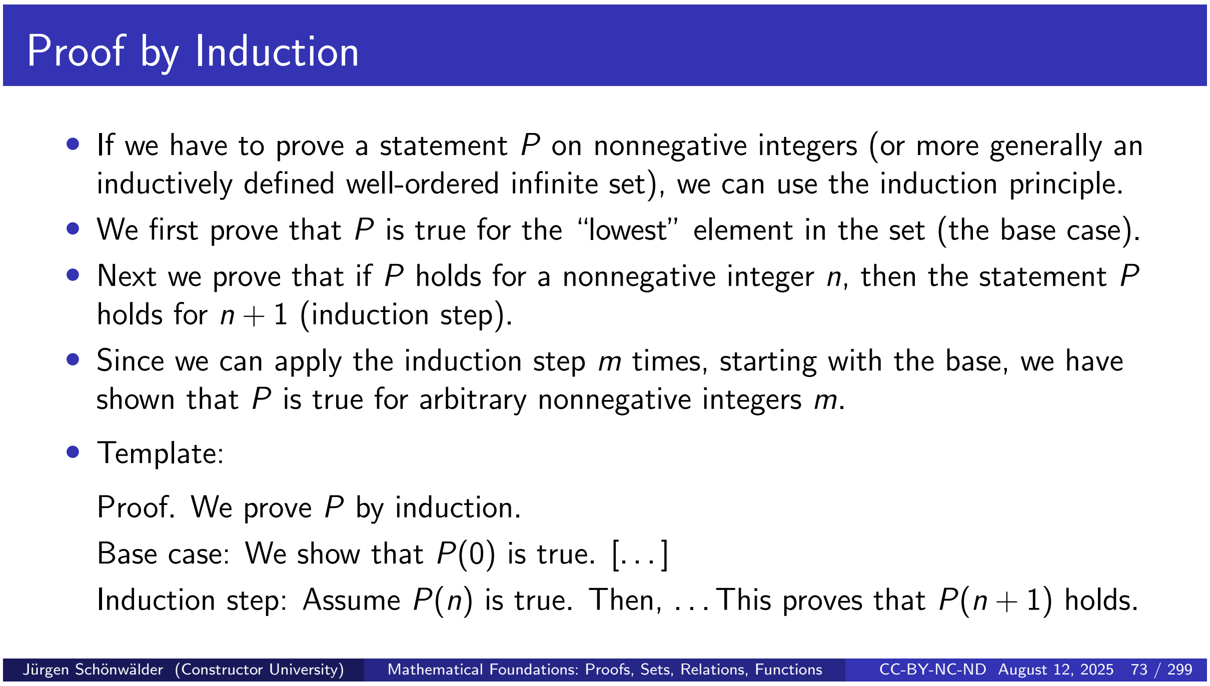
Proof. We use proof by contradiction. Suppose the claim is false and $\sqrt{2}$ is rational. Then we can write $\sqrt{2}$ as a fraction in lowest terms, i.e., $\sqrt{2} = \frac{a}{b}$ with two integers a and b. Since $\frac{a}{b}$ is in lowest terms, at least one of the integers a or b must be odd.

By squaring the equation, we get $2 = \frac{a^2}{b^2}$ which is equivalent to $a^2 = 2b^2$.

Since the square of an odd number is odd (see Theorem 2) and a^2 apparently is even (a multiple of 2), b must be odd.

On the other hand, if a is even, then a^2 is a multiple of 4. If a^2 is a multiple of 4 and $a^2 = 2b^2$, then $2b^2$ is a multiple of 4, and therefore b^2 must be even, and hence b must be even.

Obviously, b cannot be even and odd at the same time. This is a contradiction. Thus, $\sqrt{2}$ must be irrational. □



Proof by Induction

- If we have to prove a statement P on nonnegative integers (or more generally an inductively defined well-ordered infinite set), we can use the induction principle.
- We first prove that P is true for the “lowest” element in the set (the base case).
- Next we prove that if P holds for a nonnegative integer n, then the statement P holds for $n + 1$ (induction step).
- Since we can apply the induction step m times, starting with the base, we have shown that P is true for arbitrary nonnegative integers m.
- Template:

  Proof. We prove P by induction.

  Base case: We show that $P(0)$ is true. [...]

  Induction step: Assume $P(n)$ is true. Then, ... This proves that $P(n + 1)$ holds.

Theorem 7. *For all* $n \in \mathbb{N}$, $0 + 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$.

Proof. We prove $0 + 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$ by induction.

- Base case:

  We show that the equation is true for $n = 0$. Setting $n = 0$, the equation becomes

  $$0 = \frac{0(0+1)}{2} = 0\frac{1}{2} = 0$$

  and hence the equation holds for $n = 0$.

- Induction step:

  Assume that the equation holds for some $n \in \mathbb{N}$. Lets consider the case $n + 1$:

  $$\begin{aligned} 0 + 1 + 2 + 3 + \ldots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{n(n+1) + 2(n+1)}{2} \\ &= \frac{n^2 + n + 2n + 2}{2} \\ &= \frac{(n+2)(n+1)}{2} \end{aligned}$$

  This shows that the equation holds for $n + 1$.

It follows by induction that $0 + 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$ holds for arbitrary nonnegative integers n. □

In computer science, we are frequently interested in structural induction. Instead of proving a proposition over the set of natural numbers, we often prove a proposition over an inductively defined structure (hence the term structural induction). One can view structural induction as a generalization of the mathematical induction over natural numbers. (Natural numbers can be seen as just an example of an inductively defined structure.)

Summary of Proof Techniques

Statement	**Techniques**	**Description**
$A \rightarrow Z$	$A \rightarrow B \rightarrow C \rightarrow \ldots \rightarrow Z$ $\neg Z \rightarrow \neg A$	proof by derivation proof by contrapositive
$A \leftrightarrow Z$	$A \leftrightarrow B \leftrightarrow C \leftrightarrow \ldots \leftrightarrow Z$ $A \rightarrow Z \wedge Z \rightarrow A$	chain of equivalences proof by two implications
A	$\neg A \rightarrow B \rightarrow C \rightarrow \ldots \rightarrow \bot$	proof by contradiction
$\forall n \in \mathbb{N}.A(n)$	$A(0) \wedge (A(n) \rightarrow A(n+1))$	proof by induction

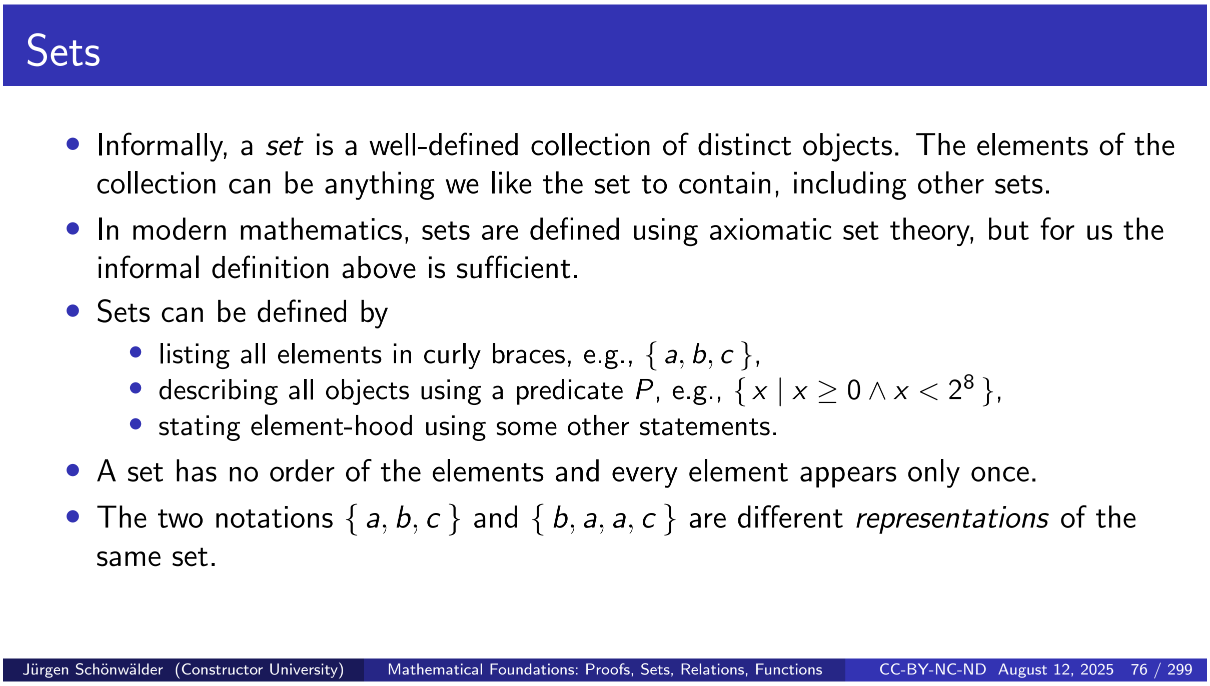
Section 6: Sets

5 Proofs

6 Sets

7 Relations

8 Functions



Sets

- Informally, a *set* is a well-defined collection of distinct objects. The elements of the collection can be anything we like the set to contain, including other sets.
- In modern mathematics, sets are defined using axiomatic set theory, but for us the informal definition above is sufficient.
- Sets can be defined by
  - listing all elements in curly braces, e.g., $\{ a, b, c \}$,
  - describing all objects using a predicate P, e.g., $\{ x \mid x \geq 0 \land x < 2^8 \}$,
  - stating element-hood using some other statements.
- A set has no order of the elements and every element appears only once.
- The two notations $\{ a, b, c \}$ and $\{ b, a, a, c \}$ are different *representations* of the same set.

Some popular sets in mathematics:

symbol	set	elements
$\emptyset$	empty set	$\{\}$
$\mathbb{N}$	nonnegative integers	$\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	integers	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{Q}$	rational numbers	$\frac{1}{2}$, 42, etc.
$\mathbb{R}$	real numbers	π, $\sqrt{2}$, 0 etc.
$\mathbb{C}$	complex numbers	i, $5 + 9i$, 0, π etc.

Sets can be very confusing. A mathematician called Georg Cantor tried to formalize the notion of sets, introducing so called naive set theory. According to naive set theory, any definable collection is a set. Bertrand Russell, another mathematician, discovered a paradox that is meanwhile known as Russell's paradox:

> Let R be the set of all sets that are not members of themselves. If R is not a member of itself, then its definition dictates that it must contain itself, and if it contains itself, then it contradicts its own definition as the set of all sets that are not members of themselves. Symbolically:
>
> Let $R = \{ x \mid x \notin x \}$, then $R \in R \Leftrightarrow R \notin R$.

Another formulation provided by Bertrand Russell and known as the barber paradox:

> You can define a barber as "one who shaves all those, and only those, who do not shave themselves." The question is, does the barber shave himself?

Basic Relations between Sets

Definition (basic relations between sets)

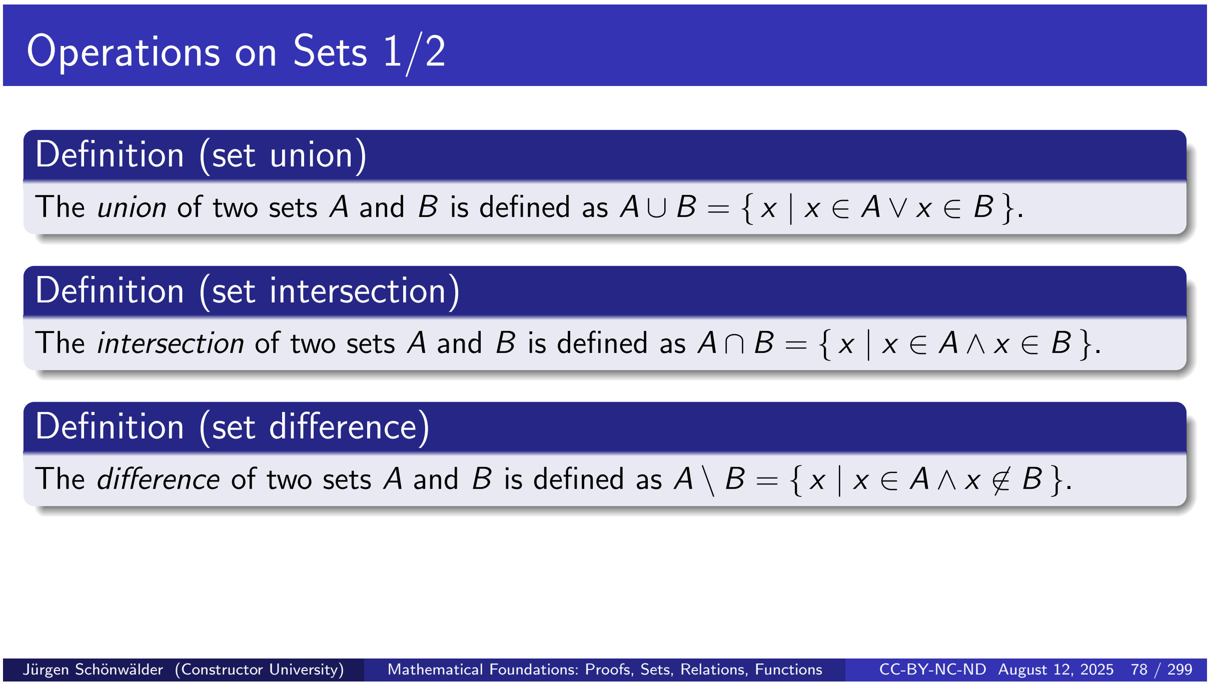
Lets A and B be two sets. We define the following relations between sets:

1. $(A \equiv B) :\leftrightarrow (\forall x . x \in A \leftrightarrow x \in B)$ (set equality)
2. $(A \subseteq B) :\leftrightarrow (\forall x . x \in A \rightarrow x \in B)$ (subset)
3. $(A \subset B) :\leftrightarrow (A \subseteq B) \land (A \not\equiv B)$ (proper subset)
4. $(A \supseteq B) :\leftrightarrow (\forall x . x \in B \rightarrow x \in A)$ (superset)
5. $(A \supset B) :\leftrightarrow (A \supseteq B) \land (A \not\equiv B)$ (proper superset)

- Obviously:
  - $(A \subseteq B) \land (B \subseteq A) \rightarrow (A \equiv B)$
  - $(A \subseteq B) \leftrightarrow (B \supseteq A)$

Apparently, $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ for the popular sets in mathematics.

In the real world, sets and their relations are often not well defined. For example, consider the set representing the faculty of a university. How do you think this set is defined? And is it a proper subset of the set of all employees of the university?



Operations on Sets 1/2

Definition (set union)

The *union* of two sets A and B is defined as $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$.

Definition (set intersection)

The *intersection* of two sets A and B is defined as $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$.

Definition (set difference)

The *difference* of two sets A and B is defined as $A \setminus B = \{\, x \mid x \in A \wedge x \notin B \,\}$.

Some basic properties of set unions:

$A \cup B = B \cup A$	commutativity
$A \cup (B \cup C) = (A \cup B) \cup C$	associativity
$A \subseteq (A \cup B)$	
$A \cup A = A$	idempotency
$A \cup \emptyset = A$	identity
$A \subseteq B \Leftrightarrow A \cup B = B$	

Some basic properties of set intersections:

$A \cap B = B \cap A$	commutativity
$A \cap (B \cap C) = (A \cap B) \cap C$	associativity
$A \cap B \subseteq A$	
$A \cap A = A$	idempotency
$A \cap \emptyset = \emptyset$	
$A \subseteq B \Leftrightarrow A \cap B = A$	

Some basic properties of set differences:

$$A \setminus A = \emptyset$$
$$A \setminus \emptyset = A$$

Some basic properties of combinations of set operations:

$A \cup (A \cap B) = A$	absorption
$A \cap (A \cup B) = A$	absorption
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	distributivity
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	distributivity

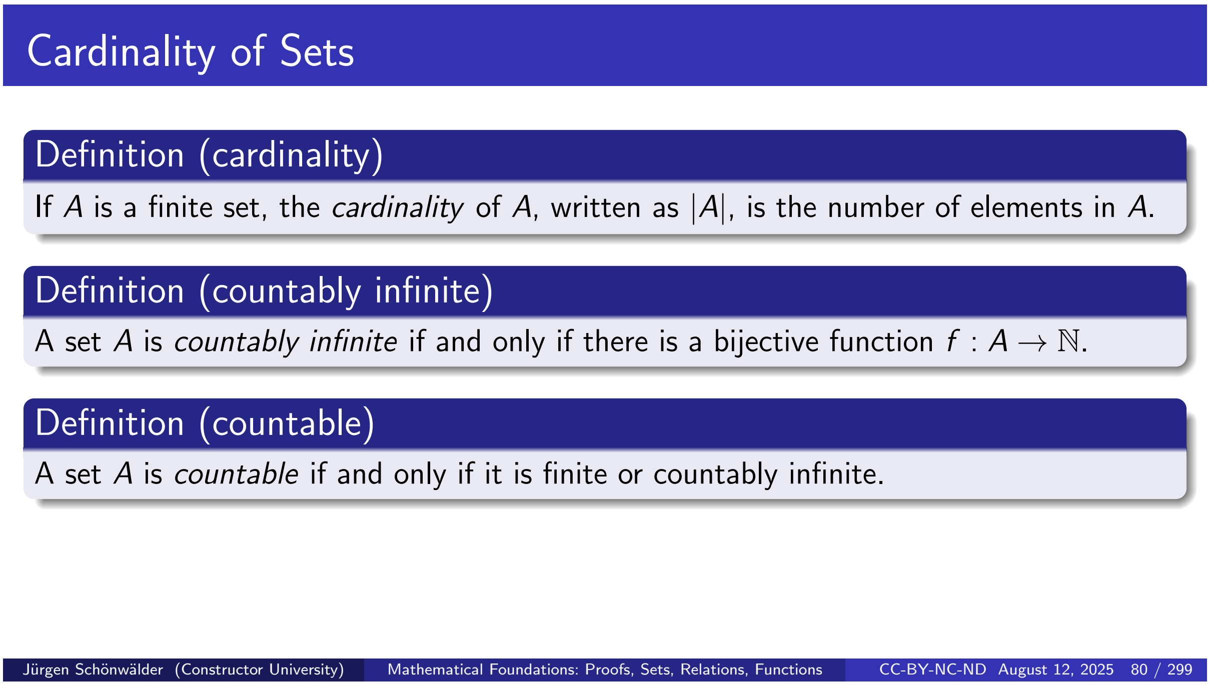
Operations on Sets 2/2

Definition (power set)

The *power set* $\mathcal{P}(A)$ of a set A is the set of all subsets S of A, including the empty set and A itself. Formally, $\mathcal{P}(A) = \{\, S \mid S \subseteq A \,\}$.

Definition (cartesian product)

The *cartesian product* of the sets $X_1, \ldots, X_n$ is defined as $X_1 \times \ldots \times X_n = \{\, (x_1, \ldots, x_n) \mid \forall i \in \{1, \ldots, n\}. x_i \in X_i \,\}$.



Cardinality of Sets

Definition (cardinality)

If A is a finite set, the *cardinality* of A, written as $|A|$, is the number of elements in A.

Definition (countably infinite)

A set A is *countably infinite* if and only if there is a bijective function $f : A \to \mathbb{N}$.

Definition (countable)

A set A is *countable* if and only if it is finite or countably infinite.

Theorem 8. *If S is a finite set with $|S| = n$ elements, then the number of subsets of S is $|\mathcal{P}(S)| = 2^n$.*

Theorem 9. *Let A and B be two finite sets. Then the following holds:*

1. $|(A \cup B)| \leq |A| + |B|$
2. $|(A \cap B)| \leq \min(|A|, |B|)$
3. $|(A \times B)| = |A| \cdot |B|$

There are sets that are not countable, so called uncountable sets. The best known example of an uncountable set is the set $\mathbb{R}$ of all real numbers. Cantors diagonal argument, published in 1891, is a famous proof that there are infinite sets that can not be counted.

Section 7: Relations

5 Proofs

6 Sets

7 Relations

8 Functions

Relations

Definition (relation)

A *relation* R over the sets $X_1, \dots, X_k$ is a subset of their Cartesian product, written $R \subseteq X_1 \times \dots \times X_k$.

- Relations are classified according to the number of sets in the defining Cartesian product:
  - A unary relation is defined over a single set X
  - A binary relation is defined over $X_1 \times X_2$
  - A ternary relation is defined over $X_1 \times X_2 \times X_3$
  - A k-ary relation is defined over $X_1 \times \dots \times X_k$

We do not really need ternary, . . . , k-ary relations. For example, we can view a ternary relation $A \times B \times C$ as a binary relation $A \times (B \times C)$. Hence, we will focus on binary relations.

Relations are a fairly general concept. Relations play an important role while developing data models for computer applications. There is a class of database systems, the so called relational database management systems (RDMS), that are based on a formal relational model. The idea is to model a domain as a collection of relations that can be represented efficiently as database tables.

Entity relationship models describe a part of a world as sets of typed objects (entities), relations between entities, and attributes of entities or relations. An example for a system like CampusNet:

- Entities:
  - $Student$
  - $Instructor$
  - $Course$
  - $Module$
  - $Person$
  - $\dots$
- Relations:
  - $enrolled\_in \subseteq (Student \times Course)$
  - $is\_a \subseteq (Student \times Person)$
  - $is\_a \subseteq (Instructor \times Person)$
  - $belongs\_to \subseteq (Course \times Module)$
  - $teaches \subseteq (Instructor \times Course)$
  - $tutor\_of \subseteq (Student \times Course)$
  - $\dots$

Entity relationship models are often defined using a graphical notation. A standard graphical notation is part of the Unified Modeling Language (UML).

Figure 1 provides a UML diagram representation of relations in a campus information system. The symbols used in UML diagrams have well defined semantics. We do not go into the details here. For

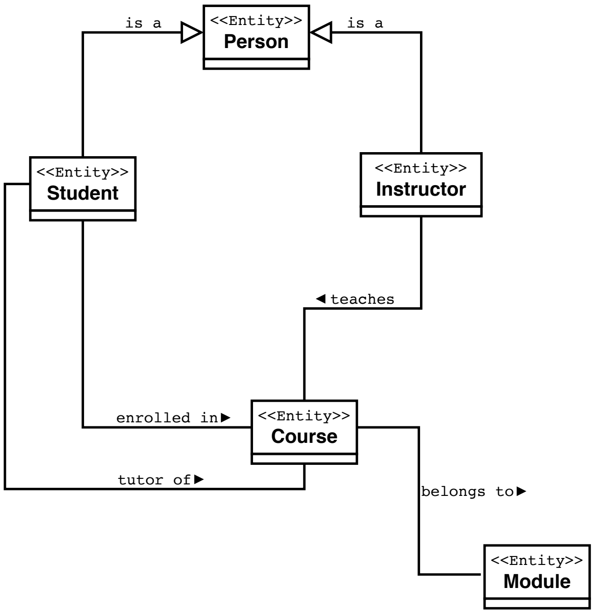
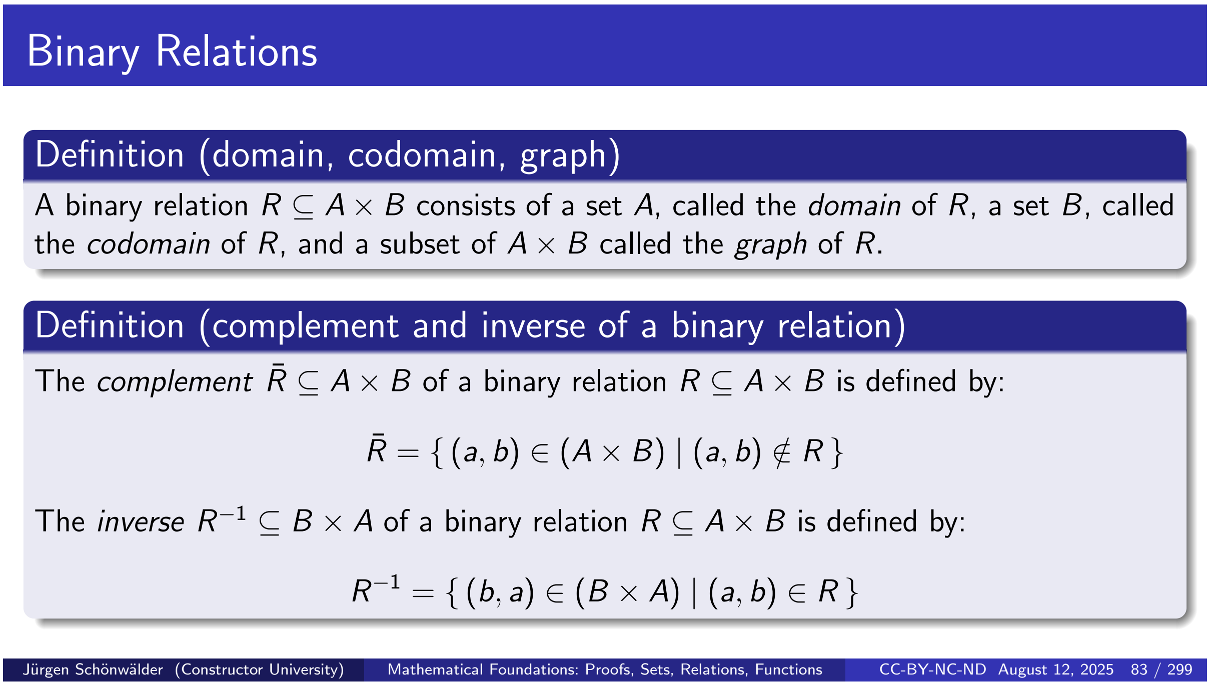


Figure 1: A UML diagram showing relations using a graphical notation

now, just note that special line ends (unfilled arrow heads) are used for is_a relations. The reason is that is_a relations are quite common and programming languages or database systems often have special support for representing is_a relations.



Binary Relations

Definition (domain, codomain, graph)

A binary relation $R \subseteq A \times B$ consists of a set A, called the *domain* of R, a set B, called the *codomain* of R, and a subset of $A \times B$ called the *graph* of R.

Definition (complement and inverse of a binary relation)

The *complement* $\bar{R} \subseteq A \times B$ of a binary relation $R \subseteq A \times B$ is defined by:

$$\bar{R} = \{\, (a, b) \in (A \times B) \mid (a, b) \notin R \,\}$$

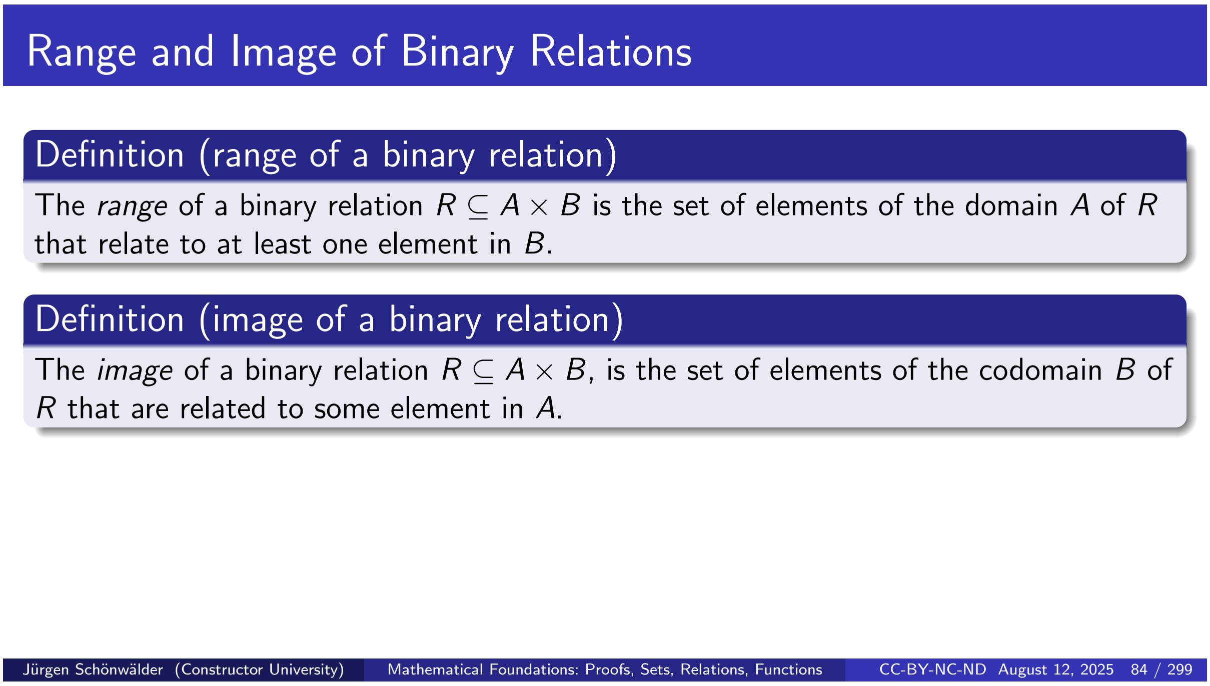
The *inverse* $R^{-1} \subseteq B \times A$ of a binary relation $R \subseteq A \times B$ is defined by:

$$R^{-1} = \{\, (b, a) \in (B \times A) \mid (a, b) \in R \,\}$$

Let $R \subseteq (A \times B)$ be a relation. For $(a, b) \in R$ we may simply write $R(a, b)$ or $a\,R\,b$. The notation $a\,R\,b$ is called *infix notation* while the notation $R(a, b)$ is called the *prefix notation*.

A good example is the relation of students with their teaching assistants. Let S be the set of students in this course and let T be the set of teaching assistants of this course. Then *is_assigned_to* is a binary relation over $S \times T$ and S is the domain and T is the codomain of this relation. We sometimes use the notation $dom(R)$ and $codom(R)$ to refer to the domain and the codomain of a relation R.

If Julia is a teaching assistant and Aydin a student and Aydin is assigned to Julia, then we can write $is\_assigned\_to(Aydin, Julia)$ or alternatively $Aydin\; is\_assigned\_to\; Julia)$.



Range and Image of Binary Relations

Definition (range of a binary relation)

The *range* of a binary relation $R \subseteq A \times B$ is the set of elements of the domain A of R that relate to at least one element in B.

Definition (image of a binary relation)

The *image* of a binary relation $R \subseteq A \times B$, is the set of elements of the codomain B of R that are related to some element in A.

For small binary relations, it is possible to draw relation diagrams, with nodes representing the domain on the left side, nodes representing the codomain on the right side, and arrows representing the relation, pointing from the domain nodes to the codomain nodes.

As an example, consider $R \subseteq A \times B$ with $A = \{a, b, c, d, e, f\}$ and $B = \{1, 2, 3, 4, 5\}$. We define R using the graphviz dot notation:

```
digraph R {
        a -> 1;
        b -> 3;
        c -> 4;
        d -> 2;
        e -> 3;
}
```

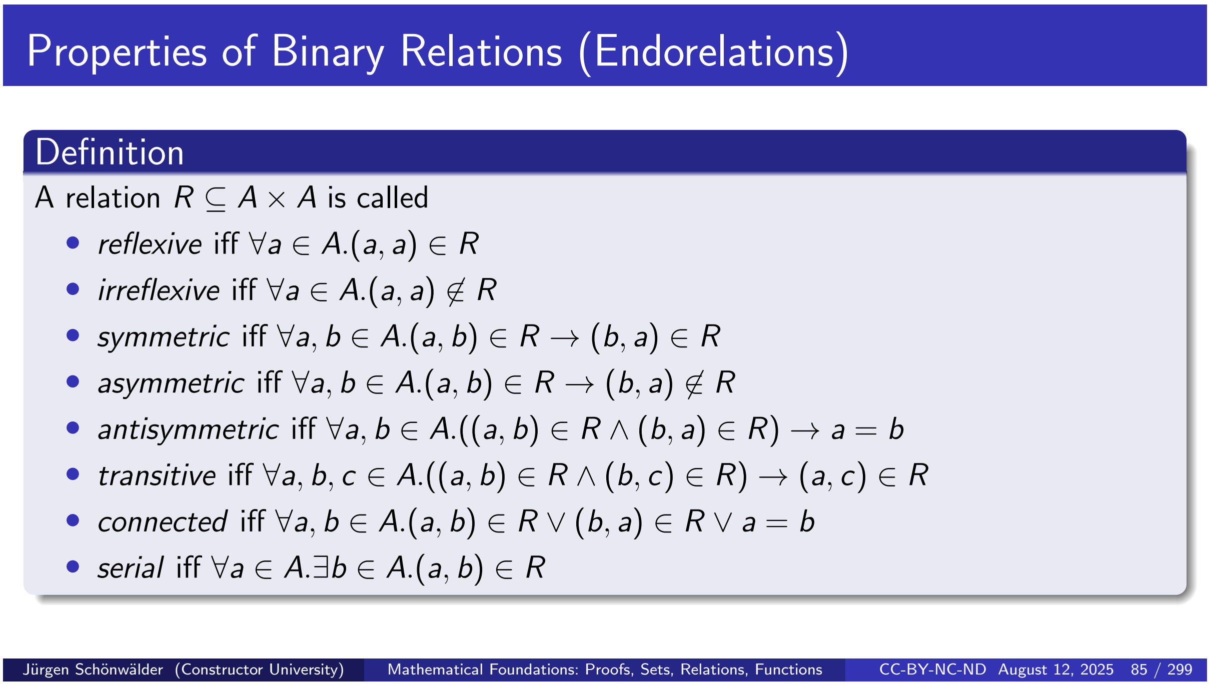
The range of R is $\{a, b, c, d, e\}$ and the image of R is $\{1, 2, 3, 4\}$.

The inverse R^{-1} of R defined in the graphviz dot notation:

```
digraph Rinv {
        1 -> a;
        3 -> b;
        4 -> c;
        2 -> d;
        3 -> e;
}
```

The complement $\bar{R}$ of R defined in the graphviz dot notation:

```
digraph Rbar {
        a -> 2; a -> 3; a -> 4; a -> 5;
        b -> 1; b -> 2; b -> 4; b -> 5;
        c -> 1; c -> 2; c -> 3; c -> 5;
        d -> 1; d -> 3; d -> 4; d -> 5;
        e -> 1; e -> 2; e -> 4; e -> 5;
        f -> 1; f -> 2; f -> 3; f -> 4; f -> 5;
}
```



Properties of Binary Relations (Endorelations)

Definition

A relation $R \subseteq A \times A$ is called

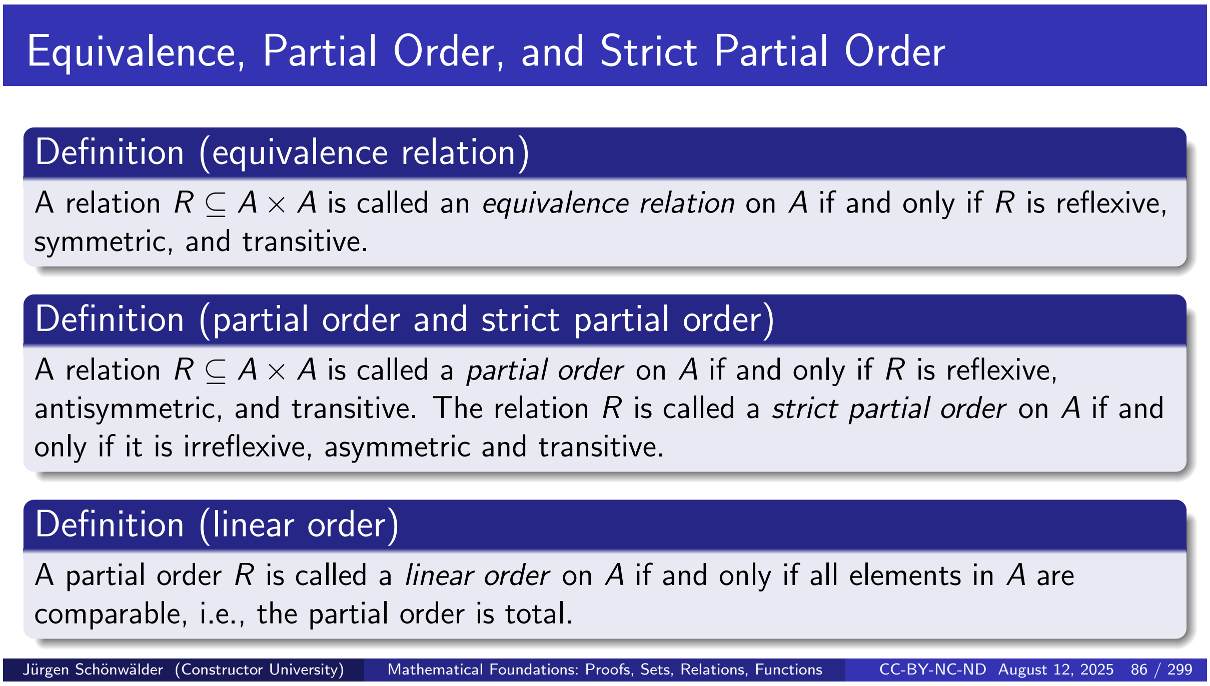
- *reflexive* iff $\forall a \in A.(a, a) \in R$
- *irreflexive* iff $\forall a \in A.(a, a) \notin R$
- *symmetric* iff $\forall a, b \in A.(a, b) \in R \rightarrow (b, a) \in R$
- *asymmetric* iff $\forall a, b \in A.(a, b) \in R \rightarrow (b, a) \notin R$
- *antisymmetric* iff $\forall a, b \in A.((a, b) \in R \land (b, a) \in R) \rightarrow a = b$
- *transitive* iff $\forall a, b, c \in A.((a, b) \in R \land (b, c) \in R) \rightarrow (a, c) \in R$
- *connected* iff $\forall a, b \in A.(a, b) \in R \lor (b, a) \in R \lor a = b$
- *serial* iff $\forall a \in A.\exists b \in A.(a, b) \in R$

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Proofs, Sets, Relations, Functions CC-BY-NC-ND August 12, 2025 85 / 299

Note: There are relations that are neither reflexive nor irreflexive.

Examples:

- The relation *is_as_old_as* on the set of persons is reflexive.
- The relation *is_older_than* on the set of persons is irreflexive.
- The relation *is_sibling_of* on the set of persons is symmetric.
- The relation *is_mother_of* on the set of persons is asymmetric.
- The relation *is_not_older_as* on the set of persons is antisymmetric.
- The relation *is_ancestor_of* on the set of persons transitive.



Equivalence, Partial Order, and Strict Partial Order

Definition (equivalence relation)

A relation $R \subseteq A \times A$ is called an *equivalence relation* on A if and only if R is reflexive, symmetric, and transitive.

Definition (partial order and strict partial order)

A relation $R \subseteq A \times A$ is called a *partial order* on A if and only if R is reflexive, antisymmetric, and transitive. The relation R is called a *strict partial order* on A if and only if it is irreflexive, asymmetric and transitive.

Definition (linear order)

A partial order R is called a *linear order* on A if and only if all elements in A are comparable, i.e., the partial order is total.

A symbol commonly used for equivalence relations is $\equiv$, a symbol commonly used for strict partial orders is $\prec$, and a symbol commonly used for non-strict partial orders is $\preceq$.

Note: A partial order $\preceq$ induces a strict partial order $a \prec b \Leftrightarrow a \preceq b \land a \neq b$.

Note: A strict partial order $\prec$ induces a partial order $a \preceq b \Leftrightarrow a \prec b \lor a = b$.

Examples:

- The relation *is_as_old_as* is reflexive, symmetric and transitive. Hence it is an equivalence relation.
- The relation *is_not_older_as* on the set of persons is reflexive, antisymmetric, and transitive and hence it is a partial order.
- The relation *is_younger_as* is irreflexive, asymmetric, and transitive and hence it is a strict partial order.

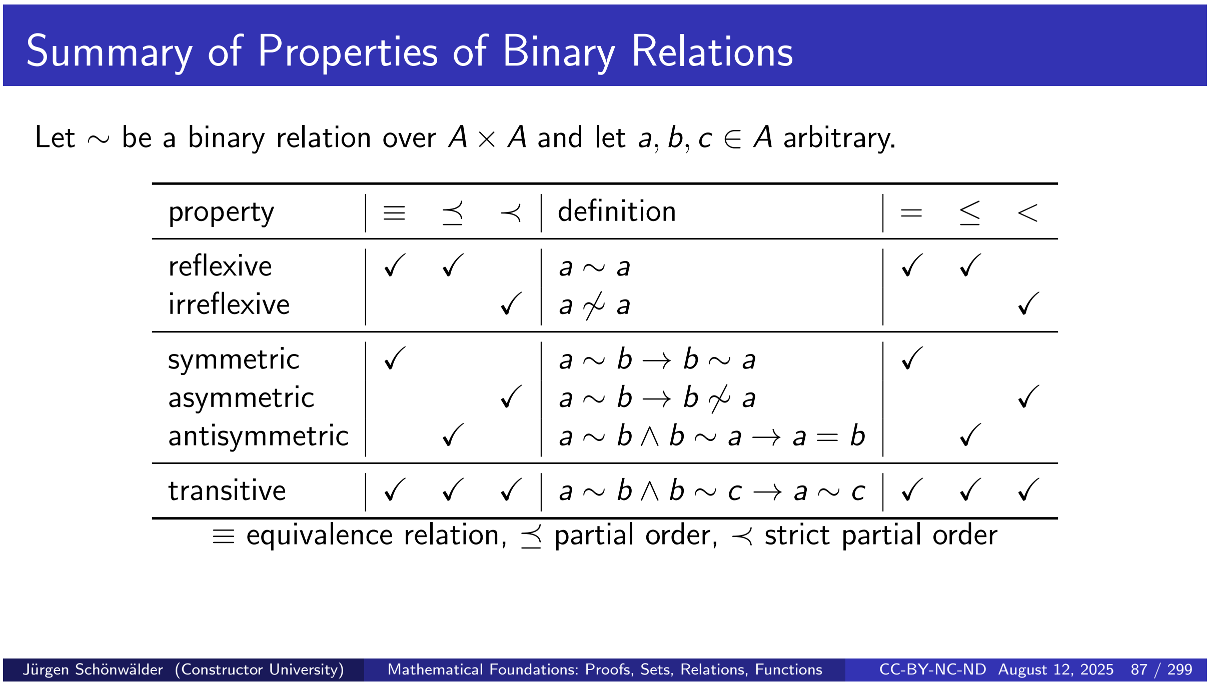
Note: An equivalence relation induces equivalence classes. Given a set of persons, the *is_as_old_as* relation induces classes of persons with the same age.

Another example for a partial order is the following order relation defined on vectors $\vec{x}$ and $\vec{y}$ in $\mathbb{R}^n$:

$$\vec{x} \preceq \vec{y} \Leftrightarrow \forall i \in \{1, \ldots, n\}. x_i \leq y_i$$

This is a partial order since the vectors $\vec{x} = (0, 1)$ and $\vec{y} = (1, 0)$ have no order relationship.

Another example for a partial order is the *happened_before* relation on events in a distributed system, which expresses the fact that an event a that happened before an event b may have influenced the event b.



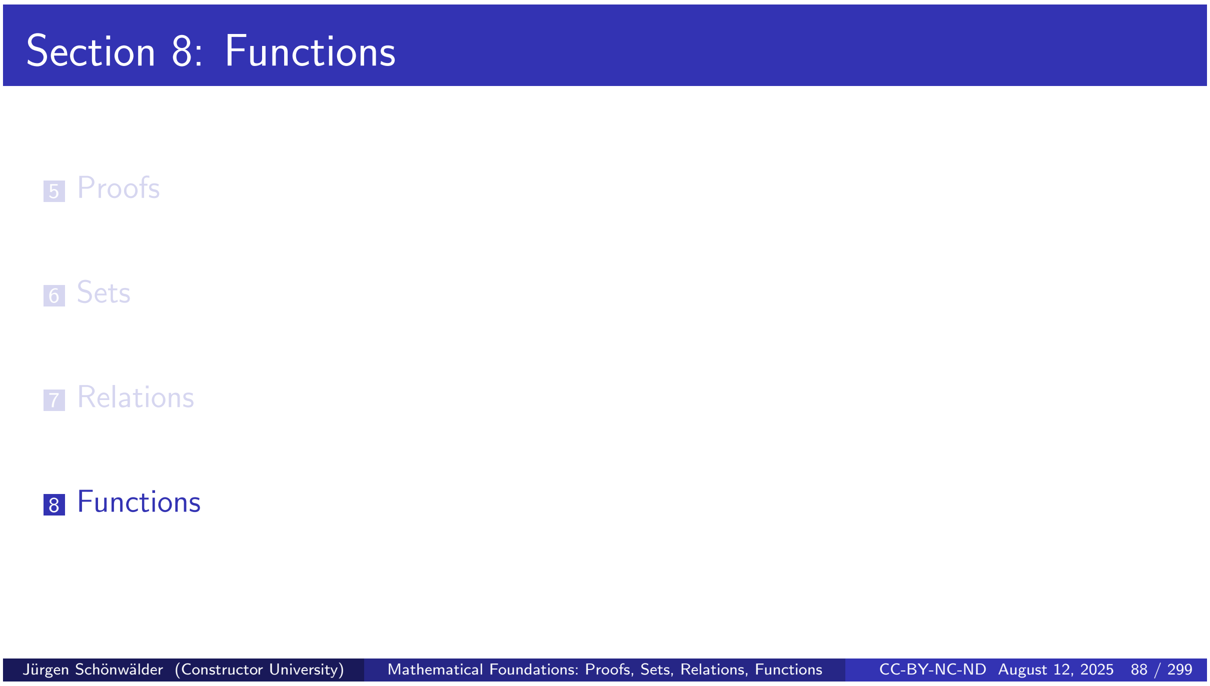
Summary of Properties of Binary Relations

Let $\sim$ be a binary relation over $A \times A$ and let $a, b, c \in A$ arbitrary.

property	$\equiv$	$\preceq$	$\prec$	definition	$=$	$\leq$	$<$
reflexive	✓	✓		$a \sim a$	✓	✓	
irreflexive			✓	$a \not\sim a$			✓
symmetric	✓			$a \sim b \rightarrow b \sim a$	✓		
asymmetric			✓	$a \sim b \rightarrow b \not\sim a$			✓
antisymmetric		✓		$a \sim b \land b \sim a \rightarrow a = b$		✓	
transitive	✓	✓	✓	$a \sim b \land b \sim c \rightarrow a \sim c$	✓	✓	✓

$\equiv$ equivalence relation, $\preceq$ partial order, $\prec$ strict partial order

This table summarizes the properties of relations and how they map to equivalence relations, partial orders, and strict partial orders. The right column shows the equivalence, less-than-or-equal, and less-than examples that we are all familiar with from the sets of numbers.



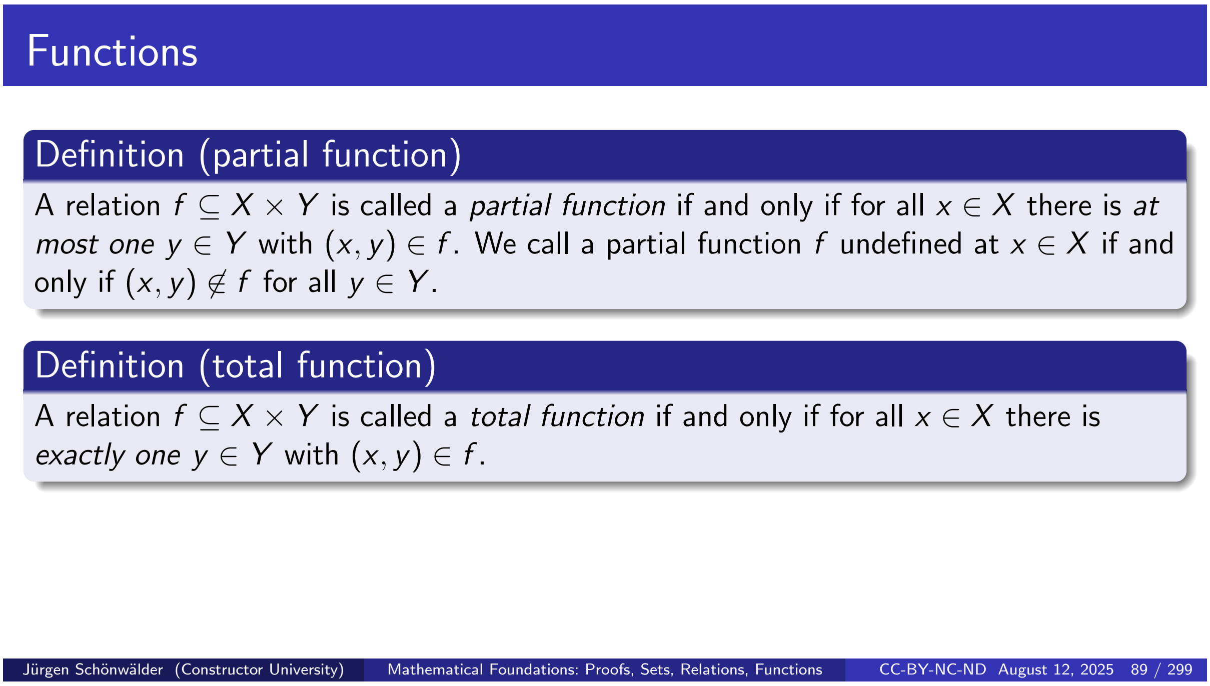
Section 8: Functions

5 Proofs

6 Sets

7 Relations

8 Functions



Functions

Definition (partial function)

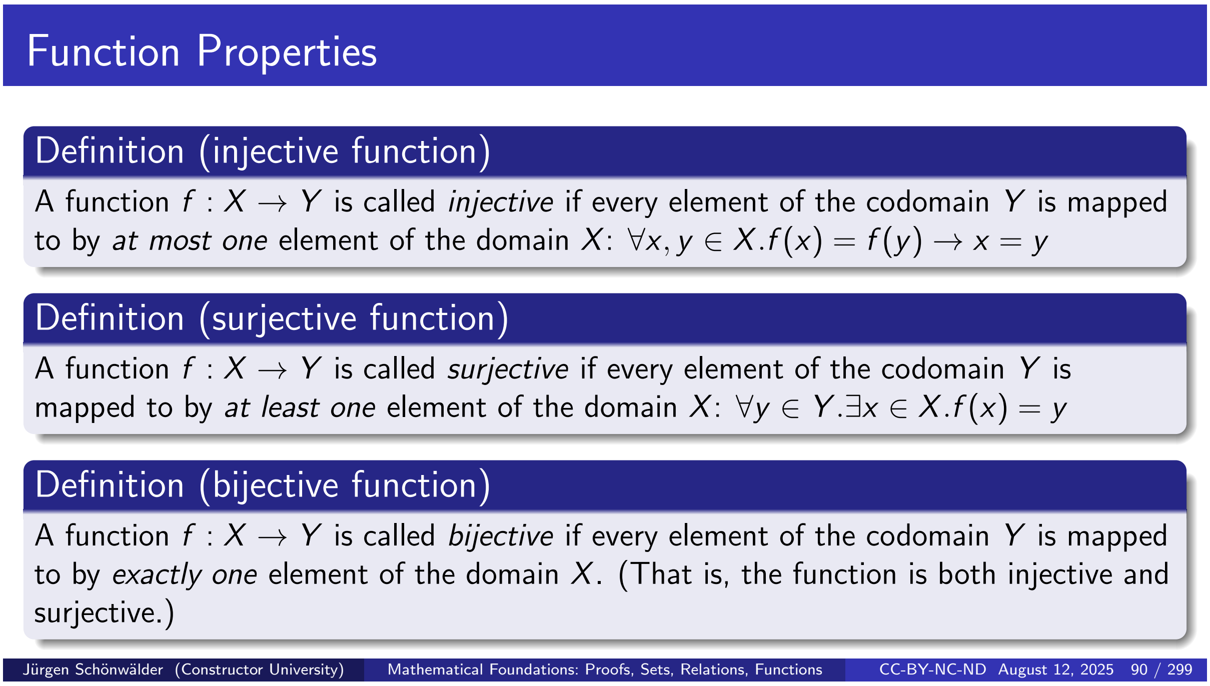
A relation $f \subseteq X \times Y$ is called a *partial function* if and only if for all $x \in X$ there is *at most one* $y \in Y$ with $(x, y) \in f$. We call a partial function f undefined at $x \in X$ if and only if $(x, y) \notin f$ for all $y \in Y$.

Definition (total function)

A relation $f \subseteq X \times Y$ is called a *total function* if and only if for all $x \in X$ there is *exactly one* $y \in Y$ with $(x, y) \in f$.

Notation:

- If $f \subseteq X \times Y$ is a total function, we write $f : X \to Y$.
- If $(x, y) \in f$, we often write $f(x) = y$.
- If a partial function f is undefined at $x \in X$, we often write $f(x) = \perp$.



Function Properties

Definition (injective function)

A function $f : X \to Y$ is called *injective* if every element of the codomain Y is mapped to by *at most one* element of the domain X: $\forall x, y \in X . f(x) = f(y) \to x = y$

Definition (surjective function)

A function $f : X \to Y$ is called *surjective* if every element of the codomain Y is mapped to by *at least one* element of the domain X: $\forall y \in Y . \exists x \in X . f(x) = y$

Definition (bijective function)

A function $f : X \to Y$ is called *bijective* if every element of the codomain Y is mapped to by *exactly one* element of the domain X. (That is, the function is both injective and surjective.)

If a function $f : X \to Y$ is bijective, we can easily obtain an inverse function $f^{-1} : Y \to X$.

- Example of an injective-only function $f : \{1, 2, 3\} \to \{A, B, C, D\}$ in graphviz dot notation:

```
digraph f {
        1->D
        2->B
        3->C
           A
}
```

- Example of a surjective-only function $f : \{1, 2, 3, 4\} \to \{B, C, D\}$ in graphviz dot notation:

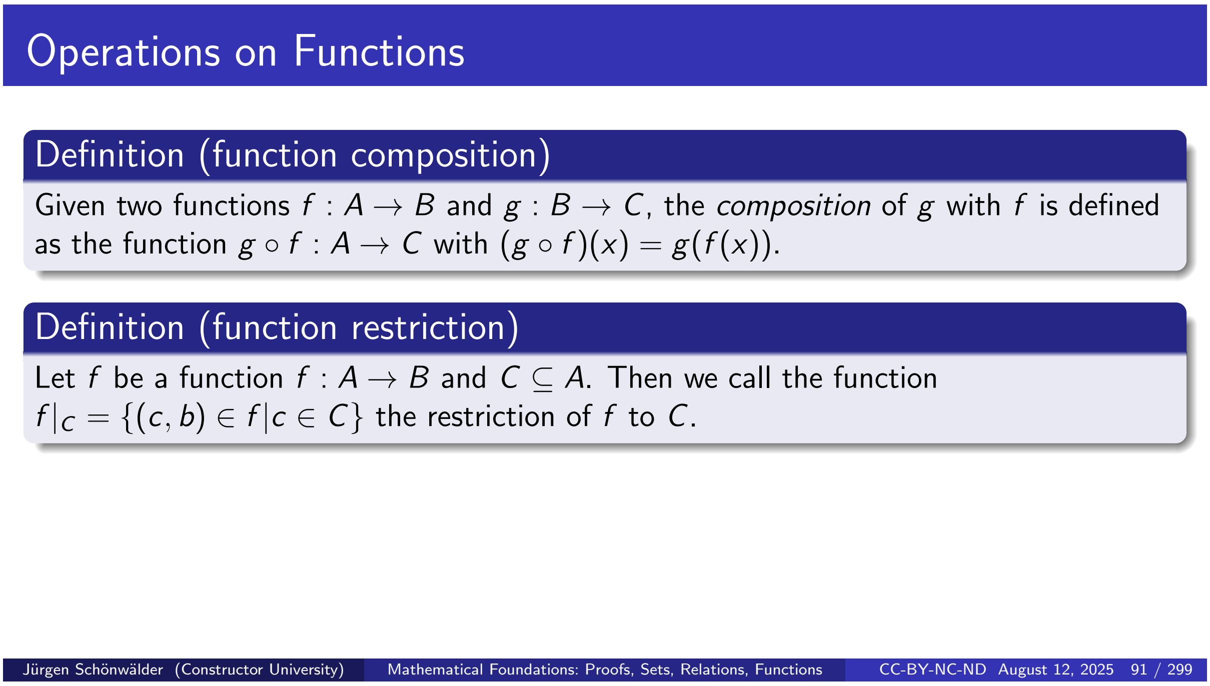
```
digraph f {
        1->D
        2->B
        3->C
        4->C
}
```

- Example of a bijective function $f : \{1, 2, 3, 4\} \to \{A, B, C, D\}$ in graphviz dot notation:

```
digraph f {
        1->D
        2->B
        3->C
        4->A
}
```

- Example of a function $f : \{1, 2, 3\} \to \{A, B, C, D\}$ that is neither:

```
digraph f {
        1->D
        2->D
           A
        3->C
           B
}
```



Operations on Functions

Definition (function composition)

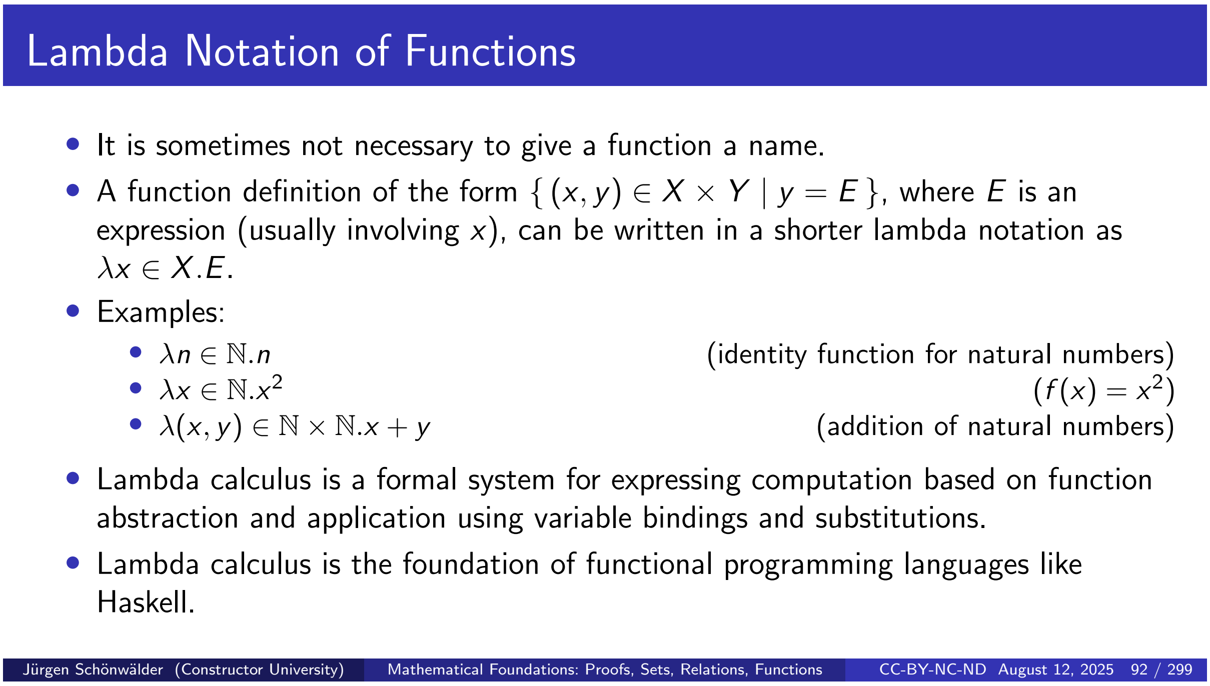
Given two functions $f : A \to B$ and $g : B \to C$, the *composition* of g with f is defined as the function $g \circ f : A \to C$ with $(g \circ f)(x) = g(f(x))$.

Definition (function restriction)

Let f be a function $f : A \to B$ and $C \subseteq A$. Then we call the function $f|_C = \{(c, b) \in f | c \in C\}$ the restriction of f to C.

Function composition is an important concept since sequences of computational steps can be understood as function compositions.

Lets consider a computational system that can be in the states $S = \{s_0, s_1, s_2, \ldots\}$ and let $f : S \to S$ be a state transition function. Then the composition of f with itself, i.e., $f \circ f \circ f \circ \ldots \circ f$ can describe a sequence of computational steps. There are classes of abstract machines, like finite state machines, push-down machines, or Turing machines, that essentially compose transition functions. In functional programming languages, command sequencing is often implemented as a form of function composition.



Lambda Notation of Functions

- It is sometimes not necessary to give a function a name.
- A function definition of the form $\{ (x, y) \in X \times Y \mid y = E \}$, where E is an expression (usually involving x), can be written in a shorter lambda notation as $\lambda x \in X.E$.
- Examples:
  - $\lambda n \in \mathbb{N}.n$ (identity function for natural numbers)
  - $\lambda x \in \mathbb{N}.x^2$ $(f(x) = x^2)$
  - $\lambda(x, y) \in \mathbb{N} \times \mathbb{N}.x + y$ (addition of natural numbers)
- Lambda calculus is a formal system for expressing computation based on function abstraction and application using variable bindings and substitutions.
- Lambda calculus is the foundation of functional programming languages like Haskell.

The lambda notation $\lambda x \in X.E$ implies a set that consists of elements of the form $(x, y) \in X \times Y$, i.e., the function argument(s) and the function value.

- Identity function for natural numbers:

$$\begin{aligned} \lambda n \in \mathbb{N}.n &= \{ (x, y) \in \mathbb{N} \times \mathbb{N} \mid y = x \} \\ &= \{ (0, 0), (1, 1), (2, 2), \dots \} \end{aligned}$$

- $f(x) = x^2$:

$$\begin{aligned} \lambda x \in \mathbb{N}.x^2 &= \{ (x, y) \in \mathbb{N} \times \mathbb{N} \mid y = x^2 \} \\ &= \{ (0, 0), (1, 1), (2, 4), \dots \} \end{aligned}$$

- Addition of natural numbers:

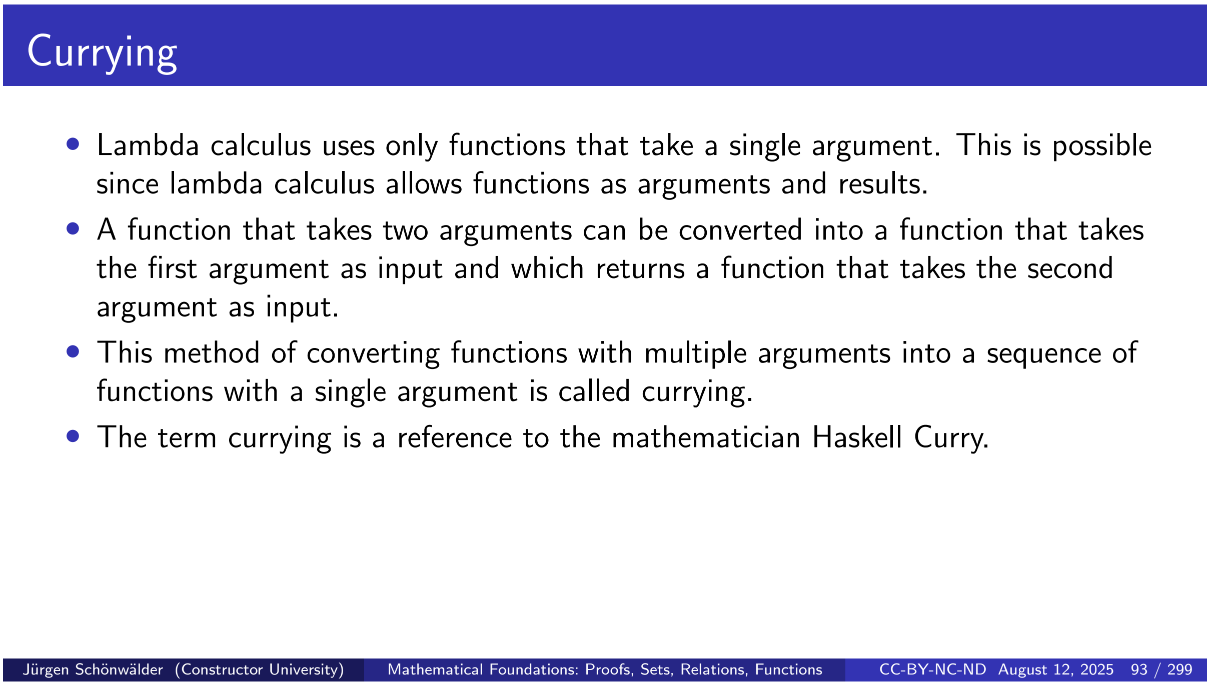
$$\begin{aligned} \lambda(x, y) \in \mathbb{N} \times \mathbb{N}.x + y &= \{ ((x, y), z) \in (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \mid z = x + y \} \\ &= \{ ((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 2), \dots \} \end{aligned}$$

The following example demonstrates lambda functions and function composition in Haskell.

```haskell
f :: Num a => a -> a
f = (\x->x^2) . (\x->x+1)
```

Further online information:

- **Wikipedia**: Lambda Calculus



Currying

- Lambda calculus uses only functions that take a single argument. This is possible since lambda calculus allows functions as arguments and results.
- A function that takes two arguments can be converted into a function that takes the first argument as input and which returns a function that takes the second argument as input.
- This method of converting functions with multiple arguments into a sequence of functions with a single argument is called currying.
- The term currying is a reference to the mathematician Haskell Curry.

The Haskell programming language uses currying to convert all functions with multiple arguments into a sequence of functions with a single argument.

Part III

Data Representation

In this part, we look at some fundamental data types such as different kinds of numbers, characters, strings, or dates and time. We explore how such data is commonly represented in computing machines.

We start by looking at different number systems. While we know number systems such as natural numbers, rational numbers, or real numbers from school, it turns out that computers tend to prefer some restricted versions of these number systems. It is important to be aware of the differences in order to produce software that behaves well.

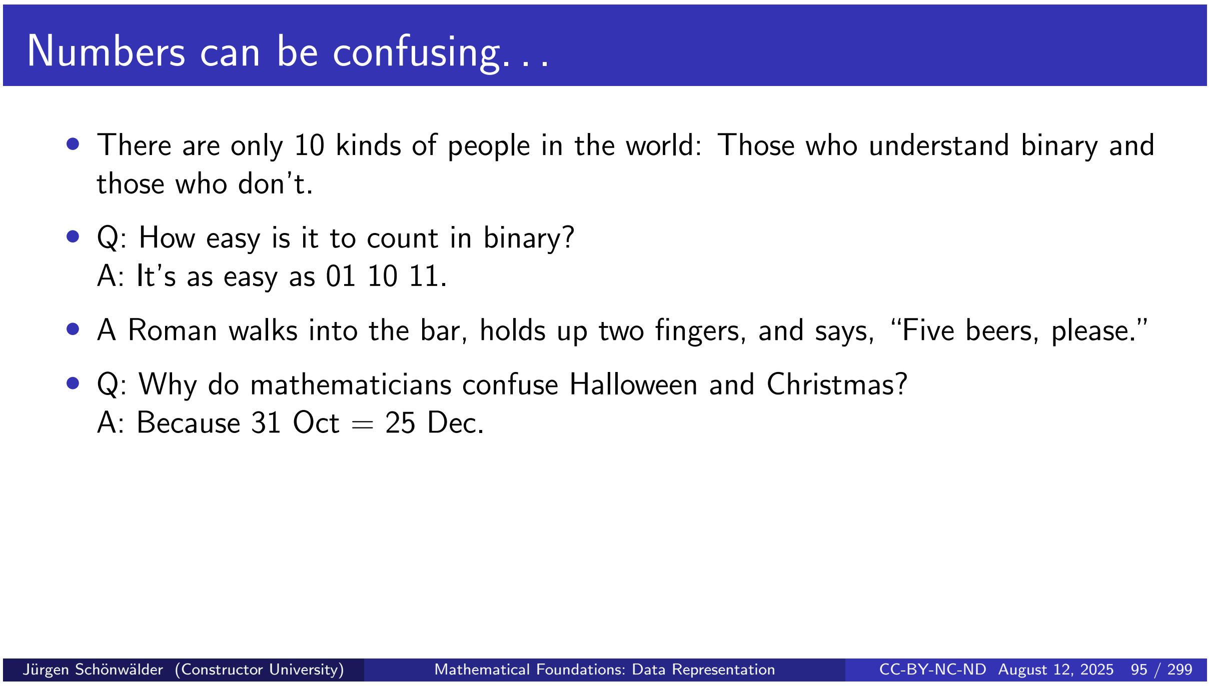
Most numbers have associated units and hence we briefly discuss the international system of units and metric prefixes.

We then turn to characters and some character representation and encoding issues. While characters in principle look like a simple concept, they actually are not if we consider the different character sets used in the word. Operations like character comparisons can become quite challenging in practice.

Finally, we look at the notion of time in computer systems and the representation of time and dates. This again turns out to be much more complicated than one might have hoped. Time is often not a good concept in distributed systems since establishing an accurate common notion of time is quite challenging.

By the end of this part, students should be able to

- represent natural numbers in number systems with arbitrary bases;
- convert integer numbers into fixed size (b-1) and b-complement representations;
- explain the difference between real numbers and floating point numbers;
- convert decimal fractions into IEEE floating point numbers;
- describe the limitations of floating point numbers;
- recognize the importance of units and the International system of units;
- apply metric and binary prefixes properly;
- illustrate different representations of characters and strings;
- summarize the ASCII and UNICODE character sets and the UTF-8 encoding;
- explain the value of standardized date and time formats.



Numbers can be confusing…

- There are only 10 kinds of people in the world: Those who understand binary and those who don't.
- Q: How easy is it to count in binary?
  A: It's as easy as 01 10 11.
- A Roman walks into the bar, holds up two fingers, and says, "Five beers, please."
- Q: Why do mathematicians confuse Halloween and Christmas?
  A: Because 31 Oct = 25 Dec.

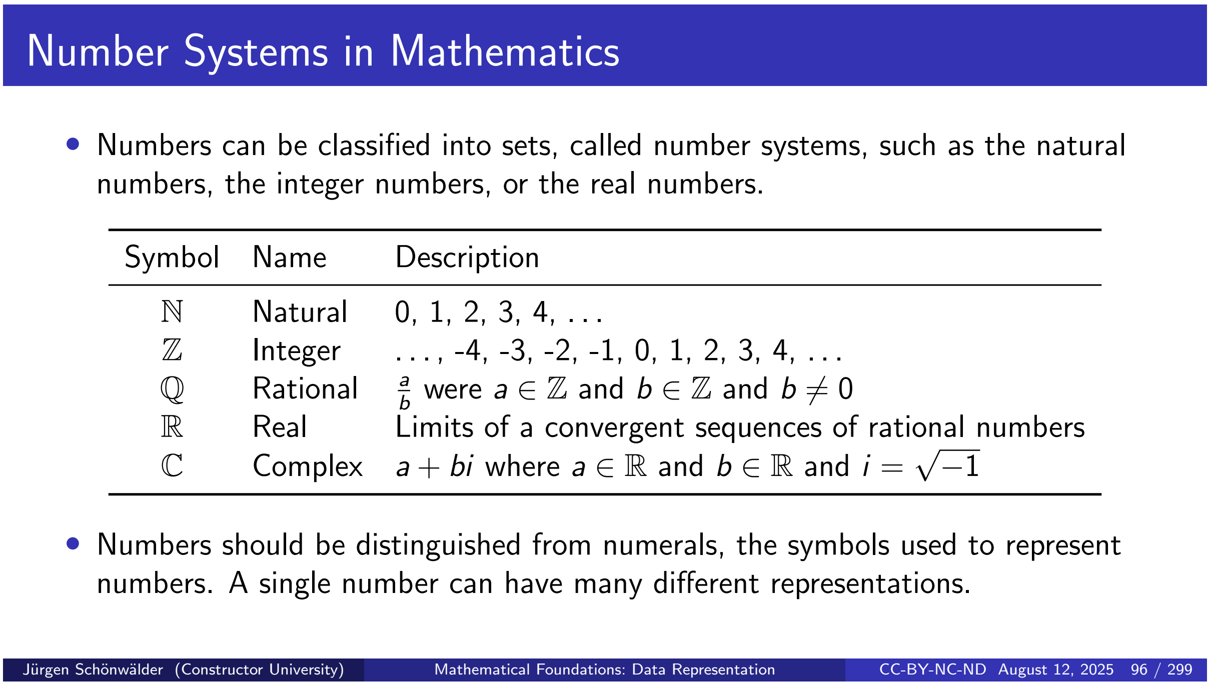
Jürgen Schönwälder (Constructor University) Mathematical Foundations: Data Representation CC-BY-NC-ND August 12, 2025 95 / 299

Computers are machines that can do calculations (or computations) with numbers much faster than humans. However, things can also go badly wrong if numbers are represented in different formats or limits of precision are reached. Even seemingly simple calculations (for humans used to the decimal number system) can sometimes not be carried out by computers accurately (and this is independent of how expensive the computer is).

Hence it is of crucial importance to understand how computers represent numbers and which errors can appear in computations performed by them.

Further online information:

- http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html



Number Systems in Mathematics

- Numbers can be classified into sets, called number systems, such as the natural numbers, the integer numbers, or the real numbers.

Symbol	Name	Description
$\mathbb{N}$	Natural	0, 1, 2, 3, 4, . . .
$\mathbb{Z}$	Integer	. . . , -4, -3, -2, -1, 0, 1, 2, 3, 4, . . .
$\mathbb{Q}$	Rational	$\frac{a}{b}$ were $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ and $b \neq 0$
$\mathbb{R}$	Real	Limits of a convergent sequences of rational numbers
$\mathbb{C}$	Complex	$a + bi$ where $a \in \mathbb{R}$ and $b \in \mathbb{R}$ and $i = \sqrt{-1}$

- Numbers should be distinguished from numerals, the symbols used to represent numbers. A single number can have many different representations.

Note the difference between a number and its representations. The number forty-two can be represented as:

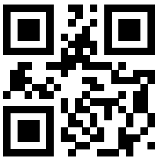
(a) 42 (decimal number)

(b) 2a (hexadecimal number)

(c) 101010 (binary number)

(d) XLII (roman number)

(e) - . . . - . . - - - (morse code)

(f)  (qr code)

Section 9: Natural Numbers

9 Natural Numbers

10 Integer Numbers

11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time

Numeral Systems for Natural Numbers

- Natural numbers can be represented using different bases. We commonly use decimal (base 10) number representations in everyday life.
- In computer science, we also frequently use binary (base 2), octal (base 8), and hexadecimal (base 16) number representations.
- In general, natural numbers represented in the base b system are of the form:

$$(a_n a_{n-1} \cdots a_1 a_0)_b = \sum_{k=0}^{n} a_k b^k$$

hex	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12
dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21	22
bin	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001	10010

Algorithm 1 Conversion of a decimal natural number n into a bit string representing a binary number

```
function DEC2BIN($n$)
    $s \leftarrow$ ""                                  ▷ $s$ holds a bit string
    repeat
        $b \leftarrow n$ mod 2                         ▷ the remainder gives us the next bit
        prepend bit $b$ to the bit string $s$
        $n \leftarrow n$ div 2                         ▷ the number still left to convert
    until $n = 0$
    return $s$
end function
```

Algorithm 2 Conversion of a bit string s representing a binary number into a decimal number

```
function BIN2DEC($s$)
    $n \leftarrow 0$                                   ▷ $n$ holds a natural number
    while bit string $s$ is not empty do
        $b \leftarrow$ leftmost bit of the bit string $s$    ▷ removes the bit from the bit string
        $n \leftarrow 2 \cdot n + b$
    od
    return $n$
end function
```

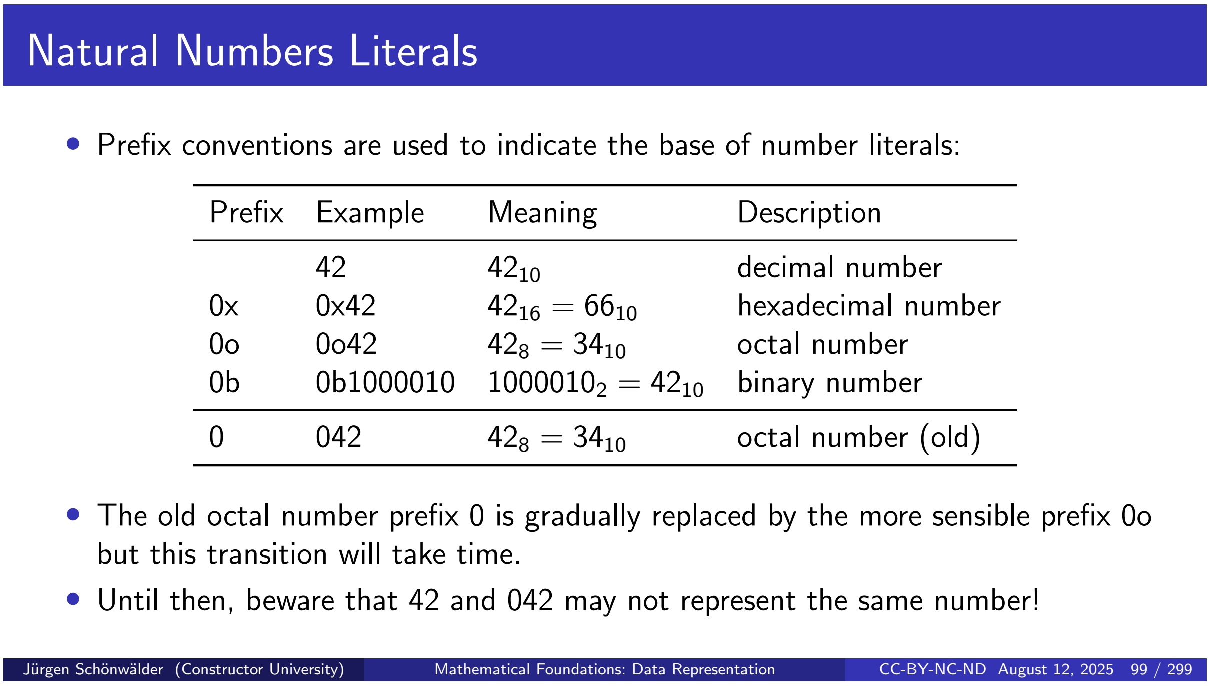
To convert binary numbers to octal or hexadecimal numbers or vice versa, group triples or quadrupels of binary digits into recognizable chunks (add leading zeros as needed):

$$110001101011100_2 = \underbrace{0110_2}_{6_{16}}\underbrace{0011_2}_{3_{16}}\underbrace{0101_2}_{5_{16}}\underbrace{1100_2}_{c_{16}} = 635c_{16}$$

$$110001101011100_2 = \underbrace{110_2}_{6_8}\underbrace{001_2}_{1_8}\underbrace{101_2}_{5_8}\underbrace{011_2}_{3_8}\underbrace{100_2}_{4_8} = 61534_8$$

This trick also works in the other direction: Convert every octal or hexadecimal digit individually into a binary chunk:

$$deadbeaf_{16} = \underbrace{d_{16}}_{1101_2}\underbrace{e_{16}}_{1110_2}\underbrace{a_{16}}_{1010_2}\underbrace{d_{16}}_{1101_2}\underbrace{b_{16}}_{1011_2}\underbrace{e_{16}}_{1110_2}\underbrace{a_{16}}_{1010_2}\underbrace{f_{16}}_{1111_2} = 11011110101011011011111011101111_2$$



Natural Numbers Literals

- Prefix conventions are used to indicate the base of number literals:

Prefix	Example	Meaning	Description
	42	42_{10}	decimal number
0x	0x42	$42_{16} = 66_{10}$	hexadecimal number
0o	0o42	$42_8 = 34_{10}$	octal number
0b	0b1000010	$1000010_2 = 42_{10}$	binary number
0	042	$42_8 = 34_{10}$	octal number (old)

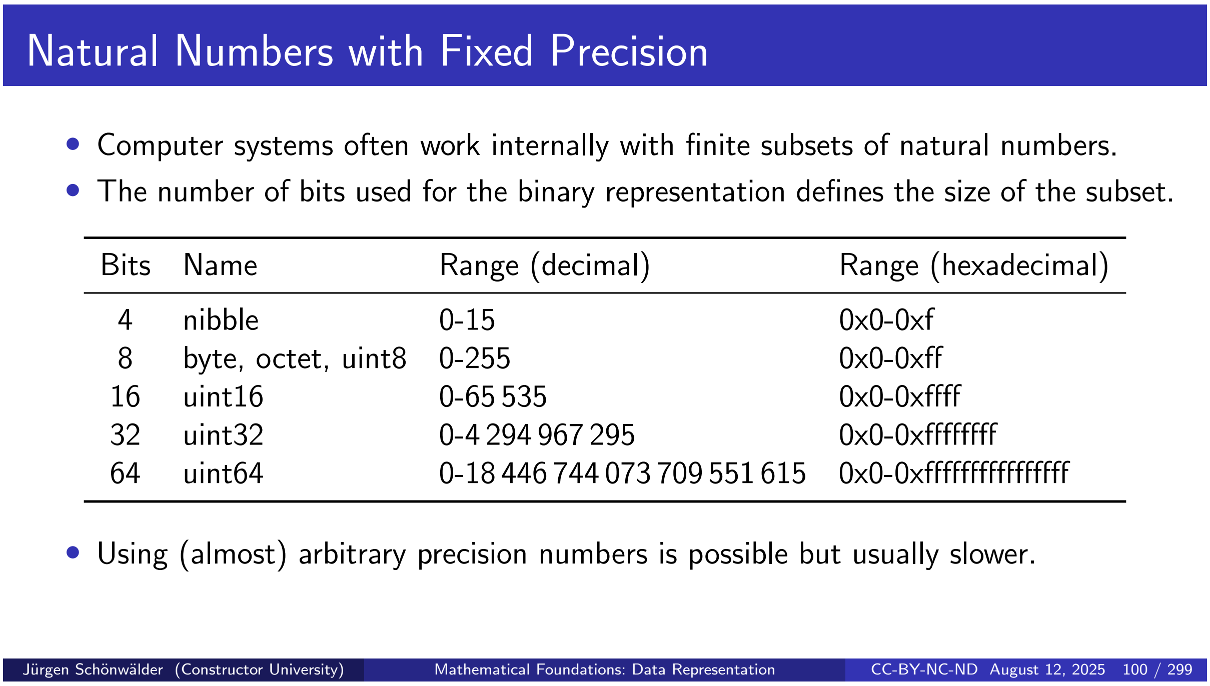
- The old octal number prefix 0 is gradually replaced by the more sensible prefix 0o but this transition will take time.
- Until then, beware that 42 and 042 may not represent the same number!

Conversion of the decimal number 123 into binary using the algorithm dec2bin:

$$\begin{aligned}
123 \bmod 2 &= 1 & 1_2 \\
61 \bmod 2 &= 1 & 11_2 \\
30 \bmod 2 &= 0 & 011_2 \\
15 \bmod 2 &= 1 & 1011_2 \\
7 \bmod 2 &= 1 & 11011_2 \\
3 \bmod 2 &= 1 & 111011_2 \\
1 \bmod 2 &= 1 & 1111011_2
\end{aligned}$$

Conversion of the binary number 1111011_2 into a decimal number using the algorithm bin2dec:

$$\begin{aligned}
1111011_2 & & 0 \cdot 2 + 1 = 1 \\
111011_2 & & 1 \cdot 2 + 1 = 3 \\
11011_2 & & 3 \cdot 2 + 1 = 7 \\
1011_2 & & 7 \cdot 2 + 1 = 15 \\
011_2 & & 15 \cdot 2 + 0 = 30 \\
11_2 & & 30 \cdot 2 + 1 = 61 \\
1_2 & & 61 \cdot 2 + 1 = 123
\end{aligned}$$



Natural Numbers with Fixed Precision

- Computer systems often work internally with finite subsets of natural numbers.
- The number of bits used for the binary representation defines the size of the subset.

Bits	Name	Range (decimal)	Range (hexadecimal)
4	nibble	0-15	0x0-0xf
8	byte, octet, uint8	0-255	0x0-0xff
16	uint16	0-65 535	0x0-0xffff
32	uint32	0-4 294 967 295	0x0-0xffffffff
64	uint64	0-18 446 744 073 709 551 615	0x0-0xffffffffffffffff

- Using (almost) arbitrary precision numbers is possible but usually slower.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Data Representation CC-BY-NC-ND August 12, 2025 100 / 299

Fixed point natural numbers often silently wrap around, as demonstrated by the following C program:

```c
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

static uint8_t add(uint8_t a, uint8_t b)
{
    uint8_t s = a + b;
    printf("%u + %u = %u\n", a, b, s);
    return s;
}

static uint8_t sub(uint8_t a, uint8_t b)
{
    uint8_t s = a - b;
    printf("%u - %u = %u\n", a, b, s);
    return s;
}

int main(void)
{
    assert(add(UINT8_MAX, 1) == 0);
    assert(sub(0, 1) == 255);
    assert(add(add(42, UINT8_MAX), 1) == 42);
    assert(sub(sub(42, UINT8_MAX), 1) == 42);
    return 0;
}
```

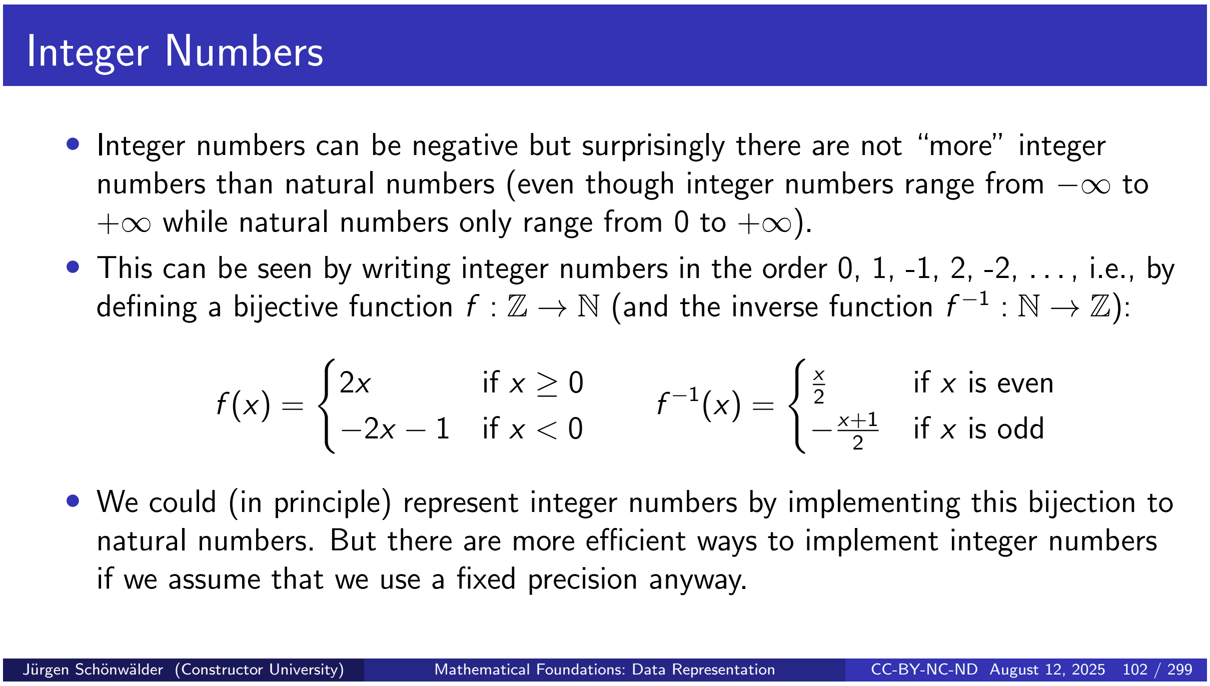
This program produces the following output:

```
255 + 1 = 0
0 - 1 = 255
42 + 255 = 41
41 + 1 = 42
42 - 255 = 43
43 - 1 = 42
```

Careless choice of fixed precision number ranges can lead to serious disasters. (The situation is even more problematic in programming languages that perform automatic type conversions.) Some modern programming languages take a different approach. Rust, for example, treats integer overflows as errors as long as programs are compiled in debug mode. Once a program is compiled into release mode, the overflow checks are removed in order to optimize for speed. Special APIs are provided for situations where a certain overflow behaviour is required.

```c
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

static uint8_t add(uint8_t a, uint8_t b)
{
    uint8_t s = a + b;
    printf("%u + %u = %u\n", a, b, s);
    return s;
}

static uint8_t sub(uint8_t a, uint8_t b)
{
    uint8_t s = a - b;
    printf("%u - %u = %u\n", a, b, s);
    return s;
}

int main(void)
{
    assert(add(UINT8_MAX, 1) == 0);
    assert(sub(0, 1) == 255);
    assert(add(add(42, UINT8_MAX), 1) == 42);
    assert(sub(sub(42, UINT8_MAX), 1) == 42);
    return 0;
}
```

Section 10: Integer Numbers



Integer Numbers

- Integer numbers can be negative but surprisingly there are not “more” integer numbers than natural numbers (even though integer numbers range from $-\infty$ to $+\infty$ while natural numbers only range from 0 to $+\infty$).
- This can be seen by writing integer numbers in the order 0, 1, -1, 2, -2, . . . , i.e., by defining a bijective function $f : \mathbb{Z} \to \mathbb{N}$ (and the inverse function $f^{-1} : \mathbb{N} \to \mathbb{Z}$):

$$f(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ -2x - 1 & \text{if } x < 0 \end{cases} \qquad f^{-1}(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ -\frac{x+1}{2} & \text{if } x \text{ is odd} \end{cases}$$

- We could (in principle) represent integer numbers by implementing this bijection to natural numbers. But there are more efficient ways to implement integer numbers if we assume that we use a fixed precision anyway.

This representation of negative numbers has disadvantages:

- Consecutive integer numbers do not have consecutive positions in the unsigned representation since positive and negative numbers alternate.
- Positive natural numbers and positive integer numbers have different representations. For fixed precision 8-bit numbers, the decimal number 5 is represented as 0b00000101 as an unsigned natural number while the same decimal number 5 would be represented as 0b00001010 as a signed integer number.

Hence it is desirable to find a different bijection that represents “small” positive integer numbers in the same way as corresponding unsigned natural numbers and that maps negative numbers into the space of the “larger” natural numbers. In addition, it would be fantastic if a logic circuit used to add unsigned natural numbers can also be used to add signed integer numbers.

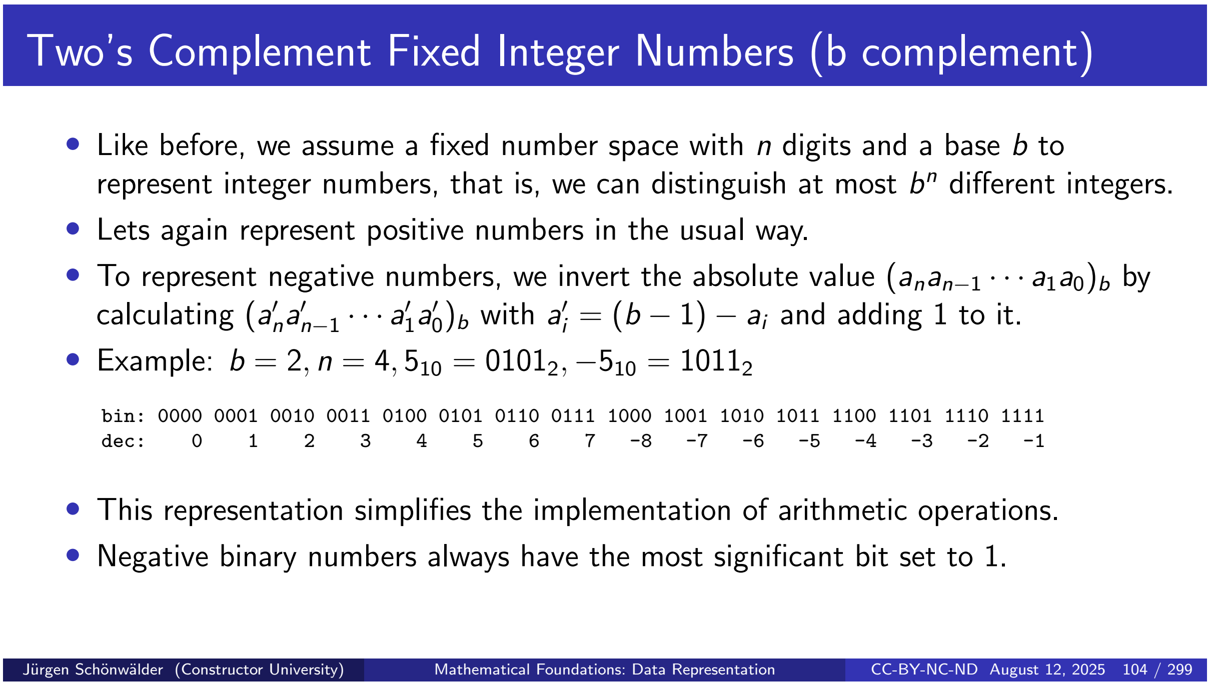
One's Complement Fixed Integer Numbers (b-1 complement)

- We have a fixed number space with n digits and base b to represent integer numbers, that is, we can distinguish at most b^n different integers.
- Lets represent positive numbers in the usual way.
- To represent negative numbers, we invert the absolute value $(a_n a_{n-1} \cdots a_1 a_0)_b$ by calculating $(a'_n a'_{n-1} \cdots a'_1 a'_0)_b$ with $a'_i = (b-1) - a_i$.
- Example: $b = 2, n = 4, 5_{10} = 0101_2, -5_{10} = 1010_2$

```
bin: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
dec:    0    1    2    3    4    5    6    7   -7   -6   -5   -4   -3   -2   -1   -0
```

- Note that this gives us $+0$ and -0, i.e., we only represent $b^n - 1$ different integers.
- Negative binary numbers always have the most significant bit set to 1.

Having both +0 and -0 can be avoided by shifting all negative numbers one position to the right. This removes the -0 and creates space to represent an additional negative number. This idea leads us to the b complement discussed next.



Two's Complement Fixed Integer Numbers (b complement)

- Like before, we assume a fixed number space with n digits and a base b to represent integer numbers, that is, we can distinguish at most b^n different integers.
- Lets again represent positive numbers in the usual way.
- To represent negative numbers, we invert the absolute value $(a_n a_{n-1} \cdots a_1 a_0)_b$ by calculating $(a'_n a'_{n-1} \cdots a'_1 a'_0)_b$ with $a'_i = (b-1) - a_i$ and adding 1 to it.
- Example: $b = 2, n = 4, 5_{10} = 0101_2, -5_{10} = 1011_2$

```
bin: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
dec:    0    1    2    3    4    5    6    7   -8   -7   -6   -5   -4   -3   -2   -1
```

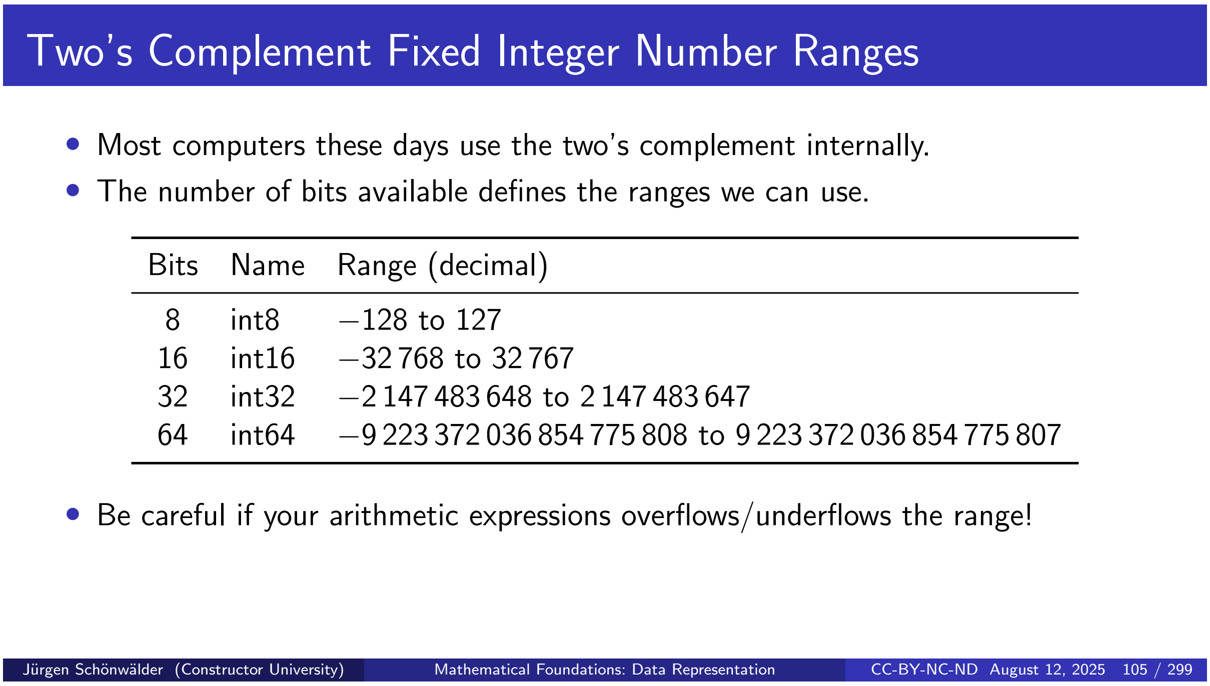
- This representation simplifies the implementation of arithmetic operations.
- Negative binary numbers always have the most significant bit set to 1.

Benefits of the two's complement for binary numbers:

- Positive numbers and 0 have the most significant bit set to 0.
- Negative numbers have the most significant bit set to 1.
- There is only a single representation for 0.
- For positive numbers, the two's complement representation corresponds to the normal binary representation.

Example: Calculate $2_{10} - 6_{10} = 2_{10} + (-6_{10})$ using binary numbers using two's complement representation for negative numbers with 4 digits.

> Conversion into binary numbers yields $2_{10} = 0010_2$ and $-6_{10} = 1001_2 + 0001_s = 1010_2$. With this, we can simply add the two numbers: $0010_2 + 1010_2 = 1100_2$. The result 1100_2 is a negative number. Inverting the bits and adding one gives us $0100_2 = 4_{10}$. Hence, the result is -4.



Two's Complement Fixed Integer Number Ranges

- Most computers these days use the two's complement internally.
- The number of bits available defines the ranges we can use.

Bits	Name	Range (decimal)
8	int8	−128 to 127
16	int16	−32 768 to 32 767
32	int32	−2 147 483 648 to 2 147 483 647
64	int64	−9 223 372 036 854 775 808 to 9 223 372 036 854 775 807

- Be careful if your arithmetic expressions overflows/underflows the range!

Note that computer hardware usually does not warn you about integer overflows or underflows. Instead, numbers simply wrap around.

```c
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

static int8_t add(int8_t a, int8_t b)
{
    int8_t s = a + b;
    printf("%d + %d = %d\n", a, b, s);
    return s;
}

static int8_t sub(int8_t a, int8_t b)
{
    int8_t s = a - b;
    printf("%d - %d = %d\n", a, b, s);
    return s;
}

int main(void)
{
    assert(add(INT8_MAX, 1) == INT8_MIN);
    assert(sub(INT8_MIN, 1) == INT8_MAX);
    return 0;
}
```

This program produces the following output:

```
127 + 1 = -128
-128 - 1 = 127
```

Section 11: Rational and Real Numbers

9 Natural Numbers

10 Integer Numbers

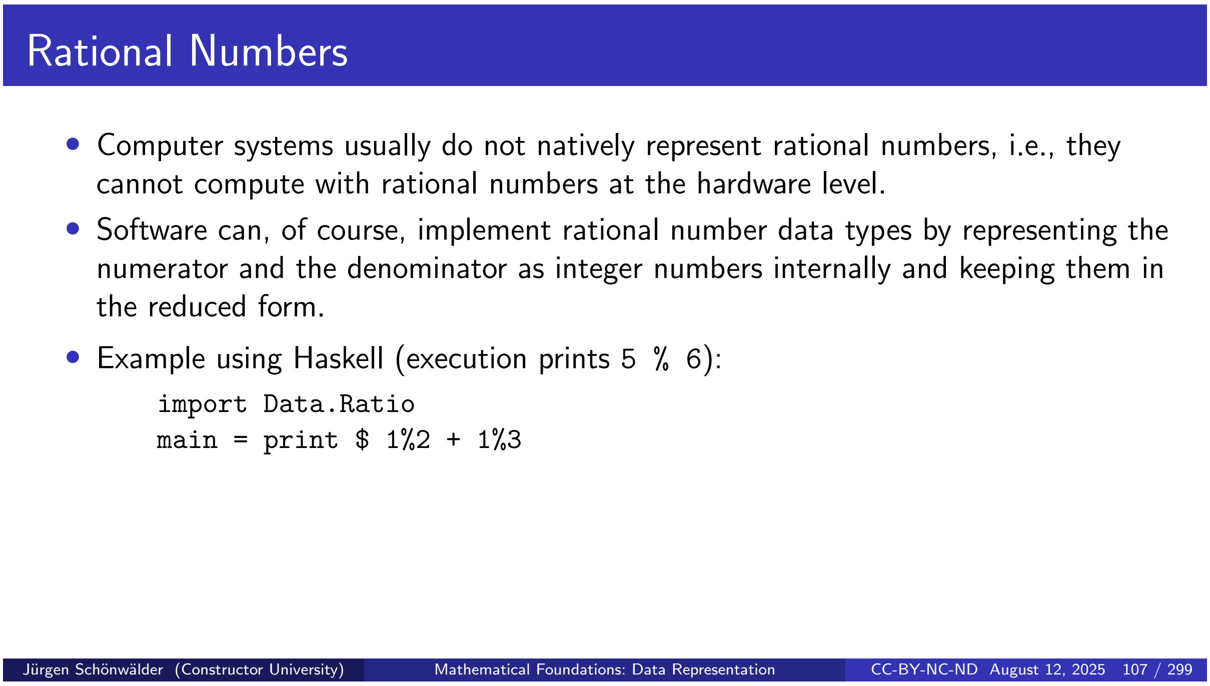
11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time



Rational Numbers

- Computer systems usually do not natively represent rational numbers, i.e., they cannot compute with rational numbers at the hardware level.
- Software can, of course, implement rational number data types by representing the numerator and the denominator as integer numbers internally and keeping them in the reduced form.
- Example using Haskell (execution prints 5 % 6):

```
import Data.Ratio
main = print $ 1%2 + 1%3
```

The following code examples are illustrative.

The equivalent Python code would look like this:

```python
from fractions import Fraction
a = Fraction("1/2")
b = Fraction("1/3")
print(a + b)
```

C++ has support for rational numbers in the standard library:

```cpp
#include <iostream>
#include <ratio>

int main()
{
    typedef std::ratio<1, 2> a;
    typedef std::ratio<1, 3> b;
    typedef std::ratio_add<a, b> sum;
    std::cout << sum::num << '/' << sum::den << '\n';
    return 0;
}
```

C programmers can use the GNU multiple precision arithmetic library:

```c
#include <gmp.h>

int main()
{
    mpq_t a, b, c;
    mpq_inits(a, b, c, NULL);
    mpq_set_str(a, "1/2", 10);
    mpq_set_str(b, "1/3", 10);
    mpq_add(c, a, b);
    gmp_printf("%Qd\n", c);
    mpq_clears(a, b, c, NULL);
    return 0;
}
```

Real Numbers

- Computer systems usually do not natively represent real numbers, i.e., they cannot compute with real numbers at the hardware level.
- The primary reason is that real numbers like the result of $\frac{1}{7}$ or numbers like π have by definition not a finite representation.
- So the best we can do is to have a finite approximation...
- Since all we have are approximations of real numbers, we *always* make rounding errors when we use these approximations. If we are not extremely cautious, these rounding errors can *accumulate* badly.
- Numeric algorithms can be analyzed according to how good or bad they propagate rounding errors, leading to the notion of *numeric stability*.

Numeric stability is an important criterion for numeric algorithms. Numeric algorithms must be designed with numeric stability in mind. While it may seem easy to translate certain mathematical formulae into code, naive translations can produce pretty surprising results. If in doubt, use an algorithm for which it has been shown that it has numeric stability and implement the algorithm as defined; resist the idea to create a variation, unless you do an analysis of the numeric stability of your variation.

Section 12: Floating Point Numbers

9 Natural Numbers

10 Integer Numbers

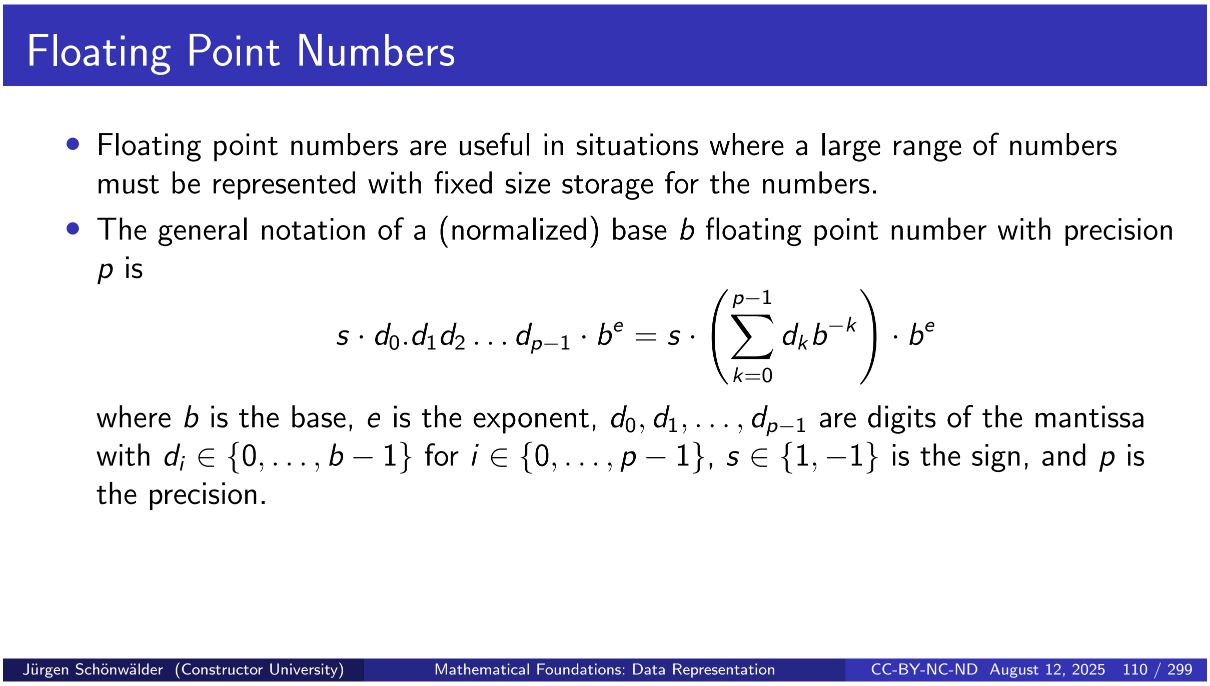
11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time



Floating Point Numbers

- Floating point numbers are useful in situations where a large range of numbers must be represented with fixed size storage for the numbers.
- The general notation of a (normalized) base b floating point number with precision p is

$$s \cdot d_0.d_1d_2 \ldots d_{p-1} \cdot b^e = s \cdot \left(\sum_{k=0}^{p-1} d_k b^{-k} \right) \cdot b^e$$

  where b is the base, e is the exponent, $d_0, d_1, \ldots, d_{p-1}$ are digits of the mantissa with $d_i \in \{0, \ldots, b-1\}$ for $i \in \{0, \ldots, p-1\}$, $s \in \{1, -1\}$ is the sign, and p is the precision.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Data Representation CC-BY-NC-ND August 12, 2025 110 / 299

As humans, we are used to base $b = 10$ numbers and the so called scientific notation of large (or small) numbers. For example, we write the speed of light as $2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ and the elementary positive charge as $1.602\,176\,634 \times 10^{-19}\,\mathrm{C}$.

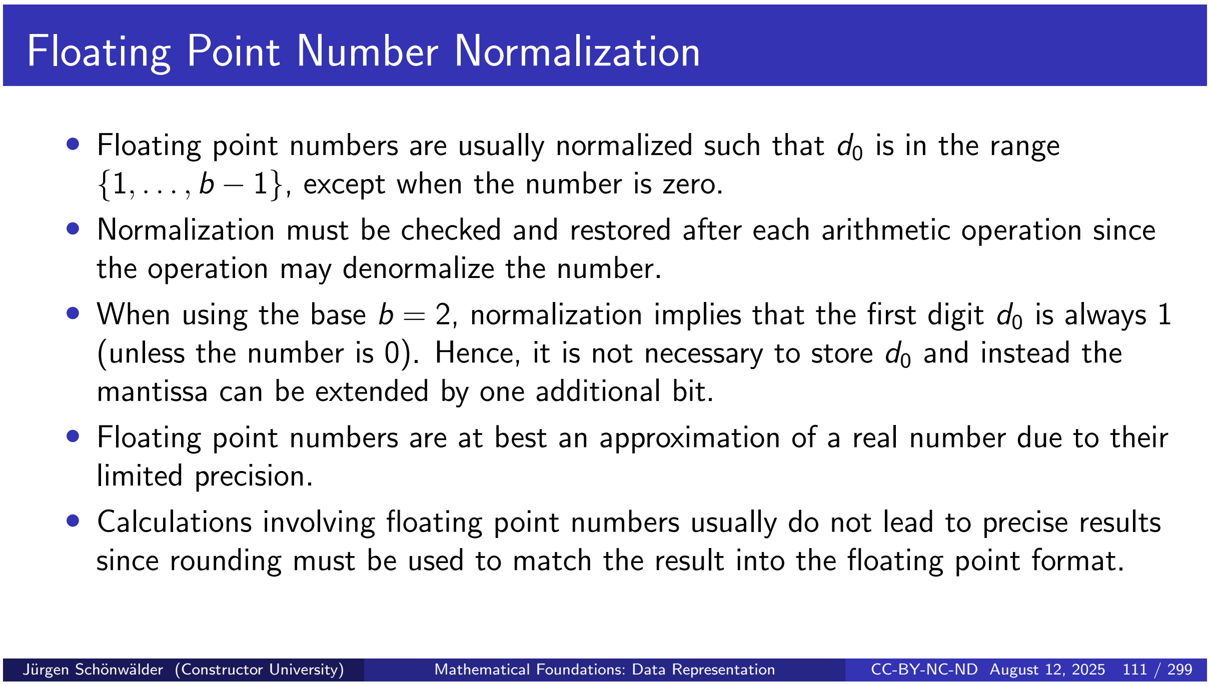
Computers prefer to use the base $b = 2$ for efficiency reasons. Even if you input and output the number in a decimal scientific notation ($b = 10$), internally the number is most likely stored with base $b = 2$. This means that there is a conversion whenever you input or output floating point numbers in decimal notation.

Algorithm 3 Conversion of a decimal fraction f into a bit string representing a binary fraction

```
function DECF2BINF($f$)
    $s \leftarrow$ ""                          ▷ $s$ holds a bit string
    repeat
        $f \leftarrow f \cdot 2$
        $b \leftarrow int(f)$                  ▷ int() yields the integer part
        append $b$ to the bit string $s$
        $f \leftarrow frac(f)$                 ▷ frac() yields the fractional part
    until $f = 0$                              ▷ may not always be reached
    return $s$
end function
```

Algorithm 4 Conversion of a bit string s representing a binary fraction into a decimal fraction

```
function BINF2DECF($s$)
    $f \leftarrow 0.0$                          ▷ $n$ holds a decimal fraction
    while bit string $s$ is not empty do
        $b \leftarrow$ rightmost bit of the bit string $s$    ▷ removes the bit from the bit string
        $f \leftarrow (f + b)/2$
    od
    return $f$
end function
```



Floating Point Number Normalization

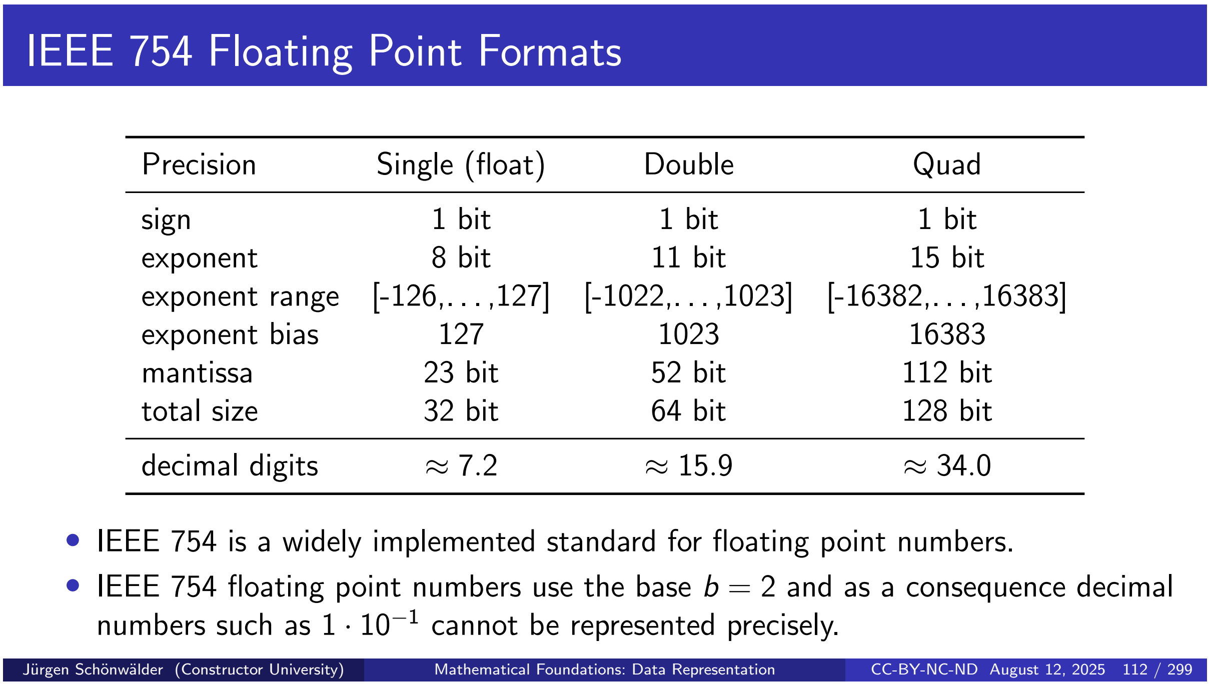
- Floating point numbers are usually normalized such that d_0 is in the range $\{1, \ldots, b-1\}$, except when the number is zero.
- Normalization must be checked and restored after each arithmetic operation since the operation may denormalize the number.
- When using the base $b = 2$, normalization implies that the first digit d_0 is always 1 (unless the number is 0). Hence, it is not necessary to store d_0 and instead the mantissa can be extended by one additional bit.
- Floating point numbers are at best an approximation of a real number due to their limited precision.
- Calculations involving floating point numbers usually do not lead to precise results since rounding must be used to match the result into the floating point format.

It is important for you to remember that floating point arithmetic is generally not exact. This applies to almost all digital calculators, regardless whether it is a school calculator, an app in your mobile phone, a calculator program in your computer. Many of these programs try to hide the fact that the results produced are imprecise by internally using more bits than what is shown to the user. While this helps a bit, it does not cure the problem, as will be demonstrated on subsequent pages.

While there are programs to do better than average when it comes to floating point numbers, many programs that are used widely may suffer from floating point imprecision. The most important thing is that you are aware of the simple fact that floating point numbers produced by a computer may simply be incorrect.

Further online information:

- **Wikipedia**: Numeric Precision in Microsoft Excel



IEEE 754 Floating Point Formats

Precision	Single (float)	Double	Quad
sign	1 bit	1 bit	1 bit
exponent	8 bit	11 bit	15 bit
exponent range	[-126,. . . ,127]	[-1022,. . . ,1023]	[-16382,. . . ,16383]
exponent bias	127	1023	16383
mantissa	23 bit	52 bit	112 bit
total size	32 bit	64 bit	128 bit
decimal digits	≈ 7.2	≈ 15.9	≈ 34.0

- IEEE 754 is a widely implemented standard for floating point numbers.
- IEEE 754 floating point numbers use the base $b = 2$ and as a consequence decimal numbers such as $1 \cdot 10^{-1}$ cannot be represented precisely.

The exponent range allows for positive and negative values. In order to avoid having to encode negative exponents, the exponent is biased by adding the exponent bias. For example, the exponent -17 of a single precision floating point number will be represented by the biased exponent value $-17+127 = 110$. (The exponent bias shifts the exponent range from $[-126, \ldots, 127]$ to $[1, \ldots, 254]$ for single precision floating point numbers.)

Single precision format:

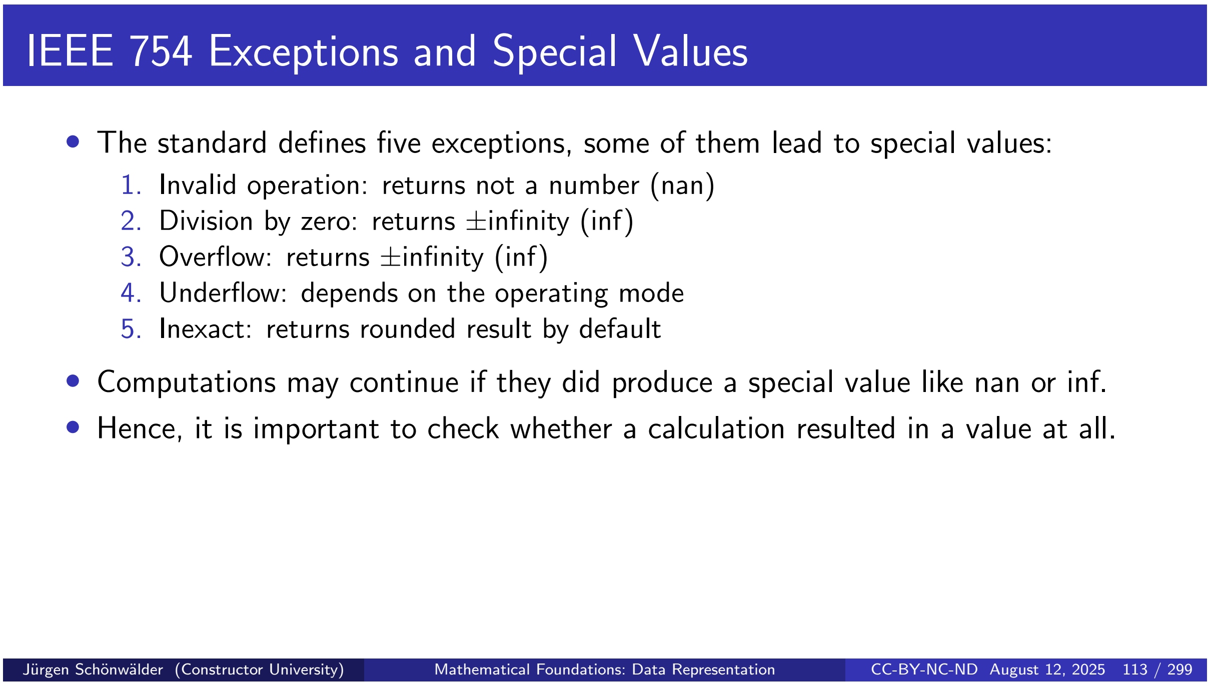
```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |S|    exponent   |         mantissa (23 bits)                  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Double precision format:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |S| exponent (11 bits)  |                                       :
    +-+-+-+-+-+-+-+-+-+-+-+-+         mantissa (52 bits)            :
    :                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Quadruple precision format:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |S|     exponent (15 bits)      |                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |
    :                                   mantissa (112 bits)         |
    :                                                               |
    :                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



IEEE 754 Exceptions and Special Values

- The standard defines five exceptions, some of them lead to special values:
  1. Invalid operation: returns not a number (nan)
  2. Division by zero: returns ±infinity (inf)
  3. Overflow: returns ±infinity (inf)
  4. Underflow: depends on the operating mode
  5. Inexact: returns rounded result by default
- Computations may continue if they did produce a special value like nan or inf.
- Hence, it is important to check whether a calculation resulted in a value at all.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Data Representation CC-BY-NC-ND August 12, 2025 113 / 299

We have seen that integer numbers usually just silently overflow (or underflow) by “wrapping” around. IEEE 754 floating point numbers behave differently when it comes to overflows, this is in a way perhaps an improvement.

The special values use special encoding rules:

- Infinity is indicated by setting all bits of the exponent field to one and all bits of the mantissa to zero. The sign bit indicate positive or negative infinity.
- NaN is encoded by setting all bits of the exponent and the mantissa fields to one and by clearning the sign bit.

Floating Point Surprises

- Any floating point computation should be treated with the utmost suspicion unless you can argue how accurate it is. [Alan Mycroft, Cambridge]
- Floating point arithmetic almost always involves rounding errors and these errors can badly aggregate.
- It is possible to "loose" the reasonably precise digits and to continue calculation with the remaining rather imprecise digits.
- Comparisons to floating point constants may not be "exact" and as a consequence loops may not end where they are expected to end.

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, y; int c;

    x = 2.0 / 0.0;
    printf("2.0 / 0.0 = %e\n", x);                         /* inf */

    x = INFINITY + 1;
    printf("INFINITY + 1 = %e\n", x);                      /* inf */

    x = INFINITY - INFINITY;
    printf("INFINITY - INFINITY = %e\n", x);               /* nan */

    x = (1 + 1e20) - 1e20;
    y = 1 + (1e20 - 1e20);
    printf("%g == %g\n", x, y);                            /* 0 == 1 */

    for (x = 0.0, c= 0; x < 1.0; x += 0.1) c++;
    printf("%d == 10\n", c);                               /* 11 == 10 */

    x = 10.0 / 9.0;
    printf("%.50g\n", x);        /* 1.1111111111111111604543566500069573521614074707031 */
    return 0;
}
```

The pogram produces the following output:

```
2.0 / 0.0 = inf
INFINITY + 1 = inf
INFINITY - INFINITY = nan
0 == 1
11 == 10
1.1111111111111111604543566500069573521614074707031
```

Further online information:

- https://www.cl.cam.ac.uk/teaching/1011/FPComp/fpcomp10slides.pdf

Section 13: International System of Units

9 Natural Numbers

10 Integer Numbers

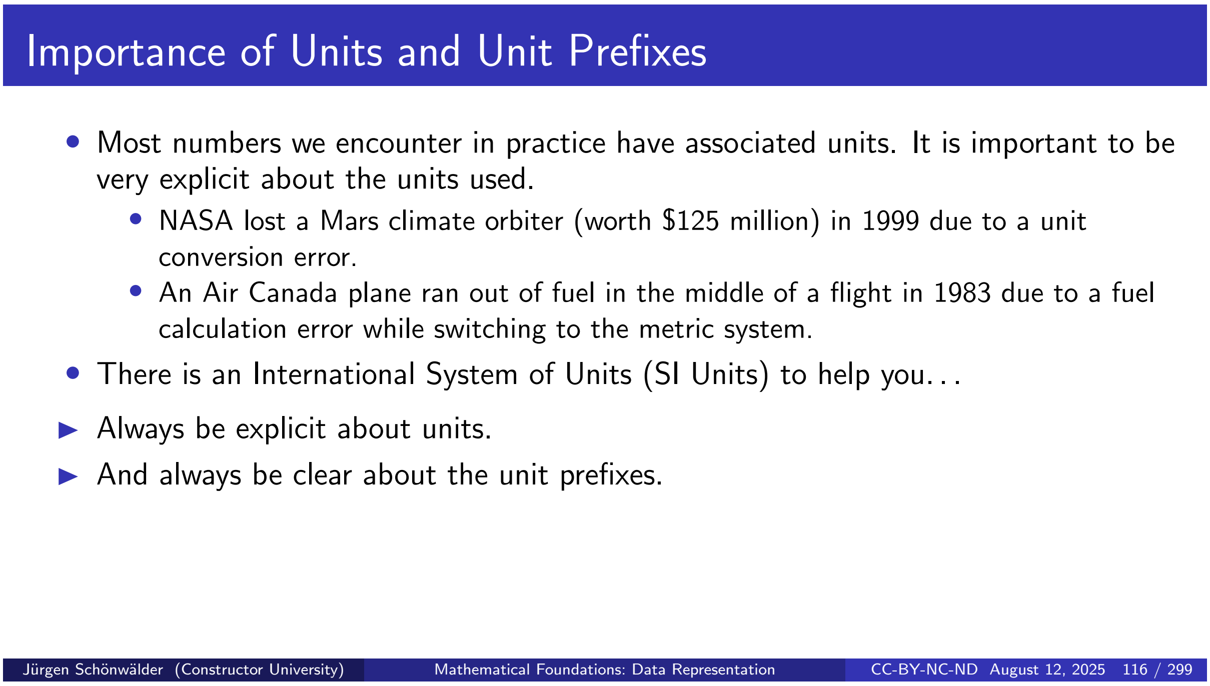
11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time



Importance of Units and Unit Prefixes

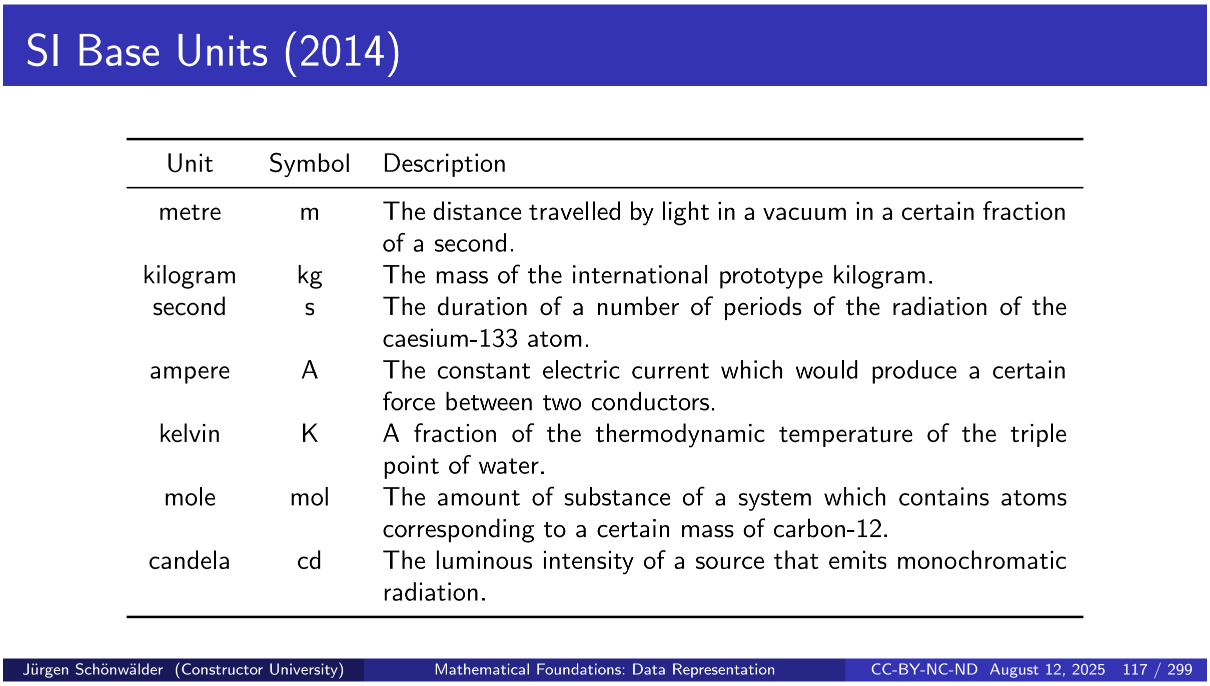
- Most numbers we encounter in practice have associated units. It is important to be very explicit about the units used.
  - NASA lost a Mars climate orbiter (worth $125 million) in 1999 due to a unit conversion error.
  - An Air Canada plane ran out of fuel in the middle of a flight in 1983 due to a fuel calculation error while switching to the metric system.
- There is an International System of Units (SI Units) to help you. . .

▶ Always be explicit about units.

▶ And always be clear about the unit prefixes.

Further online information:

- **Wikipedia**: Mars Climate Orbiter
- **Wikipedia**: Gimli Glider



SI Base Units (2014)

Unit	Symbol	Description
metre	m	The distance travelled by light in a vacuum in a certain fraction of a second.
kilogram	kg	The mass of the international prototype kilogram.
second	s	The duration of a number of periods of the radiation of the caesium-133 atom.
ampere	A	The constant electric current which would produce a certain force between two conductors.
kelvin	K	A fraction of the thermodynamic temperature of the triple point of water.
mole	mol	The amount of substance of a system which contains atoms corresponding to a certain mass of carbon-12.
candela	cd	The luminous intensity of a source that emits monochromatic radiation.

The slide summarizes the 2014 edition of the definition of the base units [1]. A new revision of the definition of the base units appeared in 2019 [2]. The SI base units were redefined in terms of natural physical constants, rather than human artefacts such as the standard kilogram.

Further online information:

- **Wikipedia**: SI Base Unit

SI Derived Units (2014)

- Many important units can be derived from the base units. Some have special names, others are simply defined by a formula over their base units. Some examples:

Name	Symbol	Definition	Description
herz	Hz	s^{-1}	frequency
newton	N	$\mathrm{kg\,m\,s}^{-1}$	force
watt	W	$\mathrm{kg\,m}^{2}\,\mathrm{s}^{-3}$	power
volt	V	$\mathrm{kg\,m}^{2}\,\mathrm{s}^{-3}\,\mathrm{A}^{-1}$	voltage
ohm	Ω	$\mathrm{kg\,m}^{2}\,\mathrm{s}^{-3}\,\mathrm{A}^{-2}$	resistance
velocity		$\mathrm{m\,s}^{-1}$	speed

Further online information:

- **Wikipedia**: SI Derived Unit

Metric Prefixes (International System of Units)

Name	Symbol	Base 10	Base 1000	Value
kilo	k	10^{3}	1000^{1}	1000
mega	M	10^{6}	1000^{2}	1 000 000
giga	G	10^{9}	1000^{3}	1 000 000 000
tera	T	10^{12}	1000^{4}	1 000 000 000 000
peta	P	10^{15}	1000^{5}	1 000 000 000 000 000
exa	E	10^{18}	1000^{6}	1 000 000 000 000 000 000
zetta	ζ	10^{21}	1000^{7}	1 000 000 000 000 000 000 000
yotta	Y	10^{24}	1000^{8}	1 000 000 000 000 000 000 000 000

Metric Prefixes (International System of Units)

Name	Symbol	Base 10	Base 1000	Value
milli	m	10^{-3}	1000^{-1}	0.001
micro	µ	10^{-6}	1000^{-2}	0.000 001
nano	n	10^{-9}	1000^{-3}	0.000 000 001
pico	p	10^{-12}	1000^{-4}	0.000 000 000 001
femto	f	10^{-15}	1000^{-5}	0.000 000 000 000 001
atto	a	10^{-18}	1000^{-6}	0.000 000 000 000 000 001
zepto	z	10^{-21}	1000^{-7}	0.000 000 000 000 000 000 001
yocto	y	10^{-24}	1000^{-8}	0.000 000 000 000 000 000 000 001

Binary Prefixes

Name	Symbol	Base 2	Base 1024	Value
kibi	Ki	2^{10}	1024^1	1024
mebi	Mi	2^{20}	1024^2	1 048 576
gibi	Gi	2^{30}	1024^3	1 073 741 824
tebi	Ti	2^{40}	1024^4	1 099 511 627 776
pebi	Pi	2^{50}	1024^5	1 125 899 906 842 624
exbi	Ei	2^{60}	1024^6	1 152 921 504 606 846 976
zebi	Zi	2^{70}	1024^7	1 180 591 620 717 411 303 424
yobi	Yi	2^{80}	1024^8	1 208 925 819 614 629 174 706 176

There is often confusion about metric and binary prefixes since metric prefixes are sometimes incorrectly used to refer to binary prefixes. Storage devices are a good example where this has led to serious confusion.

Computers generally access storage using addresses with an address range that is a power of two. Hence, with 30 bits, we can address $2^{30}\,\mathrm{B} = 1\,073\,741\,824\,\mathrm{B}$, or $1\,\mathrm{GiB}$. The industry, however, preferred to use the metric prefix system (well, to be fair, there was no binary prefix system initially), hence they used $1\,\mathrm{GB}$, which is $10^9\,\mathrm{B} = 1\,000\,000\,000\,\mathrm{B}$. The difference is $73\,741\,824\,\mathrm{B}$ (almost $7\,\%$ of $1\,\mathrm{GiB}$).

The binary prefixes, proposed in 2000, help to avoid any confusion. However, the adoption is rather slow and hence we will likely have to live with the confusion for many years to come. But of course, you can make a difference by always using the correct prefixes.

Section 14: Characters and Strings

9 Natural Numbers

10 Integer Numbers

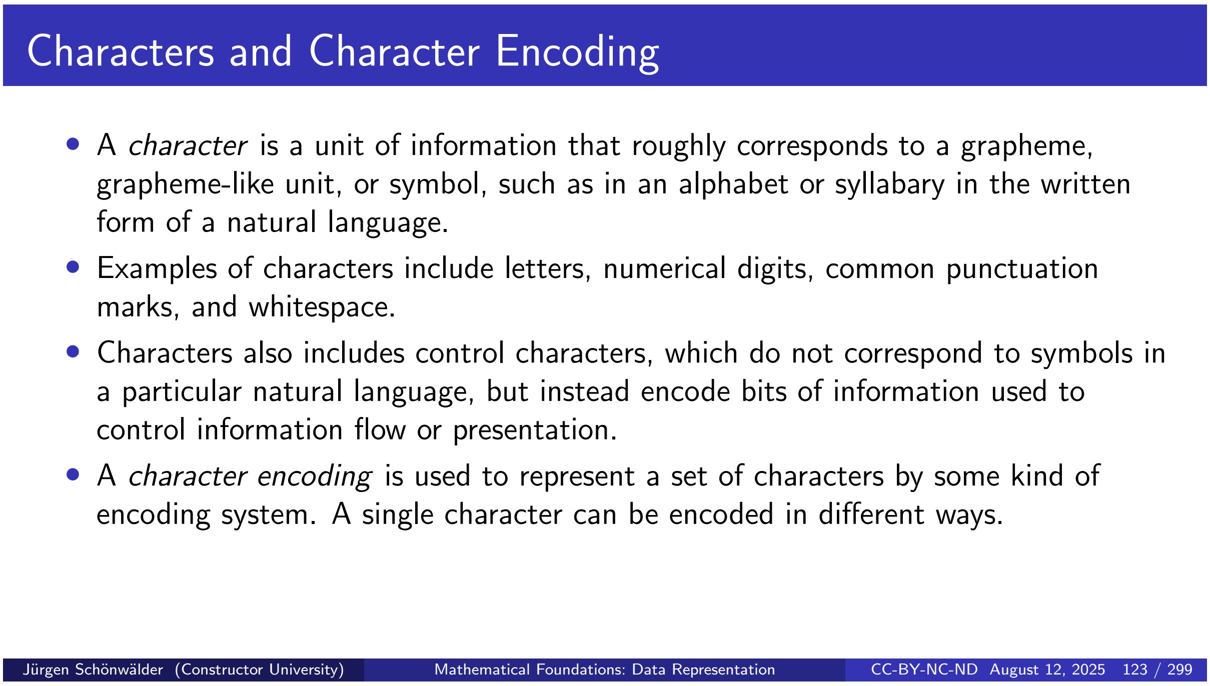
11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time



Characters and Character Encoding

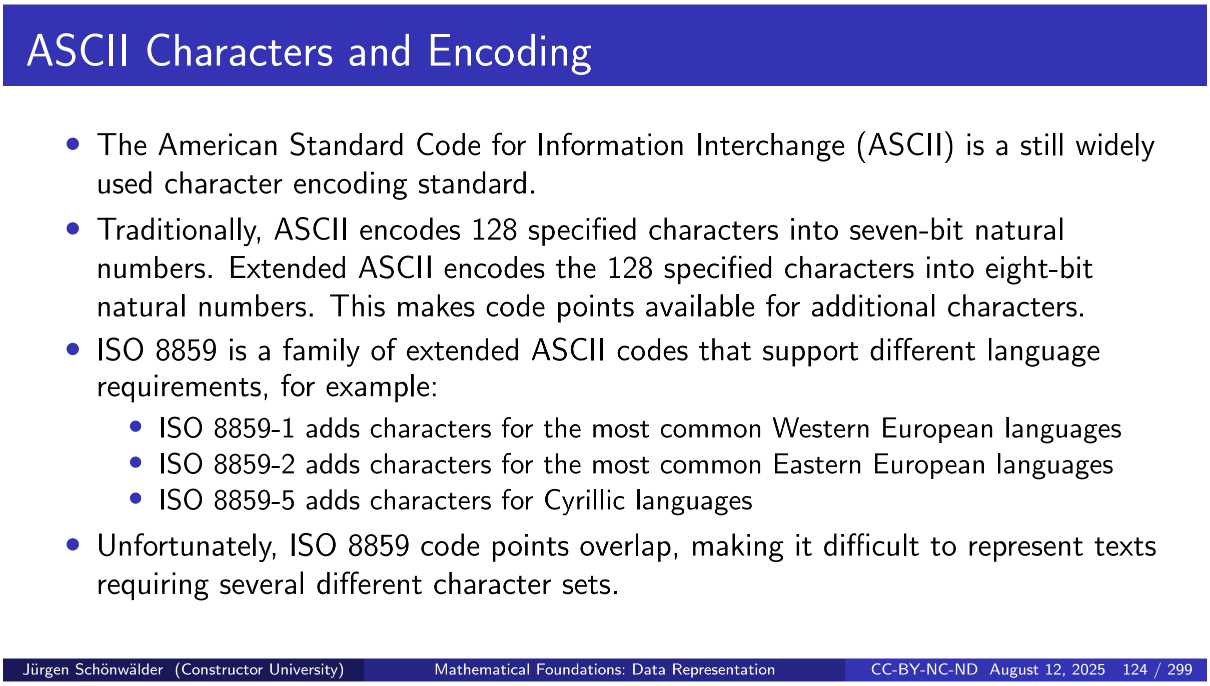
- A *character* is a unit of information that roughly corresponds to a grapheme, grapheme-like unit, or symbol, such as in an alphabet or syllabary in the written form of a natural language.
- Examples of characters include letters, numerical digits, common punctuation marks, and whitespace.
- Characters also includes control characters, which do not correspond to symbols in a particular natural language, but instead encode bits of information used to control information flow or presentation.
- A *character encoding* is used to represent a set of characters by some kind of encoding system. A single character can be encoded in different ways.

Please note again the distinction between a character and the representation or encoding of a character. There can be many different representations or encodings for the same character.

Characters are often read by humans and not just machines. This adds another problem since some characters (or character sequences) may look similar for a human reader while they are different from a program's perspective. This can lead to confusion on both sides, the human and the machine processing human input.

Further online information:

- **YouTube**: Plain Text - Dylan Beattie - NDC Copenhagen 2022



ASCII Characters and Encoding

- The American Standard Code for Information Interchange (ASCII) is a still widely used character encoding standard.
- Traditionally, ASCII encodes 128 specified characters into seven-bit natural numbers. Extended ASCII encodes the 128 specified characters into eight-bit natural numbers. This makes code points available for additional characters.
- ISO 8859 is a family of extended ASCII codes that support different language requirements, for example:
  - ISO 8859-1 adds characters for the most common Western European languages
  - ISO 8859-2 adds characters for the most common Eastern European languages
  - ISO 8859-5 adds characters for Cyrillic languages
- Unfortunately, ISO 8859 code points overlap, making it difficult to represent texts requiring several different character sets.

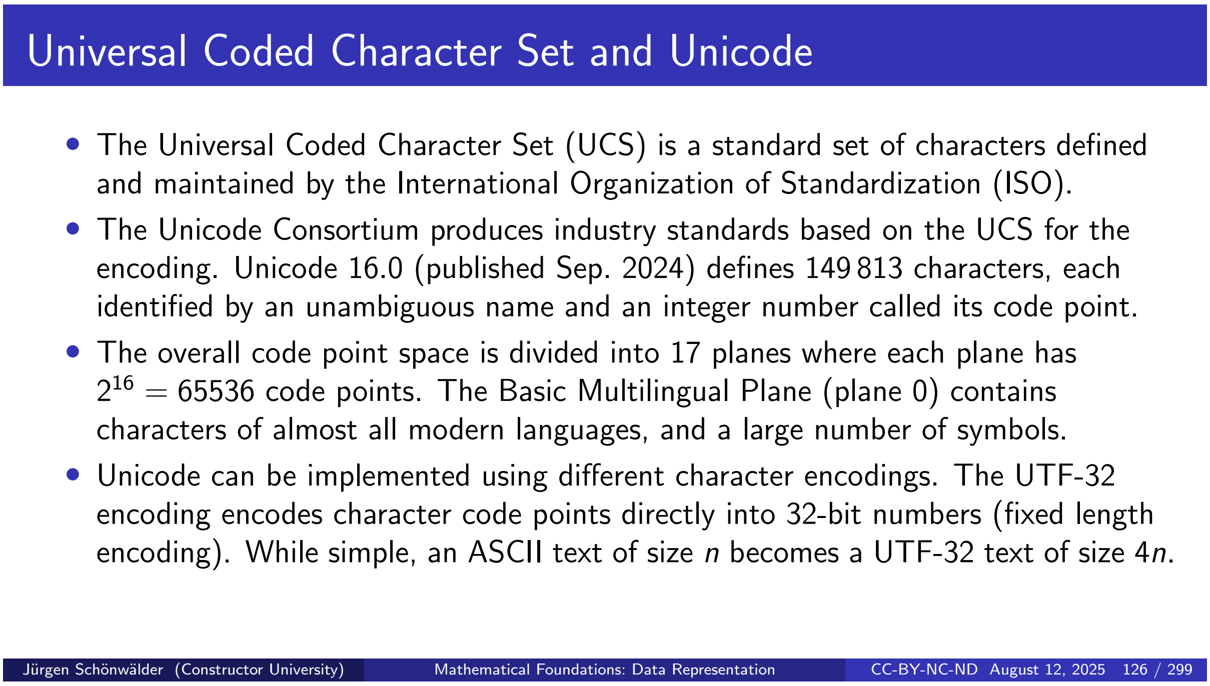
Using the ISO 8859 character code sets has been difficult since the information how the extended ASCII code points have to be interpreted was depending on the context. Hence, a text copied from one computer to another could become almost unreadable.

ASCII Characters and Code Points (decimal)

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	ˆ	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	˜	127	del

ASCII control characters:

dec	hex	name	description
0	00	NUL	null character
1	01	SOH	start of heading
2	02	STX	start of text
3	03	ETX	end of text
4	04	EOT	end of transmission
5	05	ENQ	enquiry
6	06	ACK	acknowledge
7	07	BEL	bell
8	08	BS	backspace
9	09	HT	horizontal tab
10	0A	LF	new line
11	0B	VT	vertical tab
12	0C	FF	form feed
13	0D	CR	carriage ret
14	0E	SO	shift out
15	0F	SI	shift in
16	10	DLE	data link escape
17	11	DC1	device control 1
18	12	DC2	device control 2
19	13	DC3	device control 3
20	14	DC4	device control 4
21	15	NAK	negative acknowledgment
22	16	SYN	synchronous idle
23	17	ETB	end of transmission blk
24	18	CAN	cancel
25	19	EM	end of medium
26	1A	SUB	substitute
27	1B	ESC	escape
28	1C	FS	file separator
29	1D	GS	group separator
30	1E	RS	record separator
31	1F	US	unit separator



Universal Coded Character Set and Unicode

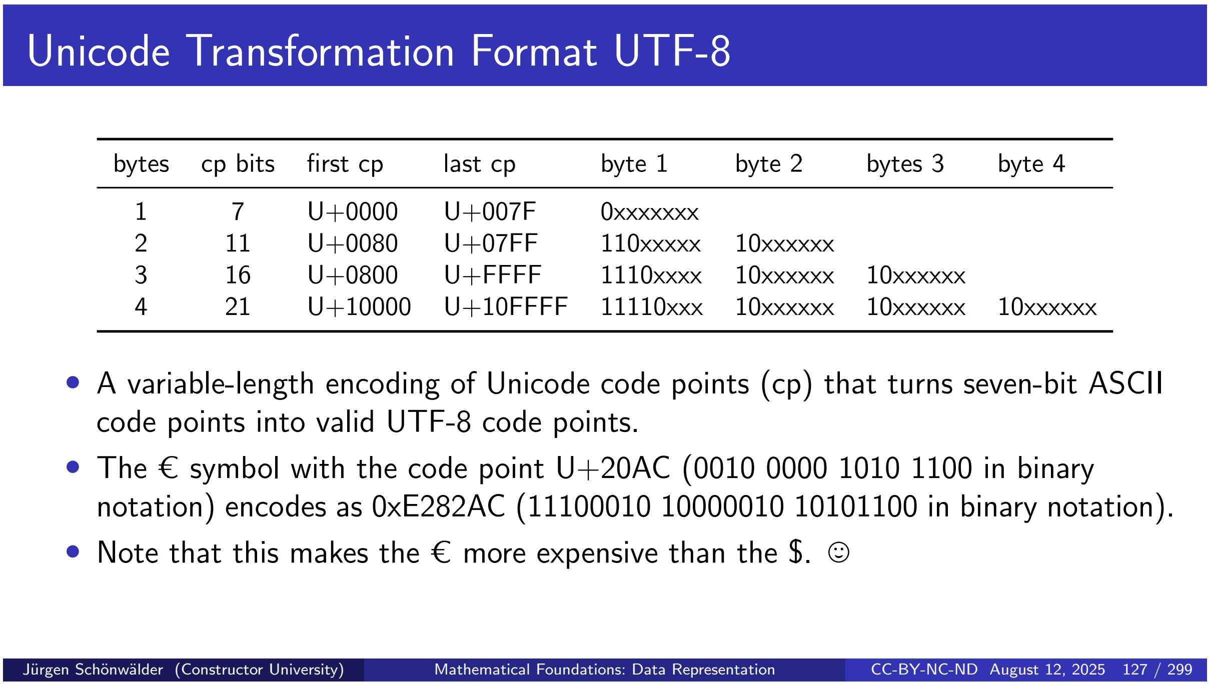
- The Universal Coded Character Set (UCS) is a standard set of characters defined and maintained by the International Organization of Standardization (ISO).
- The Unicode Consortium produces industry standards based on the UCS for the encoding. Unicode 16.0 (published Sep. 2024) defines 149 813 characters, each identified by an unambiguous name and an integer number called its code point.
- The overall code point space is divided into 17 planes where each plane has $2^{16} = 65536$ code points. The Basic Multilingual Plane (plane 0) contains characters of almost all modern languages, and a large number of symbols.
- Unicode can be implemented using different character encodings. The UTF-32 encoding encodes character code points directly into 32-bit numbers (fixed length encoding). While simple, an ASCII text of size n becomes a UTF-32 text of size $4n$.

Some programming languages natively support unicode while others require the use of unicode libraries. In general, unicode is not trivial to program with.

- Unicode characters are categorized and they have properties that go beyond a simple classification into numbers, letters, etc.
- Unicode characters and strings require normalization since some symbols may be represented in several different ways. Hence, in order to compare unicode characters, it is important to choose a suitable normalization form.
- Unicode characters can be encoded in several different formats and this requires character and string conversion functionality.
- Case mappings are not trivial since certain characters do not have a matching lowercase and uppercase representation. Some case conversions are also language specific and not character specific.

The GNU libunistring library is an example of a Unicode string library for C programmers.

In modern programming languages like Rust or Go, characters (or runes) are Unicode code points and strings are usually UTF-8 encoded sequences of Unicode characters.



Unicode Transformation Format UTF-8

bytes	cp bits	first cp	last cp	byte 1	byte 2	bytes 3	byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- A variable-length encoding of Unicode code points (cp) that turns seven-bit ASCII code points into valid UTF-8 code points.
- The € symbol with the code point U+20AC (0010 0000 1010 1100 in binary notation) encodes as 0xE282AC (11100010 10000010 10101100 in binary notation).
- Note that this makes the € more expensive than the $. ☺

The UTF-8 format is widely used when texts are exchanged between programs or over the Internet. It is defined in RFC 3629 [5], which was published in January 1998.

Strings

- Let Σ be a non-empty finite set of symbols (or characters), called the alphabet.
- A string (or word) over Σ is any finite sequence of symbols from Σ, including (of course) the empty sequence.
- Typical operations on strings are length(), concatenation(), reverse(), ...
- There are different ways to store strings. Two common approaches are:
  - The character sequence is *null-terminated*, i.e., the sequence of characters is followed by a special terminating NUL character.
  - The sequence is *length-prefixed*, i.e., a natural number indicating the length of the character sequence is stored in front of the characters.
- In some programming languages, you need to know how strings are stored, in other languages you happily leave the details to the language implementation.

In C and C++, a simple (usually ASCII) string is a null-terminated sequence of characters. C programmers have to make sure that always sufficient memory is allocated to store the terminating byte.

In modern programming languages, characters are usually Unicode characters. In Rust, a character is an unsigned 32-bit number. But note that Rust strings are UTF-8 encoded unicode character sequences stored in a buffer with a controlled length and capacity.

Note that for modern pogramming language, the exact space required to store a string with a given number of characters (a given length) may not be a simple mapping from the string length.

Section 15: Date and Time

9 Natural Numbers

10 Integer Numbers

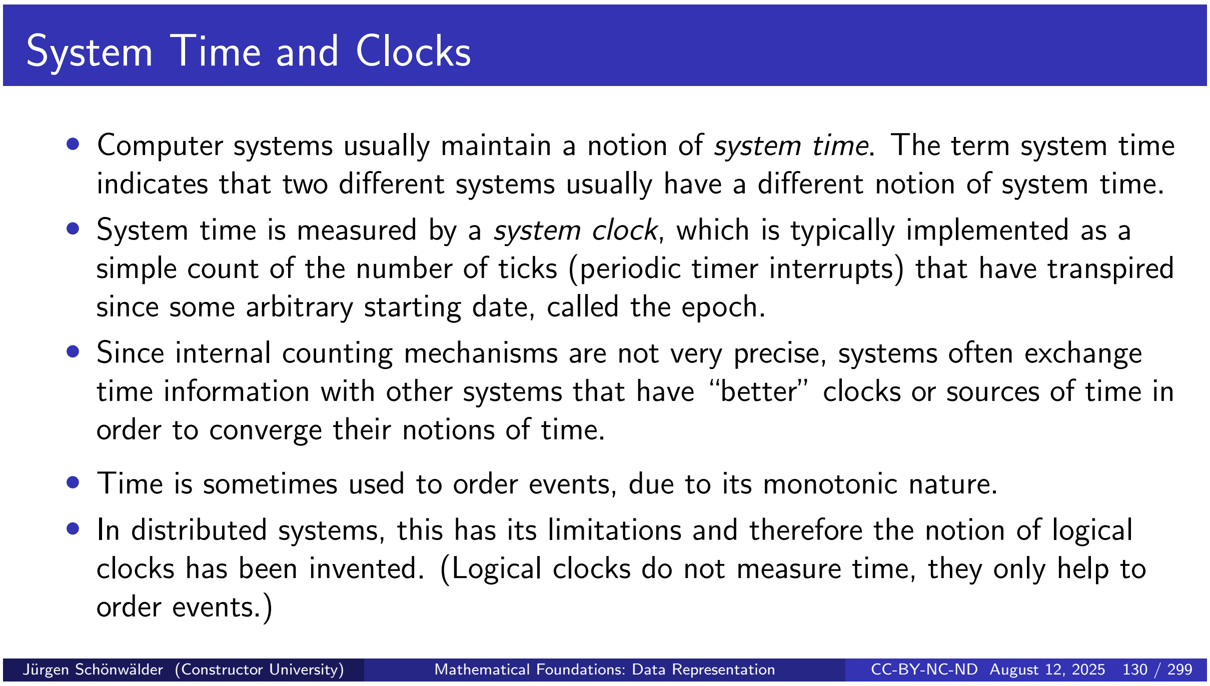
11 Rational and Real Numbers

12 Floating Point Numbers

13 International System of Units

14 Characters and Strings

15 Date and Time



System Time and Clocks

- Computer systems usually maintain a notion of *system time*. The term system time indicates that two different systems usually have a different notion of system time.
- System time is measured by a *system clock*, which is typically implemented as a simple count of the number of ticks (periodic timer interrupts) that have transpired since some arbitrary starting date, called the epoch.
- Since internal counting mechanisms are not very precise, systems often exchange time information with other systems that have "better" clocks or sources of time in order to converge their notions of time.
- Time is sometimes used to order events, due to its monotonic nature.
- In distributed systems, this has its limitations and therefore the notion of logical clocks has been invented. (Logical clocks do not measure time, they only help to order events.)

Most computer systems have relatively poor clocks and hence the notion of a computer's time drifts quickly. Time synchronization protocols are commonly used to synchronize a computer's notion of time with a more robust time source over a network. A widely deployed time synchronization protocol is the Network Time Protocol (NTP). An alternative solution is the Precision Time Protocol (PTP). A common problem of these protocols is that the exchange of synchronization messages itself introduces errors that must be compensated for. Another commonly available source of time is the Global Positioning System (GPS).

Unix systems represent time as the number of seconds that have passed since the 1st of January 1970 00:00:00 UTC, a meanwhile pretty large number of type `time_t`. Traditionally, a 32-bit signed number has been used to represent a `time_t`. The maximum positive 32-bit signed integer number will be reached on Tuesday, 19 January 2038 at 03:14:07 UTC and then the number will warp, leading to dates starting sometime on 13 December 1901. This is known as the "year 2038 problem". Many operating systems running on 64-bit hardware moved to 64-bit signed integers for `time_t`, which solves the problem. However, embedded systems often still use 32-bit signed integers as `time_t` and they will be vulnerable when we hit the year 2038 (less than 20 years left).

In Rust, you can get the system time in the following way:

```rust
use std::time::SystemTime;

fn main() {
    let now = SystemTime::now();
    println!("{:?}", now);
}
```

Further online information:

- **Wikipedia**: Network Time Protocol
- **Wikipedia**: Precision Time Protocol
- **Wikipedia**: Year 2038 Problem

Calendar Time

- System time can be converted into *calendar time*, a reference to a particular time represented within a calendar system.
- A popular calendar is the *Gregorian calendar*, which maps a time reference into a year, a month within the year, and a day within a month.
- The Gregorian calendar was introduced by Pope Gregory XIII in October 1582.
- The Coordinated Universal Time (UTC) is the primary time standard by which the world regulates clocks and time.
- Due to the rotation of the earth, days start and end at different moments. This is reflected by the notion of a *time zone*, which is essentially an offset to UTC.
- The number of time zones is not static and time zones change occasionally.

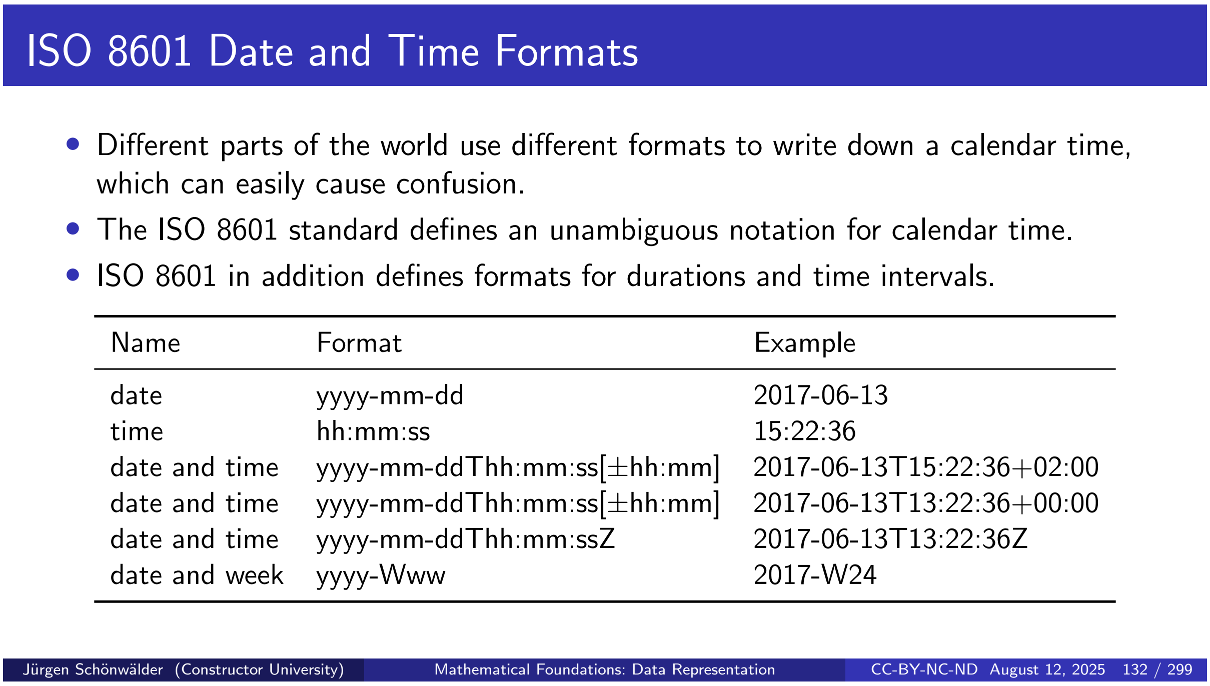
Computer systems often indicate time zones using time zone names. Associated to a time zone name are usually complicated rules that indicate for example transitions to daylight saving times or simply changes to the time zones. IANA is maintaining a time zone database that is commonly used by software systems to interpret time zone names in order to derive the correct time zone offset from UTC.

In Rust, you can get the calendar time in the following way:

```rust
use chrono::prelude::*;

fn main() {
    let local: DateTime<Local> = Local::now();
    println!("{:?}", local);
}
```

Further online information:

- **Wikipedia**: Time Zone
- https://www.iana.org/time-zones
- **YouTube**: The Problem with Time & Timezones - Computerphile



ISO 8601 Date and Time Formats

- Different parts of the world use different formats to write down a calendar time, which can easily cause confusion.
- The ISO 8601 standard defines an unambiguous notation for calendar time.
- ISO 8601 in addition defines formats for durations and time intervals.

Name	Format	Example
date	yyyy-mm-dd	2017-06-13
time	hh:mm:ss	15:22:36
date and time	yyyy-mm-ddThh:mm:ss[±hh:mm]	2017-06-13T15:22:36+02:00
date and time	yyyy-mm-ddThh:mm:ss[±hh:mm]	2017-06-13T13:22:36+00:00
date and time	yyyy-mm-ddThh:mm:ssZ	2017-06-13T13:22:36Z
date and week	yyyy-Www	2017-W24

The special letter Z indicates that the date and time is in UTC time. The ISO 8601 standard deals with timezone offsets by specifying the positive or negative offset from UTC time in hours and minutes. This in principle allows us to define +00:00 and -00:00 but the ISO 8601 disallows a negative zero offset. RFC 3339 [10], a profile of ISO 8601, however, suggested that -00:00 indicates that the time is in UTC and that the local timezone offset is unknown. RFC 9557 [18] updates RFC 3339, suggesting that the special letter Z is used to indicate that the time is in UTC and the offset to local time is unknown while +00:00 is used to indicate that the time and the reference point is in UTC.

Further online information:

- **Wikipedia**: ISO 8601
- **xkcd**: ISO 8601

Part IV

Boolean Algebra

Boolean algebra is the mathematical framework for describing anything that is *binary*, that is, anything which can have only two values. Boolean algebra is the foundation for understanding digital circuits, the foundation of today's computers. The central processing unit (CPU) of a computer is essentially a very large collection of digital circuits built from logic gates. Hence, the mathematical foundation of a CPU is Boolean algebra. Boolean algebra helps us to understand how digital circuits can be composed and where necessary verified.

By the end of this part, students should be able to

- understand the concept of Boolean variables and their interpretation;
- explain elementary boolean functions such as $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\underline{\vee}$, $\overline{\wedge}$, $\overline{\vee}$;
- illustrate how Boolean functions relate to Boolean expressions;
- differentiate between the syntax and the semantics of Boolean expressions;
- apply Boolean equivalence laws;
- describe the notion of tautologies and contradictions;
- convert Boolean expressions into conjunctive and disjunctive normal forms;
- obtain Boolean expressions in normal form from truth tables;
- apply the Quine McCluskey algorithm to minimize Boolean expressions.

Section 16: Boolean Variables and Elementary Functions

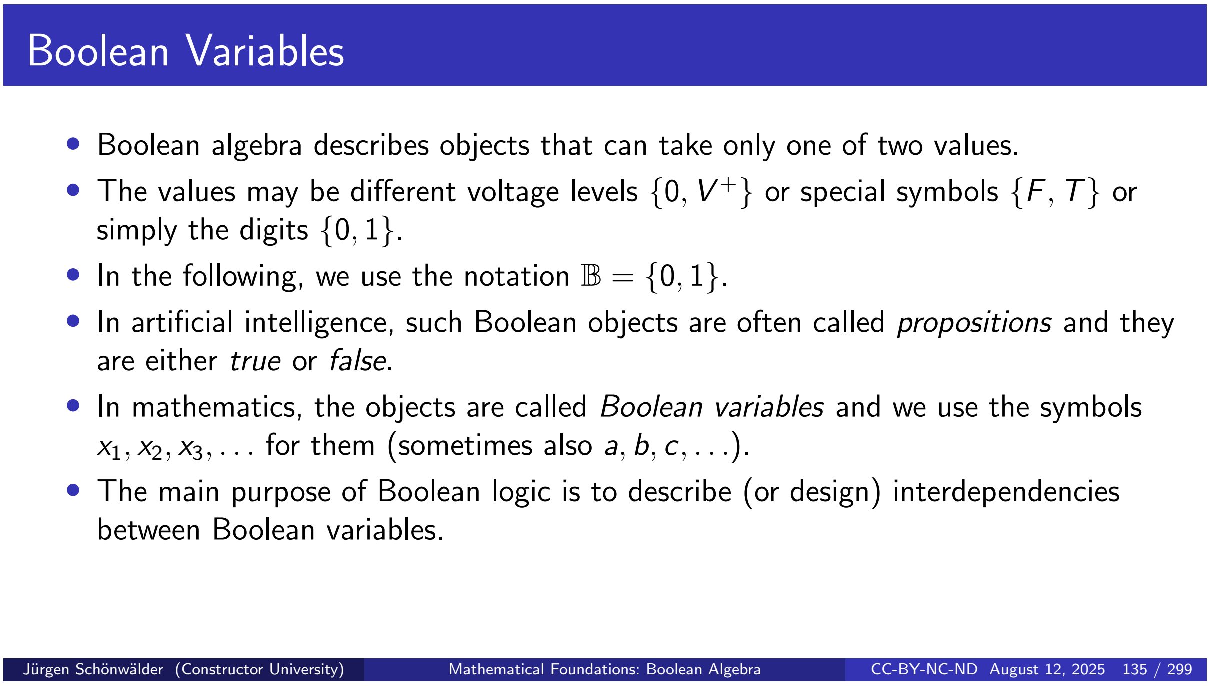
16 Boolean Variables and Elementary Functions

17 Boolean Functions and Formulas

18 Boolean Algebra Equivalence Laws

19 Conjunctive and Disjunctive Normal Forms

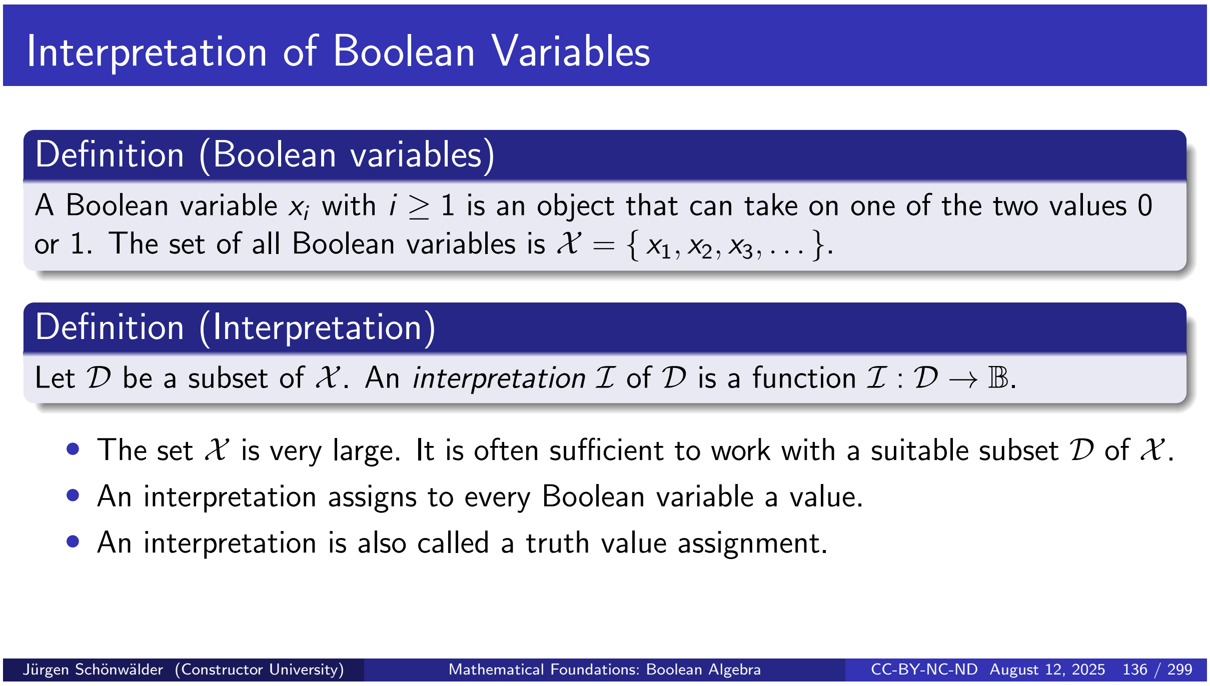
20 Complexity of Boolean Formulas



Boolean Variables

- Boolean algebra describes objects that can take only one of two values.
- The values may be different voltage levels $\{0, V^+\}$ or special symbols $\{F, T\}$ or simply the digits $\{0, 1\}$.
- In the following, we use the notation $\mathbb{B} = \{0, 1\}$.
- In artificial intelligence, such Boolean objects are often called *propositions* and they are either *true* or *false*.
- In mathematics, the objects are called *Boolean variables* and we use the symbols $x_1, x_2, x_3, \ldots$ for them (sometimes also $a, b, c, \ldots$).
- The main purpose of Boolean logic is to describe (or design) interdependencies between Boolean variables.

The name goes back to George Bool (2 November 1815 – 8 December 1864), who first defined an algebraic system of logic in the 19th century.



Interpretation of Boolean Variables

Definition (Boolean variables)

A Boolean variable x_i with $i \geq 1$ is an object that can take on one of the two values 0 or 1. The set of all Boolean variables is $\mathcal{X} = \{x_1, x_2, x_3, \ldots\}$.

Definition (Interpretation)

Let $\mathcal{D}$ be a subset of $\mathcal{X}$. An *interpretation* $\mathcal{I}$ of $\mathcal{D}$ is a function $\mathcal{I} : \mathcal{D} \to \mathbb{B}$.

- The set $\mathcal{X}$ is very large. It is often sufficient to work with a suitable subset $\mathcal{D}$ of $\mathcal{X}$.
- An interpretation assigns to every Boolean variable a value.
- An interpretation is also called a truth value assignment.

We have already know how to represent integer numbers in a certain range as n-bit binary numbers and we have discussed that the addition of binary numbers can be done like we are used to for decimal numbers. For example, the addition of 12 and 5 in b2n8 can be carried out as follows:

```
  00001100
+ 00000101
      11
----------
  00010001
```

More generically, we have a number $a = a_7a_6a_5a_4a_3a_2a_1a_0$ and a number $b = b_7b_6b_5b_4b_3b_2b_1b_0$ and we calculate the sum $s = s_7s_6s_5s_4s_3s_2s_1s_0$, which requires to also determine the carry bits $c = c_8c_7c_6c_5c_4c_3c_2c_1$. To build a digital circuit to carry out additions, we need to specify how the bits of the sum s and the carry bits c are determined from the input bits a and b. Mathematically, we can consider each input bit and each output bit a Boolean variable and we are interested in Boolean functions mapping the input bits to the output bits. As a first step, we may regard every output bit a Boolean function of all input bits.

In the C programming language, a boolean value is traditionally represented by an `int`, which is 0 if the boolean value is false and unequal to zero if the boolean value is true. While this approach is sometimes convenient, it also creates problems since two truth values may not compare. The 1999 edition of C allows to resolve this by using the bool type defined in `<stdbool.h>`.

```c
int main(void)
{
    int x = 2;
    if (x == (x || !x)) {
        return 0;
    }
    return 1;
}
```

```c
#include <stdbool.h>

int main(void)
{
    bool x = 2;
    if (x == (x || !x)) {
        return 0;
    }
    return 1;
}
```

Boolean ∧ Function (and)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

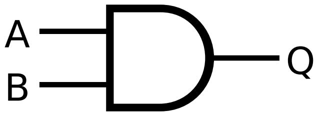
- The logical *and* (∧) can be viewed as a function mapping two Boolean values to a Boolean value:

$$\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$

- A truth table defines a Boolean operation (or function) by listing the result for all possible arguments.

- Many programming languages like C or C++ (or Rust or Haskell) use the operator `&&` to represent the ∧ function. (Python uses the `and` keyword.)

Digital systems represent the ∧ function using AND gates, which can be built using transistors or diodes. Below is a symbol for a logical AND gate taking the two inputs A and B and producing the output Q.



An interesting difference of a logical gate to a boolean function is that a gate has a time delay; changes of the inputs lead to changes of the output with a certain gate delay. Hence, digital circuits need to be designed with the gate delays in mind to reduce overall delays and to avoid undesirable instabilities.

Boolean ∨ Function (or)

x	y	$x \lor y$
0	0	0
0	1	1
1	0	1
1	1	1

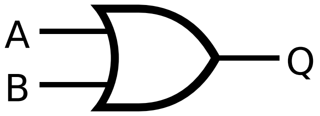
- The logical *or* ($\lor$) can be viewed as a function mapping two Boolean values to a Boolean value:

$$\lor : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$

- Each row in the truth table corresponds to one interpretation.
- A truth table simply lists all possible interpretations.

- Many programming languages like C or C++ (or Rust or Haskell) use the operator `||` to represent the $\lor$ function. (Python uses the `or` keyword.)

Digital systems represent the $\lor$ function using OR gates, which can be built using transistors or diodes. Below is a symbol for a logical OR gate taking the two inputs A and B and producing the output Q.



Boolean ¬ Function (not)

x	$\neg x$
0	1
1	0

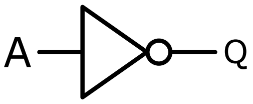
- The logical *not* (¬) can be viewed as a unary function mapping a Boolean value to a Boolean value:

$$\neg : \mathbb{B} \to \mathbb{B}$$

- The ¬ function applied to x is also written as $\overline{x}$.

- Many programming languages like C or C++ (or Rust) use the operator `!` to represent the ¬ function. (Python uses the `not` keyword while Haskell uses the function `not :: Bool -> Bool`).

Digital systems represent the ¬ function using NOT gates, which can be built using transistors or diodes. Below is a symbol for a logical NOT gate taking the two input A and producing the output Q.



Boolean $\rightarrow$ Function (implies)

x	y	$x \rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

- The logical *implication* ($\rightarrow$) can be viewed as a function mapping two Boolean values to a Boolean value:

$$\rightarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$

- The implication represents statements of the form "if x then y" (where x is called the precondition and y the consequence).

- The logical implication is often confusing to ordinary mortals. A logical implication is false only if the precondition is true, but the consequence it asserts is false.
- The claim "if cats eat dogs, then the sun shines" is logically true.

Boolean ↔ Function (equivalence)

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

- The logical *equivalence* ↔ can be viewed as a function mapping two Boolean values to a Boolean value:

$$\leftrightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$

- Many programming languages like C or C++ (or Rust or Haskell) use the operator == to represent the equivalence function.

Boolean $\veebar$ Function (exclusive or)

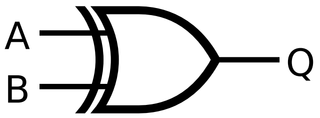
x	y	$x \veebar y$
0	0	0
0	1	1
1	0	1
1	1	0

- The logical *exclusive or* $\veebar$ can be viewed as a function mapping two Boolean values to a Boolean value:

$$\veebar : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$

- Another commonly used symbol for the exclusive or is $\oplus$.

Digital systems represent the $\veebar$ function using XOR gates, which can be built using transistors or diodes. Below is a symbol for a logical XOR gate taking the two inputs A and B and producing the output Q.



The exclusive or function has an interesting property. Lets consider the following example where we apply the exclusive or function bitwise to two longer bit strings:

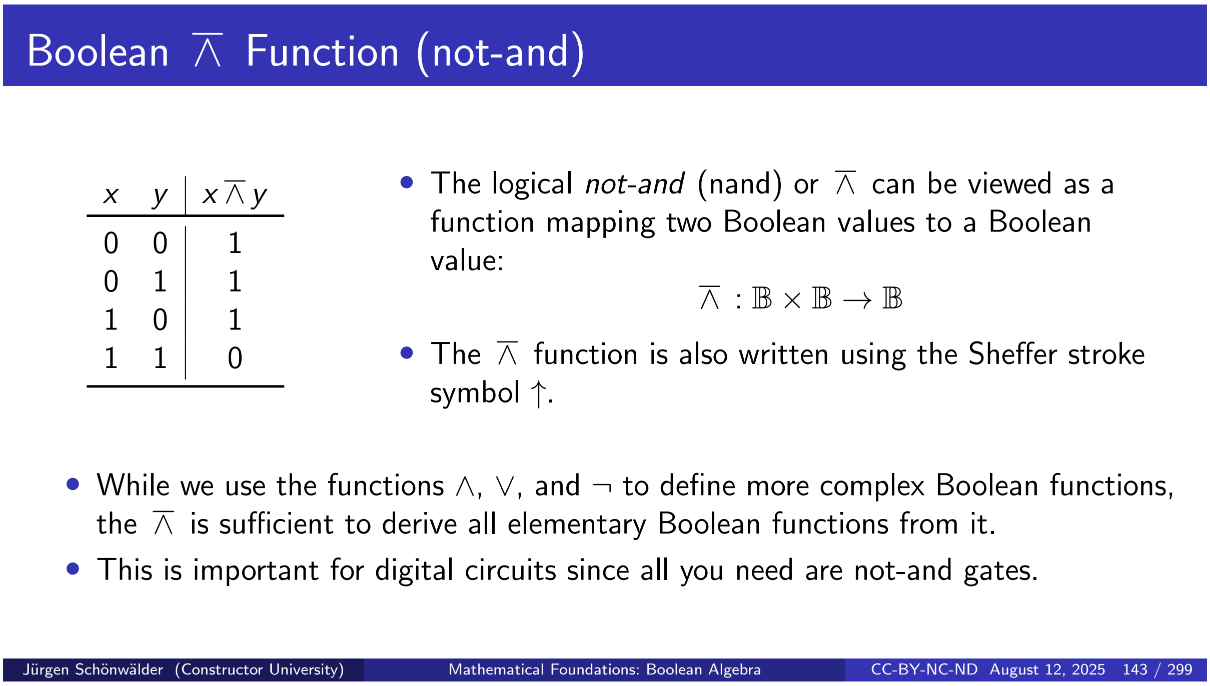
$$1234_{16} \veebar cafe_{16} = 0001\,0010\,0011\,0100_2 \veebar 1100\,1010\,1111\,1110_2 = 1101\,1000\,1100\,1010_2 = d8ca_{16}$$

If we perform an exclusive or on the result with the same second operand, we get the following:

$$d8ca_{16} \veebar cafe_{16} = 1101\,1000\,1100\,1010_2 \veebar 1100\,1010\,1111\,1110_2 = 0001\,0010\,0011\,0100_2 = 1234_{16}$$

Apparently, it seems that $(x \veebar y) \veebar y = x$. To prove this property, we can construct a truth table for all possible combinations:

x	y	$x \veebar y$	$(x \veebar y) \veebar y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1



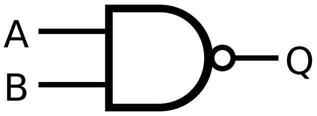
Boolean $\barwedge$ Function (not-and)

x	y	$x \barwedge y$
0	0	1
0	1	1
1	0	1
1	1	0

- The logical *not-and* (nand) or $\barwedge$ can be viewed as a function mapping two Boolean values to a Boolean value:
$$\barwedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$
- The $\barwedge$ function is also written using the Sheffer stroke symbol $\uparrow$.

- While we use the functions $\wedge$, $\vee$, and $\neg$ to define more complex Boolean functions, the $\barwedge$ is sufficient to derive all elementary Boolean functions from it.
- This is important for digital circuits since all you need are not-and gates.

Digital systems represent the $\barwedge$ function using NAND gates, which can be built using transistors or diodes. Below is a symbol for a logical NAND gate taking the two inputs A and B and producing the output Q.



The not-and is a universal function since it can be used to derive all elementary Boolean functions.

- $x \wedge y = (x \barwedge y) \barwedge (x \barwedge y)$

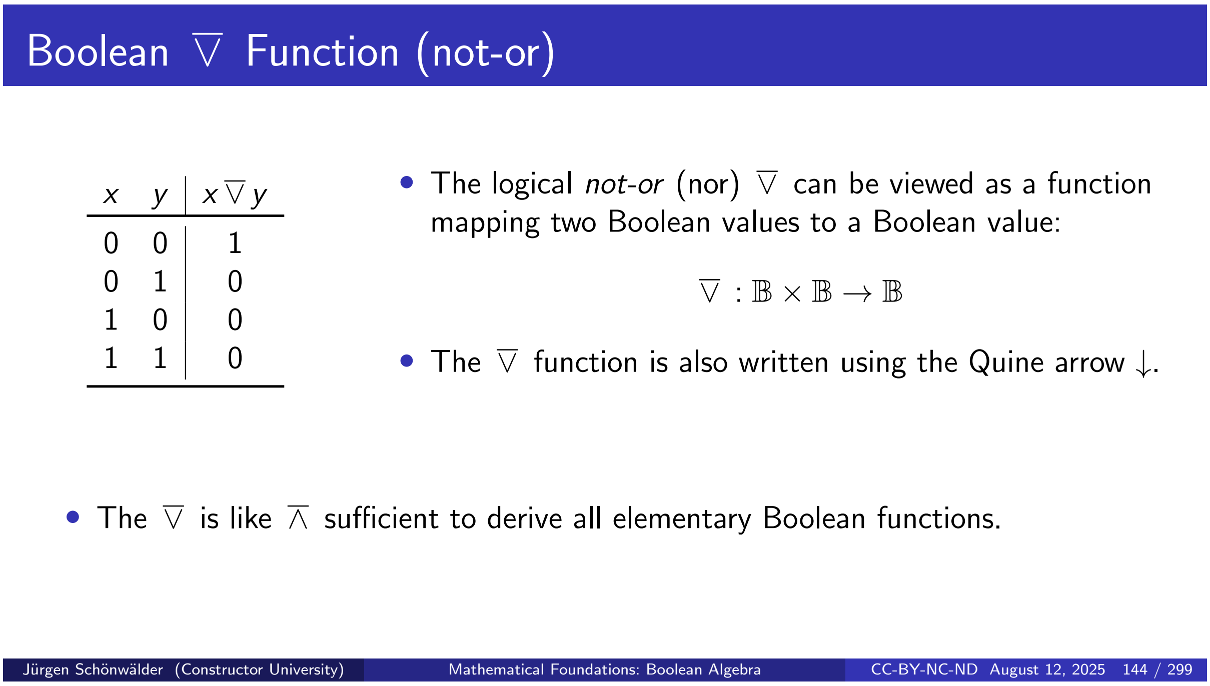
x	y	$x \wedge y$	$x \barwedge y$	$(x \barwedge y) \barwedge (x \barwedge y)$
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

- $x \vee y = (x \barwedge x) \barwedge (y \barwedge y)$

x	y	$x \vee y$	$x \barwedge x$	$y \barwedge y$	$(x \barwedge x) \barwedge (y \barwedge y)$
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	0	0	1

- $\neg x = x \barwedge x$

x	$\neg x$	$x \barwedge x$
0	1	1
1	0	0



Boolean $\overline{\vee}$ Function (not-or)

x	y	$x \overline{\vee} y$
0	0	1
0	1	0
1	0	0
1	1	0

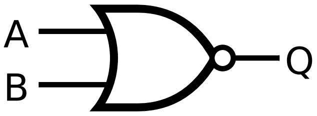
- The logical *not-or* (nor) $\overline{\vee}$ can be viewed as a function mapping two Boolean values to a Boolean value:

$$\overline{\vee} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$

- The $\overline{\vee}$ function is also written using the Quine arrow $\downarrow$.

- The $\overline{\vee}$ is like $\overline{\wedge}$ sufficient to derive all elementary Boolean functions.

Digital systems represent the $\overline{\vee}$ function using NOR gates, which can be built using transistors or diodes. Below is a symbol for a logical NOR gate taking the two inputs A and B and producing the output Q.



The not-or is a universal function since it can be used to derive all elementary Boolean functions.

- $x \wedge y = (x \overline{\vee} x) \overline{\vee} (y \overline{\vee} y)$
- $x \vee y = (x \overline{\vee} y) \overline{\vee} (x \overline{\vee} y)$
- $\neg x = x \overline{\vee} x$

Alternative Notations

function	mnemonic	mathematics	engineering	C / C++	C / C++ (bits)
and	x and y	$x \wedge y$	$x \cdot y$	`x && y`	`&`
or	x or y	$x \vee y$	$x + y$	`x || y`	`|`
not	not x	$\neg x$	$\overline{x}, x'$	`! x`	`~`
implication	x impl y	$x \rightarrow y$			
equivalence	x equiv y	$x \leftrightarrow y$		`x == y`	
exclusive-or	x xor y	$x \veebar y$	$x \oplus y$		`^`
not-and	x nand y	$x \overline{\wedge}\, y$, $x\overline{\wedge}y$	$\overline{x \cdot y}$		
not-or	x nor y	$x \overline{\vee}\, y$, $x\overline{\vee}y$	$\overline{x + y}$		

Different symbols are used in different contexts to denote basic boolean functions. Note that authors sometimes mix symbols; there is no standard notation as there are multiple quasi standards to choose from.

Section 17: Boolean Functions and Formulas

16 Boolean Variables and Elementary Functions

17 Boolean Functions and Formulas

18 Boolean Algebra Equivalence Laws

19 Conjunctive and Disjunctive Normal Forms

20 Complexity of Boolean Formulas

Boolean Functions

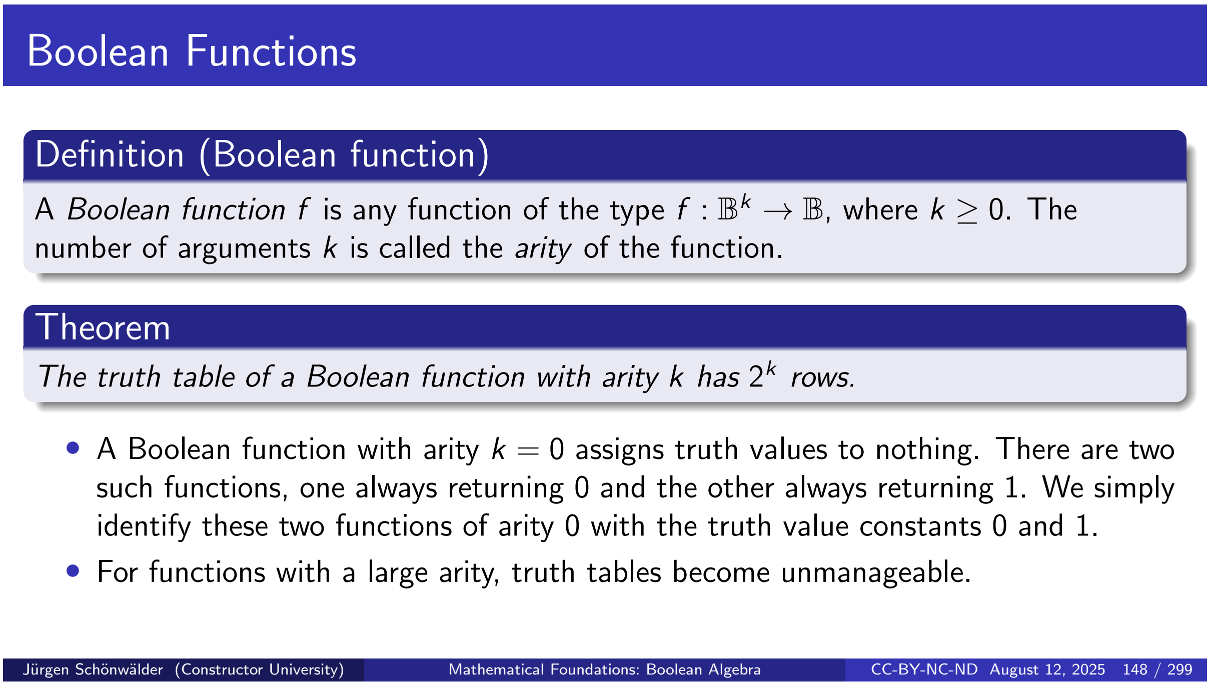
- Elementary Boolean functions $(\neg, \wedge, \vee)$ can be composed to define more complex functions.
- An example of a composed function is $f : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $(x, y) \mapsto \neg(x \wedge y)$. The meaning is "first compute the logical and of x and y, then apply $\neg$ on the result obtained."
- Boolean functions can take a large number of arguments. Here is a function $f : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ taking three arguments.
- We may define the function f using a shorthand notation:

$$f(x, y, z) = (\neg(x \wedge y) \vee (z \wedge y))$$

The left hand side of the notation above defines the function name and its arguments, the right hand side defines the function itself by means of a formula.

Below is the definition of all elementary Boolean functions we have introduced so far using just the Boolean functions $\wedge$, $\vee$, and $\neg$:

$\to (x, y) = \neg x \vee y$	implication
$\leftrightarrow (x, y) = (x \wedge y) \vee (\neg x \wedge \neg y)$	equivalence
$\veebar (x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$	exclusive-or
$\overline{\wedge} (x, y) = \neg(x \wedge y)$	not-and
$\overline{\vee} (x, y) = \neg(x \vee y)$	not-or



Boolean Functions

Definition (Boolean function)

A *Boolean function* f is any function of the type $f : \mathbb{B}^k \to \mathbb{B}$, where $k \geq 0$. The number of arguments k is called the *arity* of the function.

Theorem

The truth table of a Boolean function with arity k has 2^k rows.

- A Boolean function with arity $k = 0$ assigns truth values to nothing. There are two such functions, one always returning 0 and the other always returning 1. We simply identify these two functions of arity 0 with the truth value constants 0 and 1.
- For functions with a large arity, truth tables become unmanageable.

Boolean functions are interesting, since they can be used as computational devices. In particular, we can consider the CPU of a computer as a collection of Boolean functions (e.g., an arithmetic instruction of a modern 64-bit CPU can be viewed as a collection of 64 Boolean functions of arity 128: one function per output bit and each function taking two times 64 bits as input).

Another way to define a Boolean function is by writing down its truth table. We can define $f(x, y, z) = (\neg(x \wedge y) \vee (z \wedge y))$ by filling out the following table:

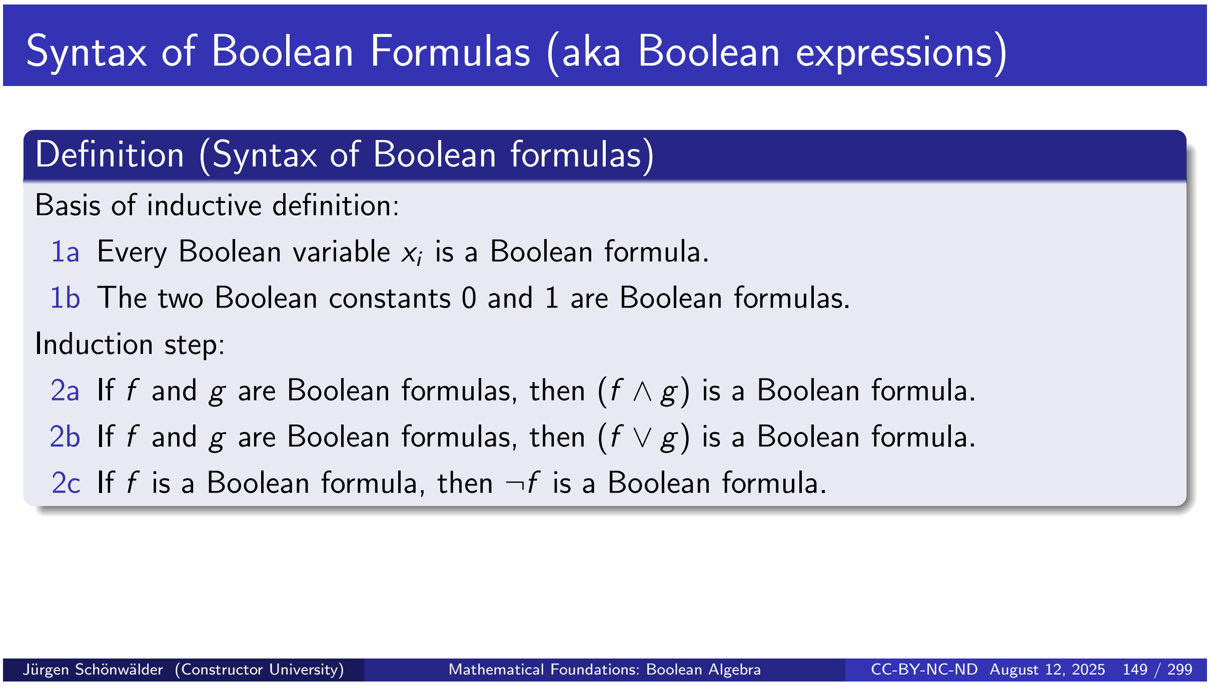
x	y	z	$(\neg(x \wedge y) \vee (z \wedge y))$
0	0	0	1
0	0	1	...
0	1	0	...
0	1	1	...
1	0	0	...
1	0	1	...
1	1	0	...
1	1	1	...

To fill in the first row (the first interpretation), one computes:

$$\begin{aligned} f(0, 0, 0) &= (\neg(0 \wedge 0) \vee (0 \wedge 0)) \\ &= (\neg 0 \vee 0) \\ &= (1 \vee 0) \\ &= 1 \end{aligned}$$

Note that boolean formulas can be considered boolean polynomials. This becomes perhaps more obvious if one uses the alternate “engineering” writing style where $x \cdot y$ is used instead of $x \wedge y$ and $x + y$ is used instead of $x \vee y$ and $\overline{x}$ is used instead of $\neg x$:

$$f(x, y, z) = \overline{(x \cdot y)} + (z \cdot y)$$



Syntax of Boolean Formulas (aka Boolean expressions)

Definition (Syntax of Boolean formulas)

Basis of inductive definition:

1a Every Boolean variable x_i is a Boolean formula.

1b The two Boolean constants 0 and 1 are Boolean formulas.

Induction step:

2a If f and g are Boolean formulas, then $(f \wedge g)$ is a Boolean formula.

2b If f and g are Boolean formulas, then $(f \vee g)$ is a Boolean formula.

2c If f is a Boolean formula, then $\neg f$ is a Boolean formula.

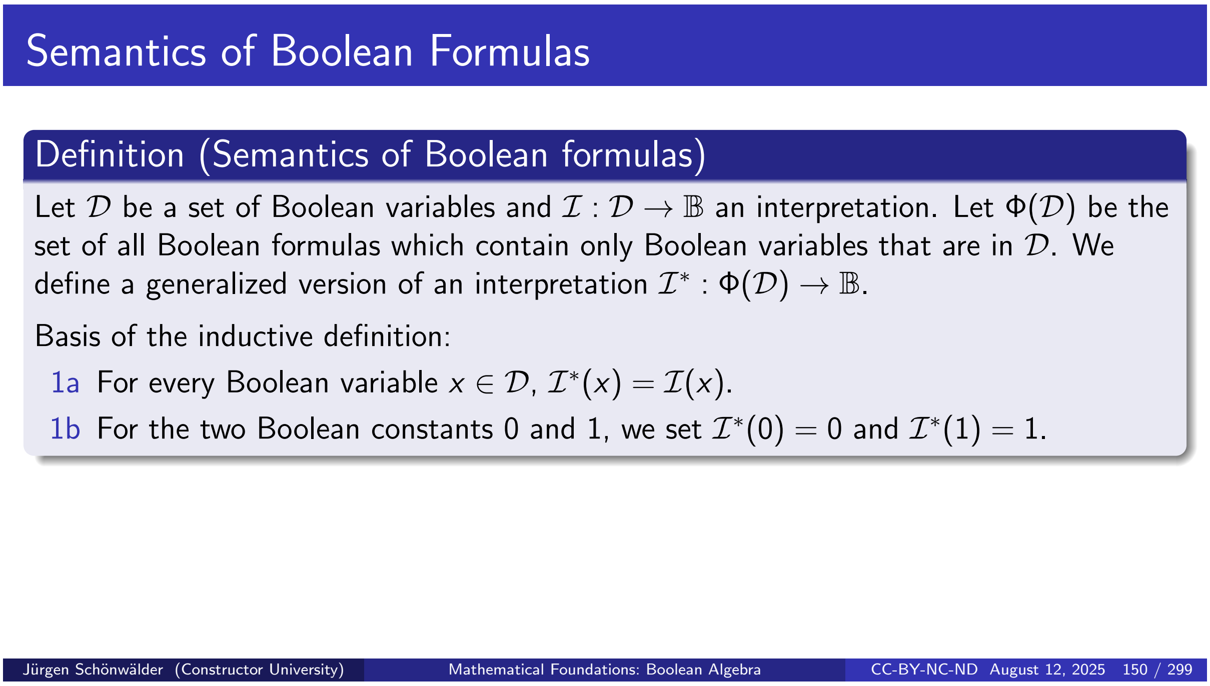
Strictly speaking, only x_i qualify for step 1a but in practice we may also use $x, y, \ldots$.

The definition provides all we need to verify whether a particular sequence of symbols qualifies as a Boolean formula. Obviously, $(\neg(x \wedge y) \vee (z \wedge y))$ is a well-formed Boolean formula while $\neg(x\wedge)$ is not.

In practice, we often use conventions that allow us to save parenthesis. For example, we may simply write $(x \wedge y \wedge z)$ instead of $((x \wedge y) \wedge z)$ or $(x \wedge (y \wedge z))$. Furthermore, we may write:

- $(x \rightarrow y)$ as a shorthand notation for $(\neg x \vee y)$
- $(x \leftrightarrow y)$ as a shorthand notation for $((x \wedge y) \vee (\neg x \wedge \neg y))$

Note that the definition for Boolean formulas defines the syntax of Boolean formulas. It provides a grammar that allows us to construct valid formulas and it allows us to decide whether a given formula is valid. However, the definition does not define what the formula means, that is, the semantics. We intuitively assume a certain semantic that does "make sense" but we have not yet defined the semantics formally.



Semantics of Boolean Formulas

Definition (Semantics of Boolean formulas)

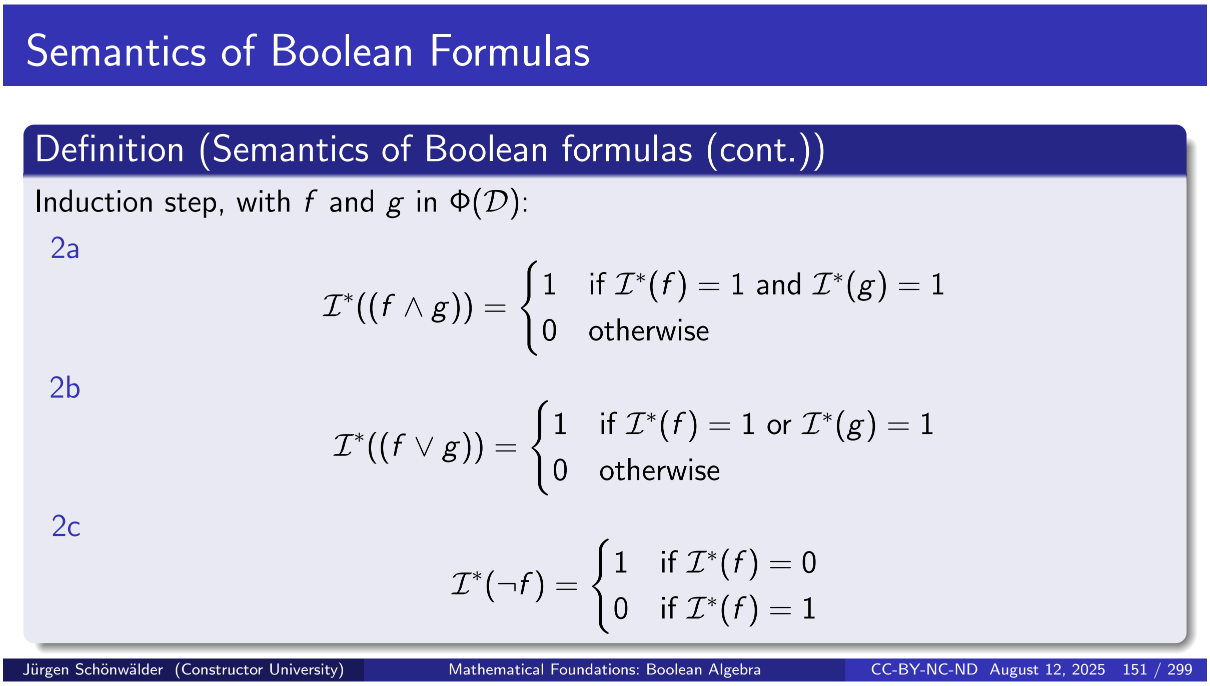
Let $\mathcal{D}$ be a set of Boolean variables and $\mathcal{I} : \mathcal{D} \to \mathbb{B}$ an interpretation. Let $\Phi(\mathcal{D})$ be the set of all Boolean formulas which contain only Boolean variables that are in $\mathcal{D}$. We define a generalized version of an interpretation $\mathcal{I}^* : \Phi(\mathcal{D}) \to \mathbb{B}$.

Basis of the inductive definition:

1a For every Boolean variable $x \in \mathcal{D}$, $\mathcal{I}^*(x) = \mathcal{I}(x)$.

1b For the two Boolean constants 0 and 1, we set $\mathcal{I}^*(0) = 0$ and $\mathcal{I}^*(1) = 1$.

Because this generalized interpretation $\mathcal{I}^*$ is the same as $\mathcal{I}$ for Boolean variables $x \in \mathcal{D}$, we say that $\mathcal{I}^*$ *extends* $\mathcal{I}$ from the domain $\mathcal{D}$ to the domain $\Phi(\mathcal{D})$. Following common practice, we will use $\mathcal{I}$ for the generalized interpretation too. Furthermore, since the set $\mathcal{D}$ is often clear from the context, we will often not specify it explicitly.



Semantics of Boolean Formulas

Definition (Semantics of Boolean formulas (cont.))

Induction step, with f and g in $\Phi(\mathcal{D})$:

2a
$$\mathcal{I}^*((f \land g)) = \begin{cases} 1 & \text{if } \mathcal{I}^*(f) = 1 \text{ and } \mathcal{I}^*(g) = 1 \\ 0 & \text{otherwise} \end{cases}$$

2b
$$\mathcal{I}^*((f \lor g)) = \begin{cases} 1 & \text{if } \mathcal{I}^*(f) = 1 \text{ or } \mathcal{I}^*(g) = 1 \\ 0 & \text{otherwise} \end{cases}$$

2c
$$\mathcal{I}^*(\neg f) = \begin{cases} 1 & \text{if } \mathcal{I}^*(f) = 0 \\ 0 & \text{if } \mathcal{I}^*(f) = 1 \end{cases}$$

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Boolean Algebra CC-BY-NC-ND August 12, 2025 151 / 299

Note that a boolean expression defines a boolean function and that multiple boolean expressions can define the same boolean function. This leads to questions such as:

- Are two boolean expressions defining the same boolean function?
- Given a boolean expression, can we find a “simpler” boolean expression defining the same boolean function?

Section 18: Boolean Algebra Equivalence Laws

16 Boolean Variables and Elementary Functions

17 Boolean Functions and Formulas

18 Boolean Algebra Equivalence Laws

19 Conjunctive and Disjunctive Normal Forms

20 Complexity of Boolean Formulas

Tautology and Contradiction

Definition (adapted interpretation)

An interpretation $\mathcal{I} : \mathcal{D} \to \mathbb{B}$ is *adapted* to a Boolean formula f if all Boolean variables that occur in f are contained in $\mathcal{D}$.

Definition (tautologies and contradictions)

A Boolean formula f is a *tautology* if for all interpretations $\mathcal{I}$, which are adapted to f, $\mathcal{I}(f) = 1$ holds. A Boolean formula f is a *contradiction* if for all interpretations $\mathcal{I}$, which are adapted to f, $\mathcal{I}(f) = 0$ holds.

A tautology is a Boolean formula which is always true, and a contradiction is never true. The classical example of a tautology is

$$(x \lor \neg x)$$

and the classical example of a contradiction is

$$(x \land \neg x).$$

Two elementary facts relating to tautologies and contradictions:

- For any Boolean formula f, $(f \lor \neg f)$ is a tautology and $(f \land \neg f)$ is a contradiction.
- If f is a tautology, then $\neg f$ is a contradiction and vice versa.

For a complex Boolean formula f with many Boolean variables, it is not easy to find out whether it is a tautology or a contradiction. One way to find out would be to compute the complete truth table. Recall that each row in a truth table corresponds to one possible interpretation of the variables in f. Recall furthermore that if f has k Boolean variables, the truth table has 2^k rows. This becomes quickly impractical if k grows large. Unfortunately, there is no known general procedure to find out whether a given f is a tautology or a contradiction, which is less costly than computing the entire truth table. In fact, logicians have reason to believe that no faster method exists (but this is an unproven conjecture!).

Satisfying a Boolean Formula

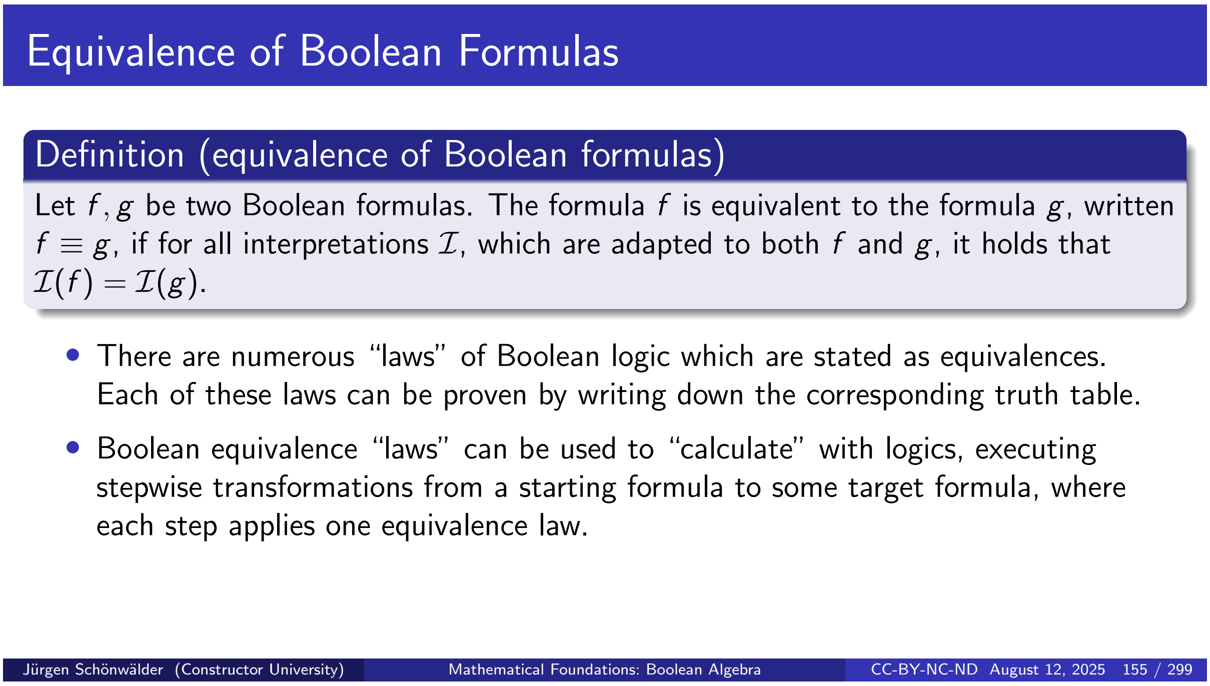
Definition (satisfying a Boolean formula)

An interpretation $\mathcal{I}$, which is adapted to a Boolean formula f, is said to *satisfy* the formula f if $\mathcal{I}(f) = 1$. A formula f is called *satisfiable* if there exists an interpretation which satisfies f.

The following two statements are equivalent characterizations of satisfiability:

- A Boolean formula is satisfiable if and only if its truth table contains at least one row that results in 1.
- A Boolean formula is satisfiable if and only if it is not a contradiction.

Note that the syntactic definition of Boolean formulas defines merely a set of legal sequences of symbols. Via the definition of the semantics of Boolean formulas, we obtain a Boolean function for every Boolean formula. In other words, there is a distinction between a Boolean formula and the Boolean function induced by the formula. In practice, this distinction is often not made and we often treat formulars as synomyms for the functions induced by the formula and we may use a function name to refer to the formula that was used to define the function.



Equivalence of Boolean Formulas

Definition (equivalence of Boolean formulas)

Let f, g be two Boolean formulas. The formula f is equivalent to the formula g, written $f \equiv g$, if for all interpretations $\mathcal{I}$, which are adapted to both f and g, it holds that $\mathcal{I}(f) = \mathcal{I}(g)$.

- There are numerous "laws" of Boolean logic which are stated as equivalences. Each of these laws can be proven by writing down the corresponding truth table.
- Boolean equivalence "laws" can be used to "calculate" with logics, executing stepwise transformations from a starting formula to some target formula, where each step applies one equivalence law.

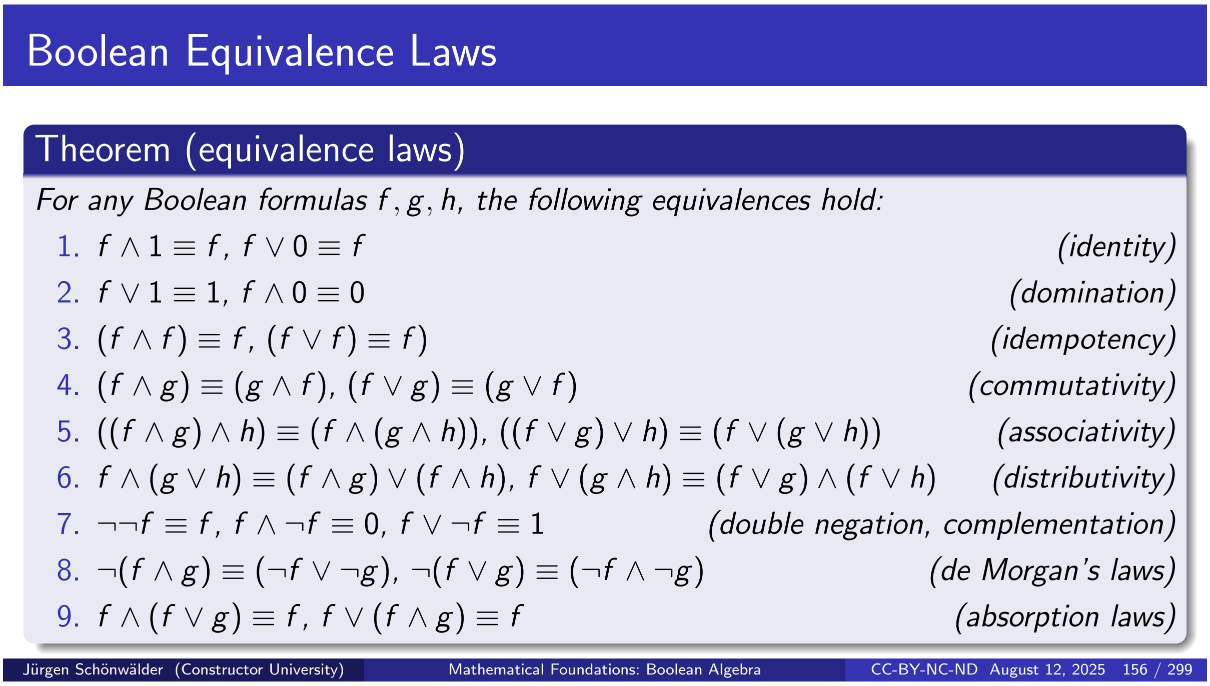
Jürgen Schönwälder (Constructor University) Mathematical Foundations: Boolean Algebra CC-BY-NC-ND August 12, 2025 155 / 299

A simple example is the following equivalence:

$$(x \lor y) \equiv (y \lor x)$$

Note that equivalent formulas are not required to have the same set of variables. The following equivalence surely holds:

$$(x \lor y) \equiv ((x \lor y) \land (z \lor \neg z))$$



Boolean Equivalence Laws

Theorem (equivalence laws)

For any Boolean formulas f, g, h, the following equivalences hold:

1. $f \wedge 1 \equiv f$, $f \vee 0 \equiv f$ *(identity)*
2. $f \vee 1 \equiv 1$, $f \wedge 0 \equiv 0$ *(domination)*
3. $(f \wedge f) \equiv f$, $(f \vee f) \equiv f)$ *(idempotency)*
4. $(f \wedge g) \equiv (g \wedge f)$, $(f \vee g) \equiv (g \vee f)$ *(commutativity)*
5. $((f \wedge g) \wedge h) \equiv (f \wedge (g \wedge h))$, $((f \vee g) \vee h) \equiv (f \vee (g \vee h))$ *(associativity)*
6. $f \wedge (g \vee h) \equiv (f \wedge g) \vee (f \wedge h)$, $f \vee (g \wedge h) \equiv (f \vee g) \wedge (f \vee h)$ *(distributivity)*
7. $\neg\neg f \equiv f$, $f \wedge \neg f \equiv 0$, $f \vee \neg f \equiv 1$ *(double negation, complementation)*
8. $\neg(f \wedge g) \equiv (\neg f \vee \neg g)$, $\neg(f \vee g) \equiv (\neg f \wedge \neg g)$ *(de Morgan's laws)*
9. $f \wedge (f \vee g) \equiv f$, $f \vee (f \wedge g) \equiv f$ *(absorption laws)*

Each of these equivalence laws can be proven by writing down the corresponding truth table. To illustrate this, here is the truth table for the first of the two de Morgan's laws:

f	g	$\neg f$	$\neg g$	$(f \wedge g)$	$\neg(f \wedge g)$	$(\neg f \vee \neg g)$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

Designing algorithms to transform starting formulas into target formulas is a practically important topic of artificial intelligence applications, more specifically "automated reasoning".

Try to simplify the following formula:

$$\begin{aligned} f(x,y,z) &= (((x \vee y) \wedge (\neg y \wedge z)) \wedge z) \\ &= \ldots \\ &= ((x \wedge \neg y) \wedge z) \end{aligned}$$

Section 19: Conjunctive and Disjunctive Normal Forms

16 Boolean Variables and Elementary Functions

17 Boolean Functions and Formulas

18 Boolean Algebra Equivalence Laws

19 Conjunctive and Disjunctive Normal Forms

20 Complexity of Boolean Formulas

Literals, Monomials, Clauses

Definition (literals)

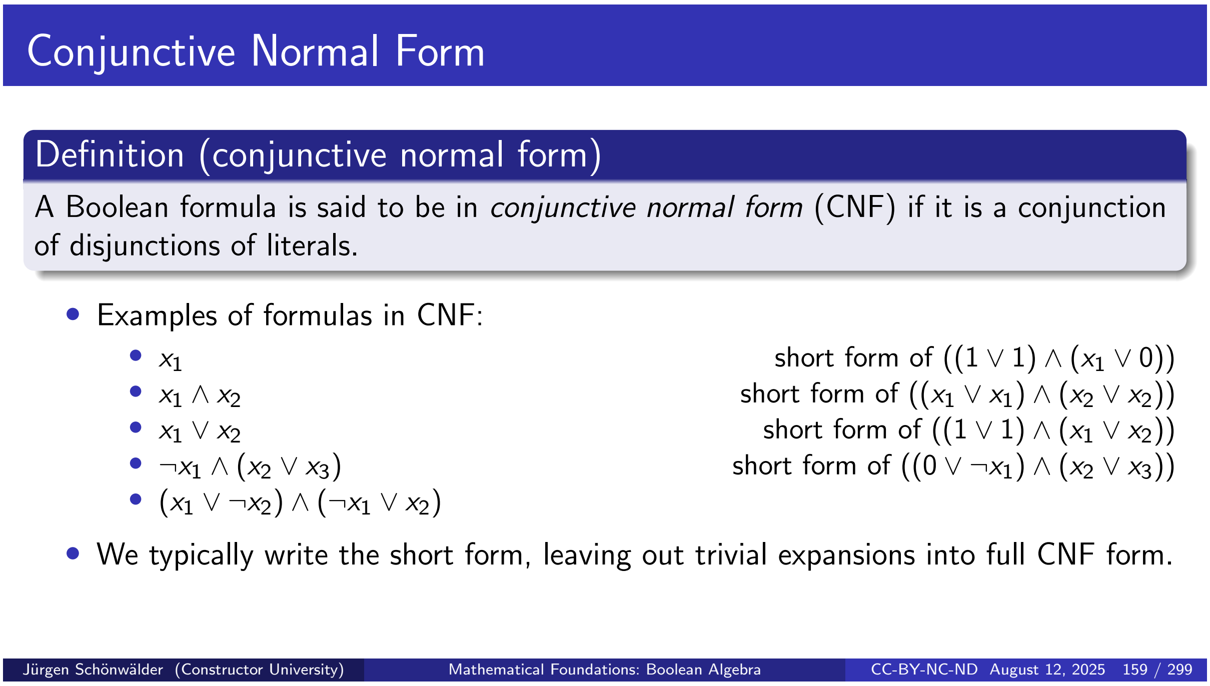
A *literal* L_i is a Boolean formula that has one of the forms x_i, $\neg x_i$, 0, 1, $\neg 0$, $\neg 1$, i.e., a literal is either a Boolean variable or a constant or a negation of a Boolean variable or a constant. The literals x_i, 0, 1 are called *positive literals* and the literals $\neg x_i$, $\neg 0$, $\neg 1$ are called *negative literals*.

Definition (monomial)

A *monomial* (or *product term*) is a literal or the conjunction (product) of literals.

Definition (clause)

A *clause* (or *sum term*) is a literal or the disjunction (sum) of literals.



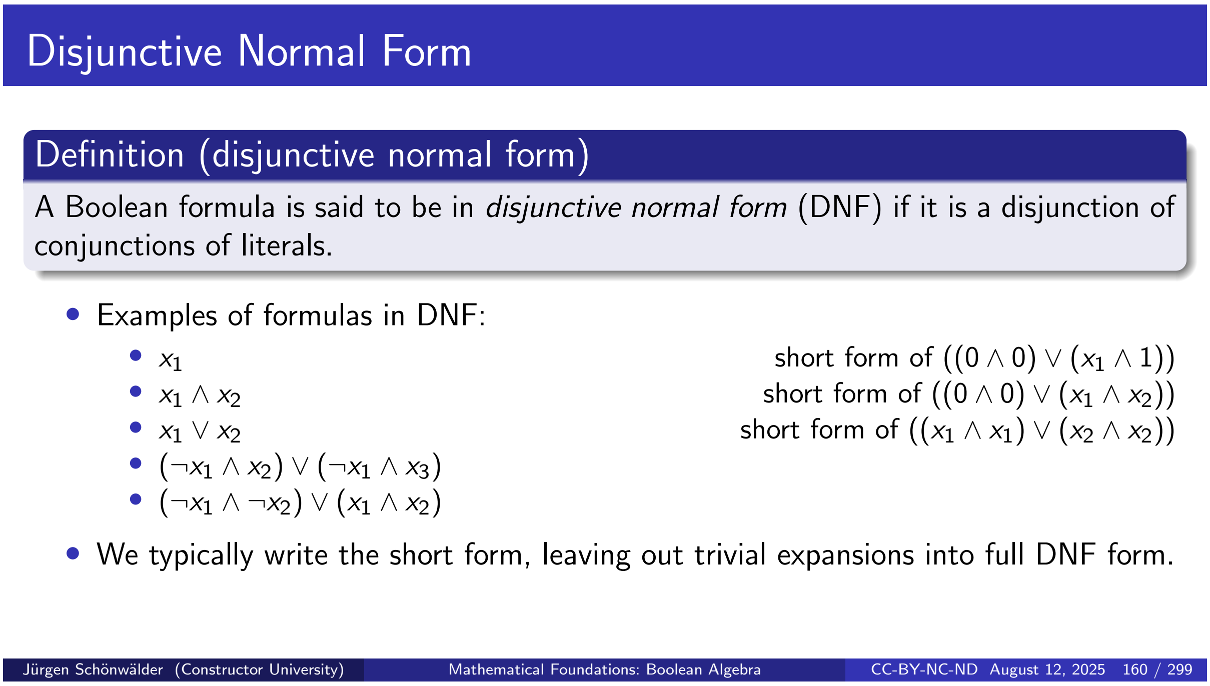
Conjunctive Normal Form

Definition (conjunctive normal form)

A Boolean formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals.

- Examples of formulas in CNF:
    - x_1 short form of $((1 \lor 1) \land (x_1 \lor 0))$
    - $x_1 \land x_2$ short form of $((x_1 \lor x_1) \land (x_2 \lor x_2))$
    - $x_1 \lor x_2$ short form of $((1 \lor 1) \land (x_1 \lor x_2))$
    - $\neg x_1 \land (x_2 \lor x_3)$ short form of $((0 \lor \neg x_1) \land (x_2 \lor x_3))$
    - $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2)$
- We typically write the short form, leaving out trivial expansions into full CNF form.

The terms of a conjunctive normal form (CNF) are all clauses. In other words, a conjunctive normal form is a product of sum terms.



Disjunctive Normal Form

Definition (disjunctive normal form)

A Boolean formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals.

- Examples of formulas in DNF:
  - x_1 short form of $((0 \land 0) \lor (x_1 \land 1))$
  - $x_1 \land x_2$ short form of $((0 \land 0) \lor (x_1 \land x_2))$
  - $x_1 \lor x_2$ short form of $((x_1 \land x_1) \lor (x_2 \land x_2))$
  - $(\neg x_1 \land x_2) \lor (\neg x_1 \land x_3)$
  - $(\neg x_1 \land \neg x_2) \lor (x_1 \land x_2)$
- We typically write the short form, leaving out trivial expansions into full DNF form.

The terms of a disjunctive normal form (DNF) are all monomials. In other words, a disjunctive normal form is a sum of product terms.

Equivalence of Normal Forms

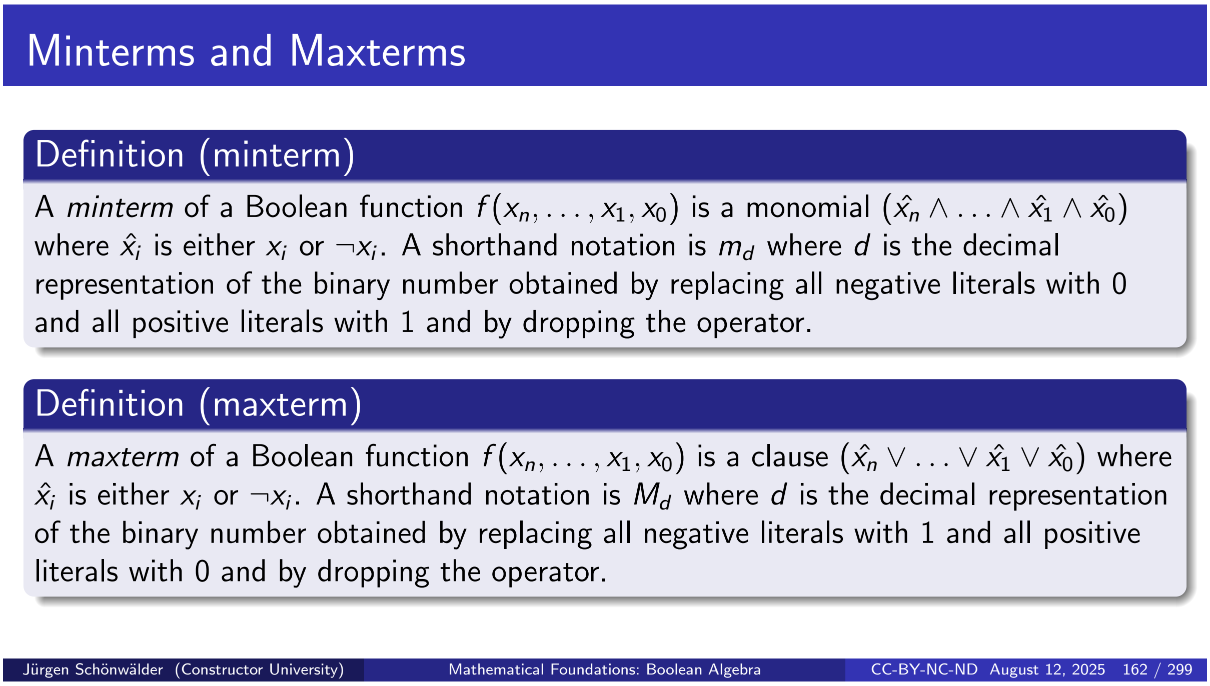
Proposition (CNF equivalence)

Every Boolean formula f is equivalent to a Boolean formula g in conjunctive normal form.

Proposition (DNF equivalence)

Every Boolean formula f is equivalent to a Boolean formula g in disjunctive normal form.

- These two results are important since we can represent any Boolean formula in a “shallow” format that does not need any “deeply nested” bracketing levels.



Minterms and Maxterms

Definition (minterm)

A *minterm* of a Boolean function $f(x_n, \ldots, x_1, x_0)$ is a monomial $(\hat{x}_n \wedge \ldots \wedge \hat{x}_1 \wedge \hat{x}_0)$ where $\hat{x}_i$ is either x_i or $\neg x_i$. A shorthand notation is m_d where d is the decimal representation of the binary number obtained by replacing all negative literals with 0 and all positive literals with 1 and by dropping the operator.

Definition (maxterm)

A *maxterm* of a Boolean function $f(x_n, \ldots, x_1, x_0)$ is a clause $(\hat{x}_n \vee \ldots \vee \hat{x}_1 \vee \hat{x}_0)$ where $\hat{x}_i$ is either x_i or $\neg x_i$. A shorthand notation is M_d where d is the decimal representation of the binary number obtained by replacing all negative literals with 1 and all positive literals with 0 and by dropping the operator.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Boolean Algebra CC-BY-NC-ND August 12, 2025 162 / 299

For example, the Boolean function

$$f(x, y, z) = (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z)$$

can be written as

$$f(x, y, z) = m_1 \vee m_2 \vee m_4 \vee m_5 \vee m_6$$

or with the alternative notation as

$$f(x, y, z) = m_1 + m_2 + m_4 + m_5 + m_6.$$

If we have the DNF, we easily get the CNF as well by selecting the maxterms for which there is no corresponding minterm. Continuing the example, we get

$$f(x, y, z) = M_0 \wedge M_3 \wedge M_7$$

or the alternative notation

$$f(x, y, z) = M_0 \cdot M_3 \cdot M_7.$$

The CNF is therefore given by:

$$f(x, y, z) = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg z \vee \neg y \vee \neg z)$$

Obtaining a DNF from a Truth Table

- Given a truth table, a DNF can be obtained by writing down a conjunction of the input values for every row where the result is 1 and connecting all obtained conjunctions together with a disjunction.

x	y	$x \veebar y$
0	0	0
0	1	1
1	0	1
1	1	0

- 2nd row: $\neg x \wedge y$
- 3rd row: $x \wedge \neg y$
- $x \veebar y = (\neg x \wedge y) \vee (x \wedge \neg y) = m_1 + m_2$

Every boolean function defined by a boolean expression can be represented as a truth table. Since it is possible to obtain the DNF directly from the truth table, every boolean expression can be represented in DNF.

We already know from the definition of $\veebar$ that $x \veebar y = (\neg x \wedge y) \vee (x \wedge \neg y)$.

Obtaining a CNF from a Truth Table

- Given a truth table, a CNF can be obtained by writing down a disjunction of the negated input values for every row where the result is 0 and connecting all obtained disjunctions together with a conjunction.

x	y	$x \veebar y$
0	0	0
0	1	1
1	0	1
1	1	0

- 1st row: $x \vee y$
- 4th row: $\neg x \vee \neg y$
- $x \veebar y = (x \vee y) \wedge (\neg x \vee \neg y) = M_0 \cdot M_3$

Every boolean function defined by a boolean expression can be represented as a truth table. Since it is possible to obtain the CNF directly from the truth table, every boolean expression can be represented in CNF.

We show that $(x \vee y) \wedge (\neg x \vee \neg y)$ is indeed the same as $x \veebar y$:

$$\begin{aligned}
(x \vee y) \wedge (\neg x \vee \neg y) &= ((x \vee y) \wedge \neg x) \vee ((x \vee y) \wedge \neg y) && \text{(distributivity)}\\
&= (x \wedge \neg x) \vee (y \wedge \neg x) \vee (x \wedge \neg y) \vee (y \wedge \neg y) && \text{(distributivity)}\\
&= 0 \vee (y \wedge \neg x) \vee (x \wedge \neg y) \vee 0 && \text{(complementation)}\\
&= (\neg x \wedge y) \vee (x \wedge \neg y) && \text{(identity)}\\
&= x \veebar y
\end{aligned}$$

Section 20: Complexity of Boolean Formulas

16 Boolean Variables and Elementary Functions

17 Boolean Functions and Formulas

18 Boolean Algebra Equivalence Laws

19 Conjunctive and Disjunctive Normal Forms

20 Complexity of Boolean Formulas

Cost of Boolean Formulas and Functions

Definition (cost of boolean formula)

The cost $C(f)$ of a boolean formula f is the number of operators in f.

Definition (cost of boolean function)

The cost $C(f)$ of a boolean function f is the minimum cost of boolean formulas defining f:

$$C(f) = \min_{g \text{ defines } f} C(g)$$

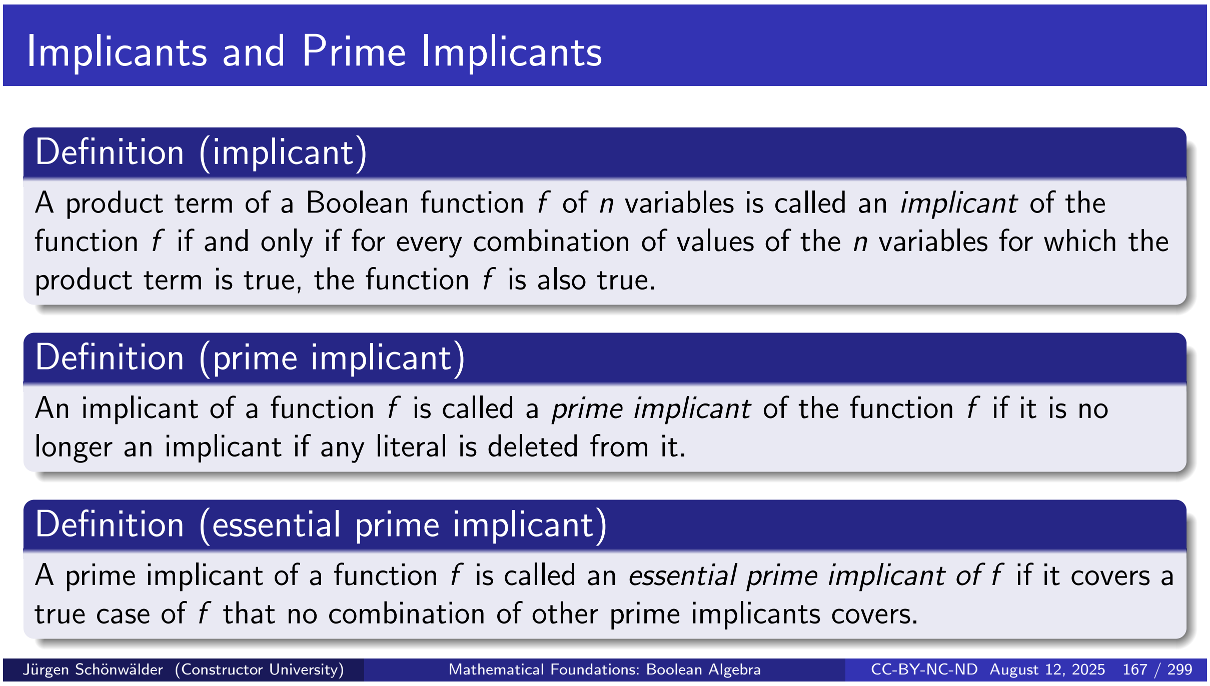
- We can find formulas of arbitrary high cost for a given boolean function.
- How do we find a formula with minimal cost for a given boolean function?

When talking about the cost of Boolean formulas, we often restrict us to a certain set of operations, e.g., the classic set $\{\wedge, \vee, \neg\}$. In some contexts, $\neg$ is not counted and only the number of $\wedge$ and $\vee$ operations is counted. (The reasoning is that negation is cheap compared to the other operations and hence negation can be applied to any input or output easily.) We will follow this approach and restrict us to the classic $\{\wedge, \vee, \neg\}$ operations and only count the number of $\wedge$ and $\vee$ operations unless stated otherwise.

With this, the cost of the Boolean formula $((x \vee y) \wedge (\neg y \wedge z)) \wedge z$ is $C(((x \vee y) \wedge (\neg y \wedge z)) \wedge z) = 4$ and the cost of the Boolean formula $x \wedge \neg y \wedge z$ is $C(x \wedge \neg y \wedge z) = 2$. Since

$$\begin{aligned} f(x, y, z) &= ((x \vee y) \wedge (\neg y \wedge z)) \wedge z \\ &= x \wedge \neg y \wedge z \end{aligned}$$

and there is no way to further minimize $x \wedge \neg y \wedge z$, the cost of the Boolean function f is $C(f) = 2$.



Implicants and Prime Implicants

Definition (implicant)

A product term of a Boolean function f of n variables is called an *implicant* of the function f if and only if for every combination of values of the n variables for which the product term is true, the function f is also true.

Definition (prime implicant)

An implicant of a function f is called a *prime implicant* of the function f if it is no longer an implicant if any literal is deleted from it.

Definition (essential prime implicant)

A prime implicant of a function f is called an *essential prime implicant of f* if it covers a true case of f that no combination of other prime implicants covers.

Observations:

- If an expression defining f is in DNF, then every minterm of the DNF is an implicant of f.
- Any term formed by combining two or more minterms of a DNF is an implicant.
- Each prime implicant of a function has a minimum number of literals; no more literals can be eliminated from it.

Example:

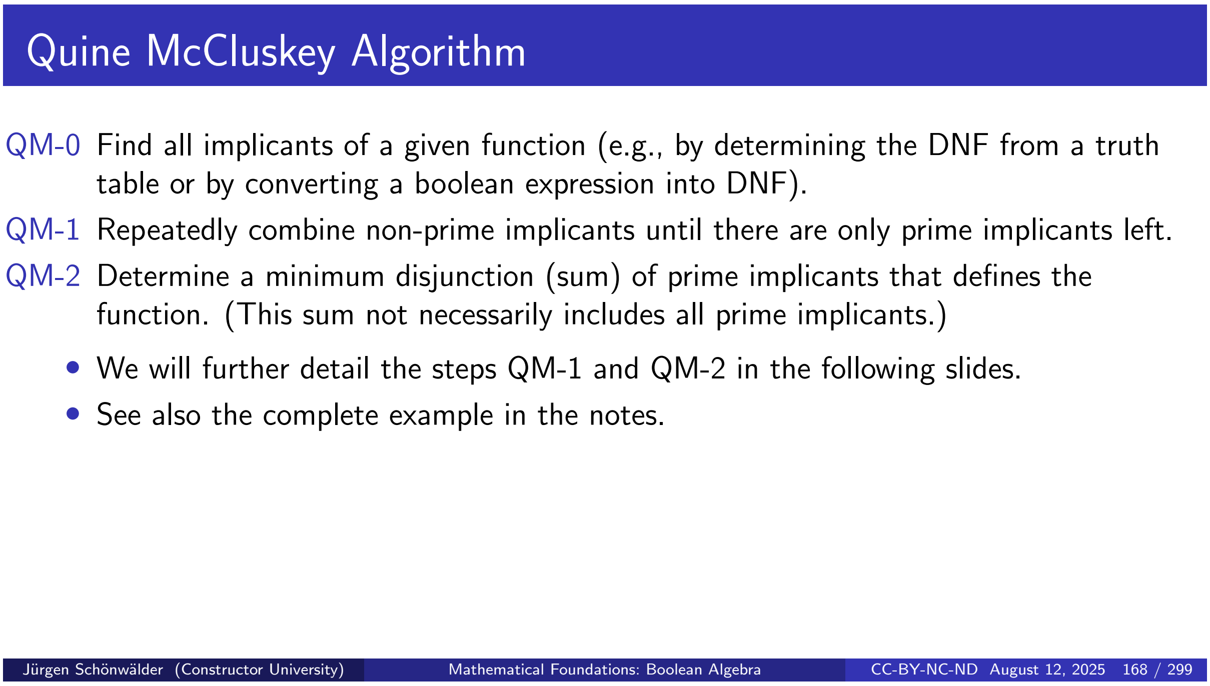
$$\begin{aligned} f(x,y,z) &= (\neg x \land \neg y \land \neg z) \lor (x \land \neg y \land \neg z) \lor (x \land \neg y \land z) \lor (x \land y \land z) \\ &= (\neg y \land \neg z) \lor (x \land z) \end{aligned}$$

- Implicant $(\neg x \land \neg y \land \neg z)$ is not a prime implicant. The first two product terms can be combined since they only differ in one variable:

  $$\begin{aligned} (\neg x \land \neg y \land \neg z) \lor (x \land \neg y \land \neg z) &= (\neg x \land x) \lor (\neg y \land \neg z) \\ &= 0 \lor (\neg y \land \neg z) \\ &= (\neg y \land \neg z) \end{aligned}$$

  The resulting product term $\neg y \land \neg z$ is still an implicant of f. In a similar way, y can be eliminated from the last two product terms.

- $(\neg y \land \neg z)$ and $(x \land z)$ are prime implicants (it is not possible to further eliminate a variable).



Quine McCluskey Algorithm

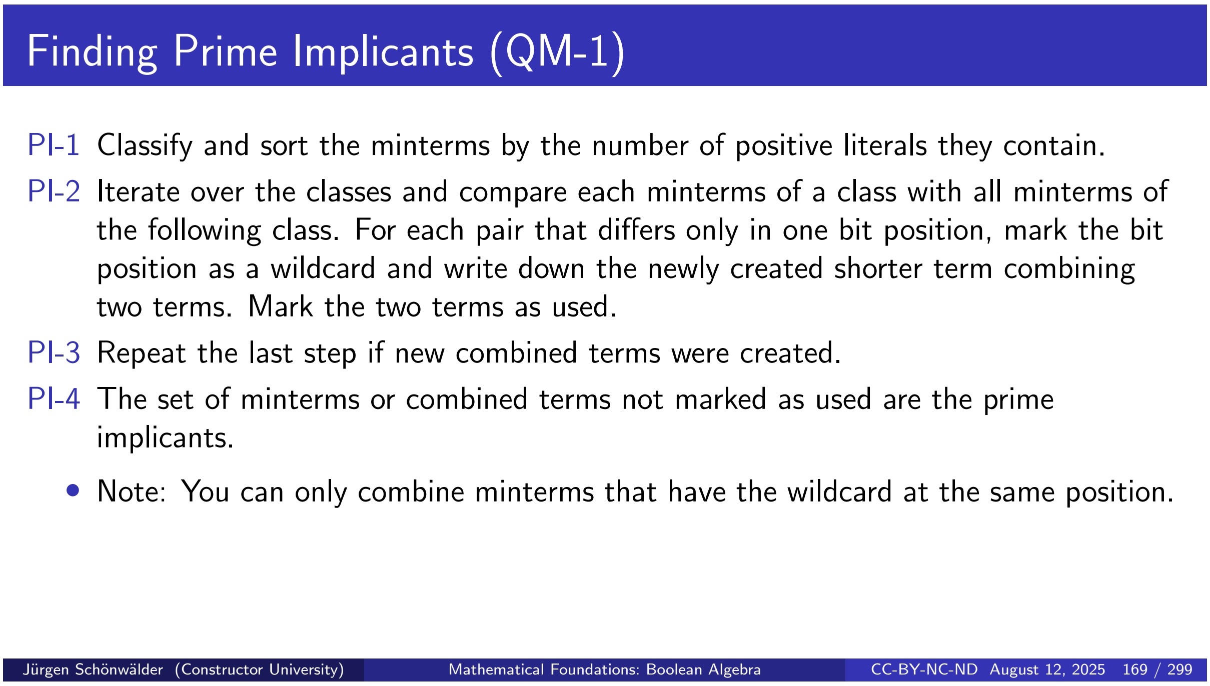
QM-0 Find all implicants of a given function (e.g., by determining the DNF from a truth table or by converting a boolean expression into DNF).

QM-1 Repeatedly combine non-prime implicants until there are only prime implicants left.

QM-2 Determine a minimum disjunction (sum) of prime implicants that defines the function. (This sum not necessarily includes all prime implicants.)

- We will further detail the steps QM-1 and QM-2 in the following slides.
- See also the complete example in the notes.

- The time complexity of the algorithm grows exponentially with the number of variables.
- The problem is known to be NP-hard (non-deterministic polynomial time hard). There is little hope that polynomial time algorithms exist for NP-hard problems.
- For large numbers of variables, it is necessary to use heuristics that run faster but which may not always find a minimal solution.



Finding Prime Implicants (QM-1)

PI-1 Classify and sort the minterms by the number of positive literals they contain.

PI-2 Iterate over the classes and compare each minterms of a class with all minterms of the following class. For each pair that differs only in one bit position, mark the bit position as a wildcard and write down the newly created shorter term combining two terms. Mark the two terms as used.

PI-3 Repeat the last step if new combined terms were created.

PI-4 The set of minterms or combined terms not marked as used are the prime implicants.

- Note: You can only combine minterms that have the wildcard at the same position.

Example: Minimize $f(a,b,c,d) = m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{14} + m_{15}$

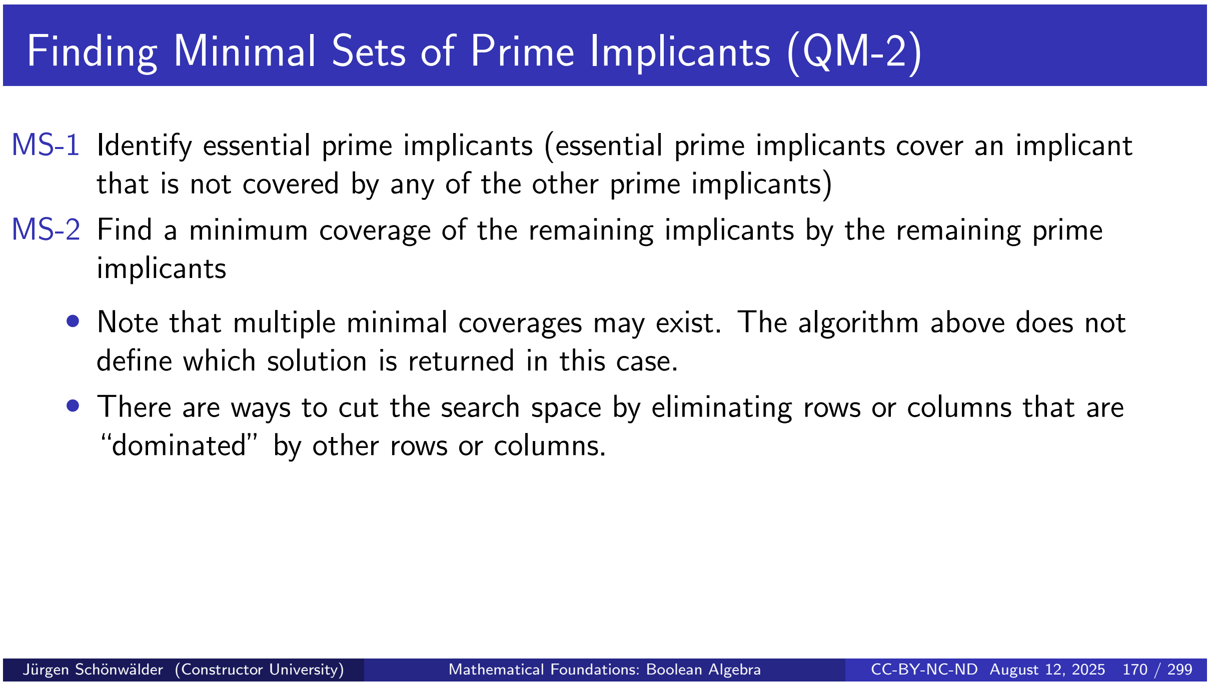
- Classify and sort minterms

minterm	pattern	used
m_4	0100	
m_8	1000	
m_9	1001	
m_{10}	1010	
m_{12}	1100	
m_{11}	1011	
m_{14}	1110	
m_{15}	1111	

- Combination steps

minterm	pattern	used	minterms	pattern	used	minterms	pattern	used
m_4	0100	✓	$m_{4,12}$	-100				
m_8	1000	✓	$m_{8,9}$	100-	✓	$m_{8,9,10,11}$	10--	
			$m_{8,10}$	10-0	✓	$m_{8,10,12,14}$	1--0	
			$m_{8,12}$	1-00	✓			
m_9	1001	✓	$m_{9,11}$	10-1	✓			
m_{10}	1010	✓	$m_{10,11}$	101-	✓	$m_{10,11,14,15}$	1-1-	
			$m_{10,14}$	1-10	✓			
m_{12}	1100	✓	$m_{12,14}$	11-0	✓			
m_{11}	1011	✓	$m_{11,15}$	1-11	✓			
m_{14}	1110	✓	$m_{14,15}$	111-	✓			
m_{15}	1111	✓						

- This gives us four prime implicants:
  - $m_{4,12} = (b \land \neg c \land \neg d)$
  - $m_{8,9,10,11} = (a \land \neg b)$
  - $m_{8,10,12,14} = (a \land \neg d)$
  - $m_{10,11,14,15} = (a \land c)$



Finding Minimal Sets of Prime Implicants (QM-2)

MS-1 Identify essential prime implicants (essential prime implicants cover an implicant that is not covered by any of the other prime implicants)

MS-2 Find a minimum coverage of the remaining implicants by the remaining prime implicants

- Note that multiple minimal coverages may exist. The algorithm above does not define which solution is returned in this case.
- There are ways to cut the search space by eliminating rows or columns that are “dominated” by other rows or columns.

We continue the example from the previous page. To find prime implicant sets, we construct a prime implicant table. The colummns are the original minterms and the rows represent the prime implicants. The marked cells in the table indicate whether a prime implicant covers a minterm. (● = essential, $*$ = covered, $\circ$ = uncovered)

	m_4	m_8	m_9	m_{10}	m_{11}	m_{12}	m_{14}	m_{15}
$m_{4,12}$	●					$*$		
$m_{8,9,10,11}$		$*$	●	$*$	$*$			
$m_{8,10,12,14}$		$*$		$*$		$*$	$*$	
$m_{10,11,14,15}$				$*$	$*$		$*$	●

Columns that only have a single marked cell indicate essential prime implicants. In this case, m_4 is only marked by $m_{4,12}$ and hence $m_{4,12}$ is an essential prime implicant. Similarly, m_9 is only marked by $m_{8,9,10,11}$, hence $m_{8,9,10,11}$ is an essential prime implicant. Finally, m_{15} is only marked by $m_{10,11,14,15}$, hence $m_{10,11,14,15}$ is an essential prime implicant as well.

The remaining prime implicant $m_{8,10,12,14}$ has marks only in columns that are covered already by prime implicants that we have already selected and hence $m_{8,10,12,14}$ is not needed in a minimal set of prime implicants.

The resulting minimal expression is $f'(a,b,c,d) = (b \land \neg c \land \neg d) \lor (a \land \neg b) \lor (a \land c)$. The minimal expression f' uses 6 operations (out of $\{\land, \lor\}$). The original expression used $8 \cdot 3 + 7 = 31$ operations (out of $\{\land, \lor\}$).

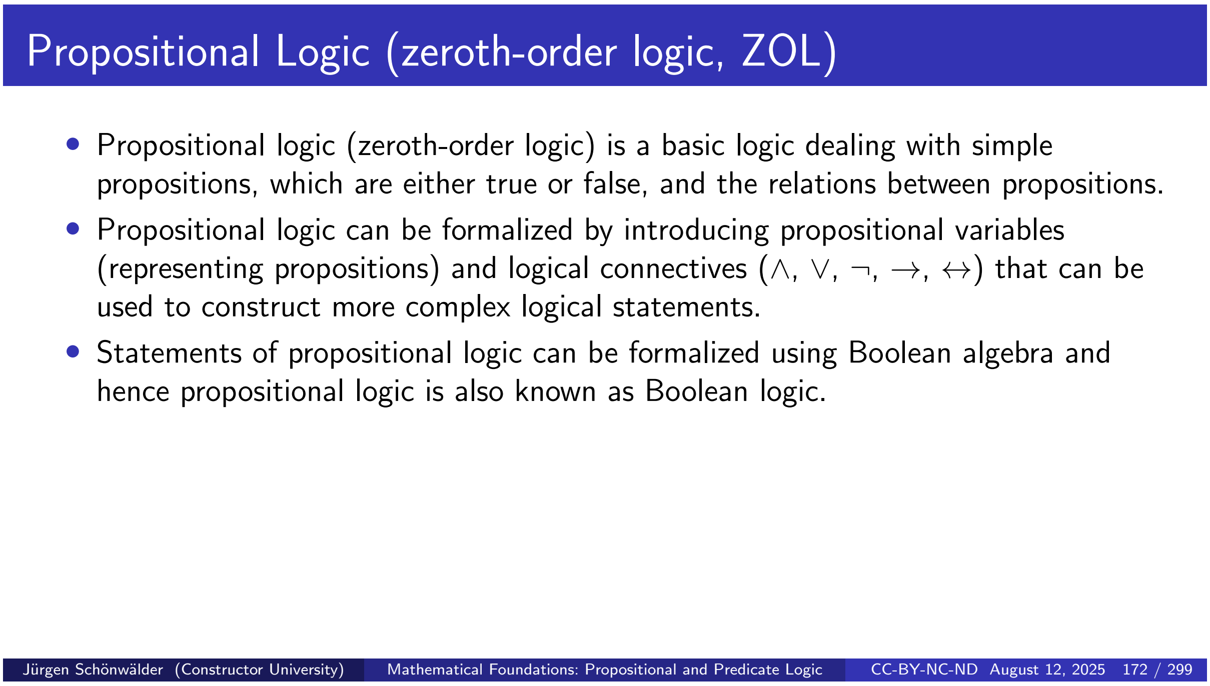
Part V

Propositional and Predicate Logic

This part provides an introduction to propositional and predicate logic with a focus on the formalization of logical statements.

By the end of this part, students should be able to

- summarize the difference between propositional logic, predicate logic and higher-order logic;
- outline how implications in propositional logic can be represented in conjunctive normal form;
- recall the satisfiability problem and its importance;
- explain the difference between the syntax and the semantics of expressions in predicate logic;
- recall the definition of free variables in expressions of predicate logic;
- translate English language statements into expressions of predicate logic;
- illustrate the domain of discourse of predicate logic;
- rephrase the definition of an interpretation of an expressions in predicate logic;
- summarize the notion of entailment and validity;
- outline concepts such as soundness and completeness.



Propositional Logic (zeroth-order logic, ZOL)

- Propositional logic (zeroth-order logic) is a basic logic dealing with simple propositions, which are either true or false, and the relations between propositions.
- Propositional logic can be formalized by introducing propositional variables (representing propositions) and logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$) that can be used to construct more complex logical statements.
- Statements of propositional logic can be formalized using Boolean algebra and hence propositional logic is also known as Boolean logic.

Propositional logic can be used to solve simple logic puzzles. Here is an example:

> I like Pat or I like Joe.
> If I like Pat, then I like Joe.
> Do I like Joe?

A common challenge is to translate English language (or a computer program, as we will see later), into a formal logic statement. The statement above has two propositions:

$P =$ ”I like Pat”

$J =$ ”I like Joe”

We know from the first sentence that P or J is true. We write this formally as $P \vee J$. We also know that the truth of J implies the truth of P. We write this formally as $P \rightarrow J$. Hence, we obtain the following formalization: $(P \vee J) \wedge (P \rightarrow J)$. Since this is a formula in Boolean algebra, we can simplify it as follows:

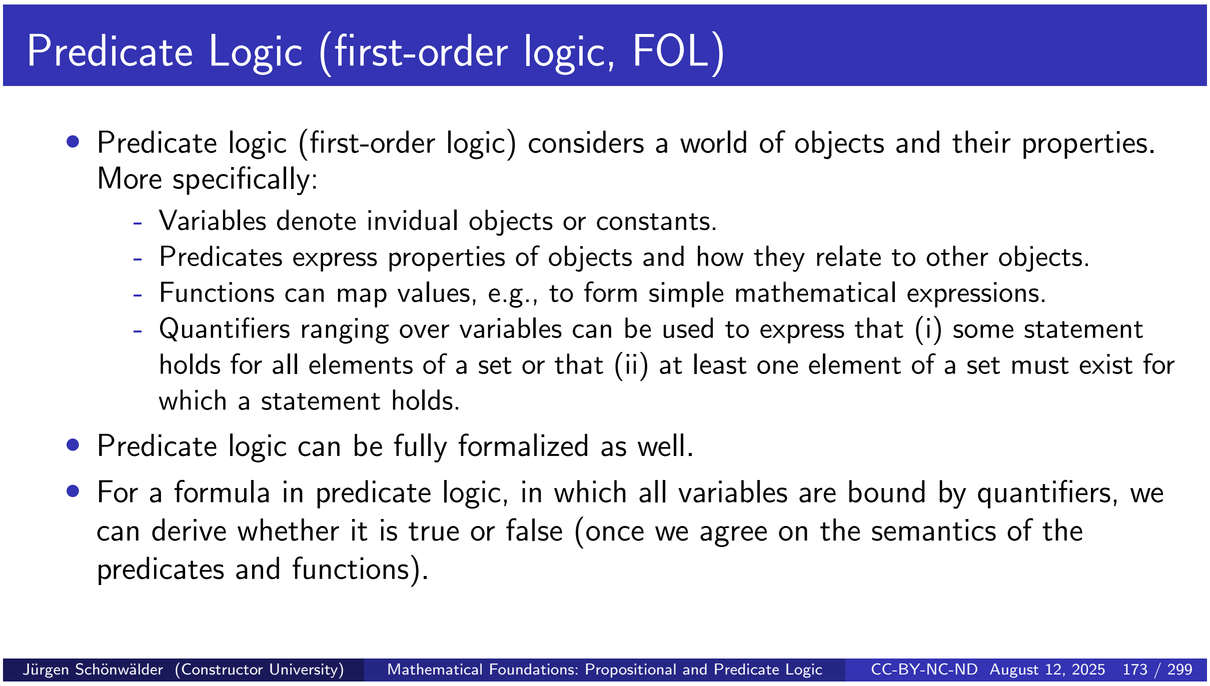
$$\begin{aligned}(P \vee J) \wedge (P \rightarrow J) &= (P \vee J) \wedge (\neg P \vee J)\\ &= (P \wedge \neg P) \vee J\\ &= 0 \vee J\\ &= J\end{aligned}$$

The first sentence is not entirely clear, it could also mean that I like either Pat or Joe. In this case, the formalization would be: $(P \veebar J) \wedge (P \rightarrow J)$. Simplifying this formula yields:

$$\begin{aligned}(P \veebar J) \wedge (P \rightarrow J) &= ((P \wedge \neg J) \vee (\neg P \wedge J)) \wedge (\neg P \vee J)\\ &= ((P \wedge \neg J) \wedge (\neg P \vee J)) \vee ((\neg P \wedge J) \wedge (\neg P \vee J))\\ &= ((P \wedge \neg J) \wedge \neg(P \wedge \neg J)) \vee (((\neg P \wedge J) \wedge \neg P) \vee ((\neg P \wedge J) \wedge J)))\\ &= ((P \wedge \neg J) \wedge \neg(P \wedge \neg J)) \vee ((\neg P \wedge J) \vee (\neg P \wedge J))\\ &= 0 \vee (\neg P \wedge J)\\ &= \neg P \wedge J\end{aligned}$$

A common source of confusion are implications. We already know from Boolean algebra that if the condition of an implication is false, then it does not matter what is implied, the implication is always true. Hence, a statement like “if cats are dogs, then the sun shines.” is true.

Propositional logic is limited to propositions that are either true or false, i.e., it does not allow us to make statements at a more fine-grained level of detail.



Predicate Logic (first-order logic, FOL)

- Predicate logic (first-order logic) considers a world of objects and their properties. More specifically:
  - Variables denote invidual objects or constants.
  - Predicates express properties of objects and how they relate to other objects.
  - Functions can map values, e.g., to form simple mathematical expressions.
  - Quantifiers ranging over variables can be used to express that (i) some statement holds for all elements of a set or that (ii) at least one element of a set must exist for which a statement holds.
- Predicate logic can be fully formalized as well.
- For a formula in predicate logic, in which all variables are bound by quantifiers, we can derive whether it is true or false (once we agree on the semantics of the predicates and functions).

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Propositional and Predicate Logic CC-BY-NC-ND August 12, 2025 173 / 299

We can translate the previous example in predicate logic by introducing the predicate like/2 and introducing constants (names) referring to specific objects (persons):

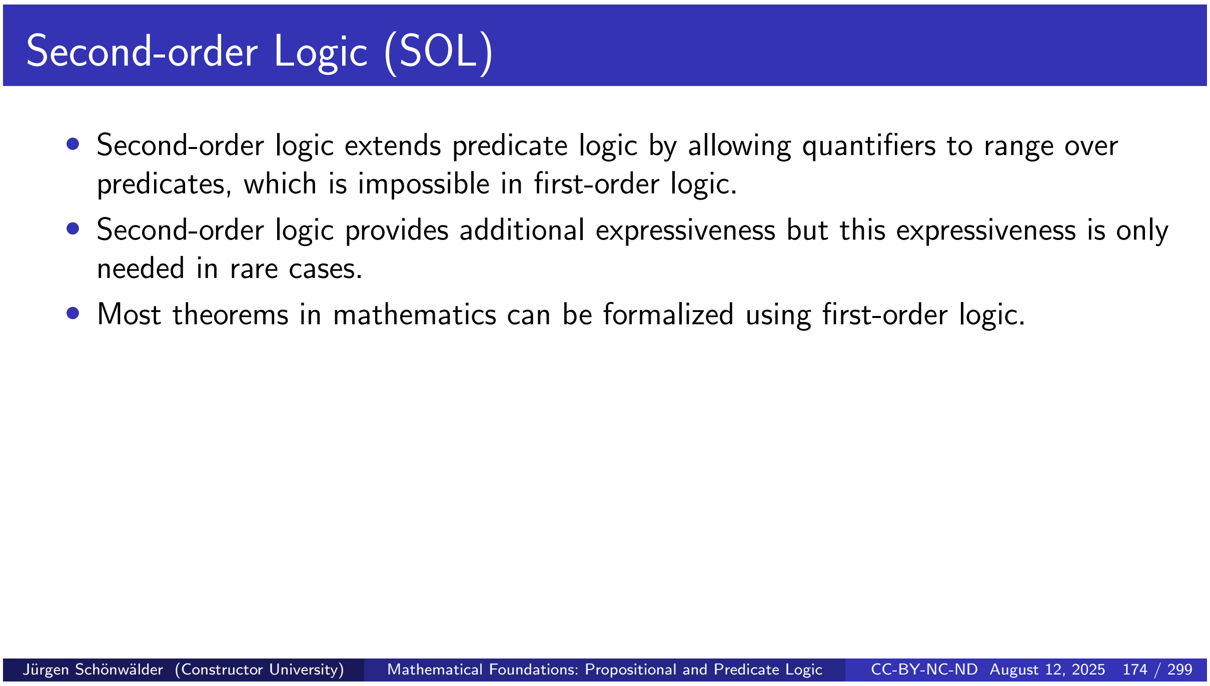
$$(\mathit{like}\,\mathit{Me}\,\mathit{Pat} \lor \mathit{like}\,\mathit{Me}\,\mathit{Joe}) \land (\mathit{like}\,\mathit{Me}\,\mathit{Pat} \rightarrow \mathit{like}\,\mathit{Me}\,\mathit{Joe})$$

To understand the power and limits of first-order logic, lets look at the Peano axioms defining natural numbers. Here are the first four of the five axioms (the predicate elem/2 is true if the first argument is an element of the set given by the second argument):

$$\begin{array}{l}
\mathit{elem}\,0\,\mathbb{N} \\
\forall n\,\mathit{elem}\,n\,\mathbb{N} \rightarrow \mathit{elem}\,s(n)\,\mathbb{N} \land \neg(n = s(n)) \\
\neg\exists n\,\mathit{elem}\,n\,\mathbb{N} \land 0 = s(n) \\
\forall n\,\forall m\,(\mathit{elem}\,n\,\mathbb{N} \land \mathit{elem}\,m\,\mathbb{N} \land s(n) = s(m)) \rightarrow n = m
\end{array}$$

Note that we cannot express the fifth Peano axiom in first-order logic since it quantifies over predicates. This is one of the rare cases where first-order logic is not expressive enough. (While there are work arounds, they do come with some subtle side effects.)

While first-order logic formulas are a bit more complex than zeroth-order logic formulas, they feel simple enough to work with. However, while it is relatively easy to read out a first-order logic formula in English, turning an English sentence into an equivalent first-order logic formula often is not that easy and a common source of errors. And if such a formalization error occurs, then the entire effort to formally prove a statement in first-order logic may be wasted.



Second-order Logic (SOL)

- Second-order logic extends predicate logic by allowing quantifiers to range over predicates, which is impossible in first-order logic.
- Second-order logic provides additional expressiveness but this expressiveness is only needed in rare cases.
- Most theorems in mathematics can be formalized using first-order logic.

We will not touch on second-order logic. We leave this to specialized courses on higher-order and alternate logics.

Non-standard Logics

- Temporal logics capture the notion of time. They can be used to reason about the temporal relationship of events, e.g. that a person is hungry until she eats something.
- Many-valued logics overcome the notion that a statement is either true or false by allowing additional truth values. For example, a three-valued logic could distinguish true, false, and unknown.
- Fuzzy logics represent the truth of a statement by a real number between 0 and 1. Fuzzy logic is based on the notion of fuzzy sets where set membership is described by a membership function returning a fuzzy value between 0 and 1.
- ...

Section 21: Propositional Logic

Logic Statements

- A common task is to decide whether statements of the following form are true:
  if *premises* P_1 **and** ... **and** P_m *hold,* **then** *conclusion* C *holds*
- The premises P_i and the conclusion C are expressed in some logic formalism, the simplest is Boolean logic (also called propositional logic).
- Restricting us to Boolean logic here, the statement above can be seen as a Boolean formula of the following structure

  $$(\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi$$

  and we are interested to find out whether such a formula is true, i.e., whether it is a tautology.

Tautology and Satisfiability

- Recall that a Boolean formula τ is a tautology if and only if $\tau' = \neg\tau$ is a contradiction. Furthermore, a Boolean formula is a contradiction if and only if it is not satisfiable. Hence, in order to check whether

  $$\tau = (\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi$$

  is a tautology, we may check whether

  $$\tau' = \neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi)$$

  is unsatisfiable.
- If we show that τ' is satisfiable, we have disproven τ.

Tautology and Satisfiability

- Since $\varphi \rightarrow \psi \equiv \neg(\varphi \land \neg\psi)$, we can rewrite the formulas as follows:

$$\tau = (\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi = \neg(\varphi_1 \land \ldots \land \varphi_m \land \neg\psi)$$

$$\tau' = \neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi) = (\varphi_1 \land \ldots \land \varphi_m \land \neg\psi)$$

- To disprove τ, it is often easier to prove that τ' is satisfiable.
- Note that τ' has a homogenous structure. If we transform the elements $\varphi_1, \ldots, \varphi_m, \psi$ into CNF, then the entire formula is in CNF.
- If τ' is in CNF, all we need to do is to invoke an algorithm that searches for interpretations $\mathcal{I}$ which satisfy a formula in CNF. If there is such an interpretation, τ is disproven, otherwise, if there is no such interpretation, then τ is proven.

Satisfiability Problem

Definition (satisfiability problem)

The satisfiability problem (SAT) is the following computational problem: Given as input a Boolean formula in CNF, compute as output a “yes” or “no” response according to whether the input formula is satisfiable or not.

- It is believed that there is no polynomial time solution for this problem.

There is no known general algorithm that efficiently (means in polynomial time) solves the SAT problem, and it is generally believed that no such algorithm exists. However, this belief has not been proven mathematically.

Resolving the question whether SAT has a polynomial-time algorithm is equivalent to answering the P versus NP problem, which is a famous open problem in the theory of computer science: The complexity class P contains all problems that are solvable in polynomial time by a deterministic machine and the complexity class NP contains all problems that are solvable in polynomial time by a non-deterministic machine (i.e., a machine that guesses the next best step). Obviously, P is a subset of NP. The big question is whether P is equivalent to NP. (Please remember that NP stands for non-deterministic polynomial time, it does not stand for non-polynomial time.)

Note that it is rather trivial to check whether a Boolean formula in DNF is satisfiable since it is sufficient to show that one of the conjunctions is satisfiable (since every conjunction is an implicant). A conjunction is satisfiable if it does not contain x and $\neg x$ for some variable x. Given an arbitrary Boolean formula, the conversion into DNF may unfortunately require exponential time.

By solving the general SAT problem, you will become a famous mathematician and you can secure a one million dollar price.

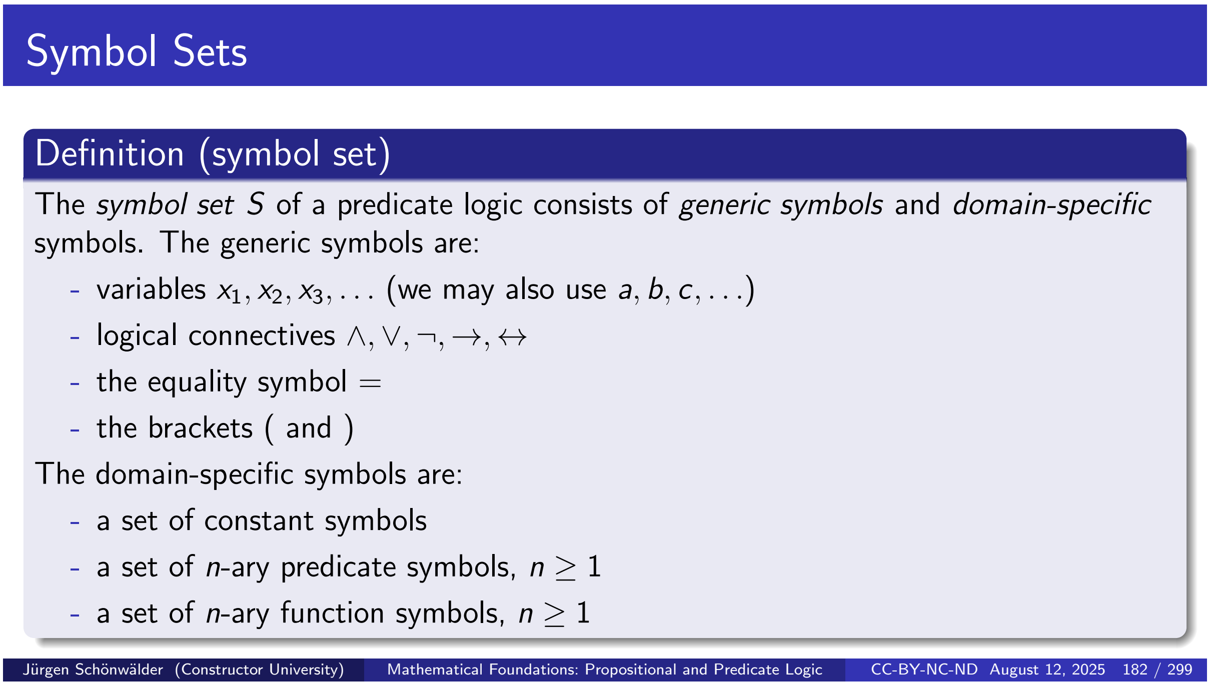
Further online information:

- **Wikipedia**: Boolean satisfiability problem
- **Wikipedia**: Millennium Prize Problems
- **YouTube**: P vs. NP – The Biggest Unsolved Problem in Computer Science

Section 22: Predicate Logic

21 Propositional Logic

22 Predicate Logic



Symbol Sets

Definition (symbol set)

The *symbol set S* of a predicate logic consists of *generic symbols* and *domain-specific* symbols. The generic symbols are:

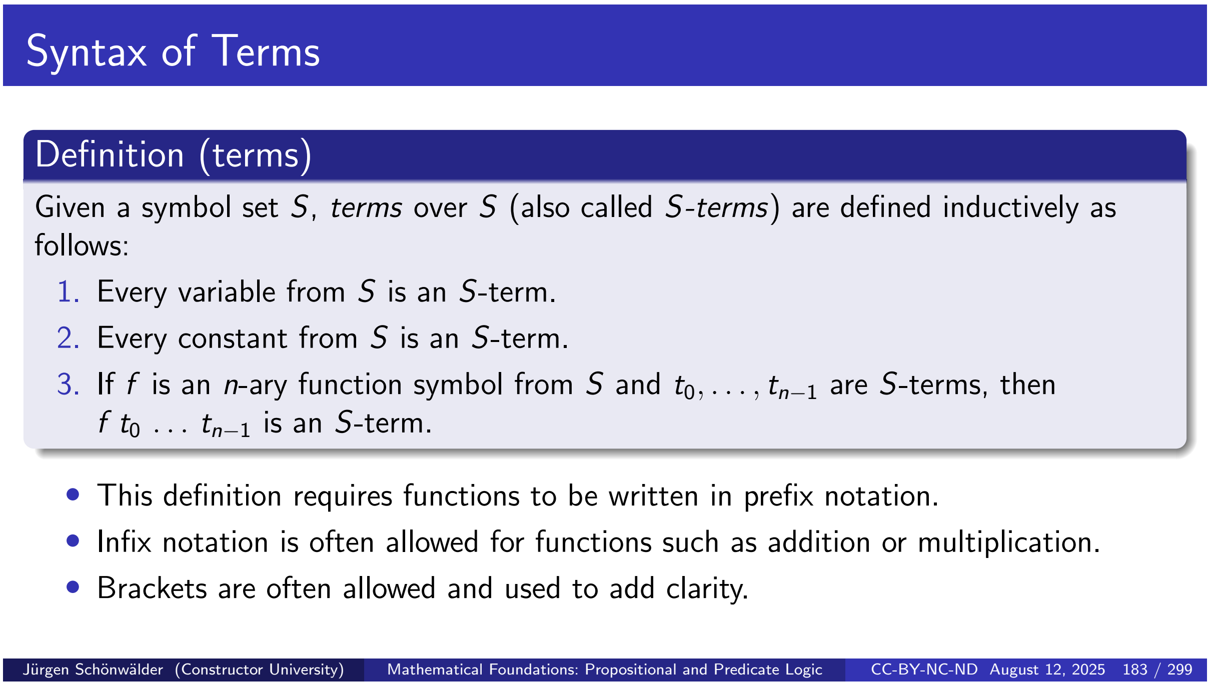
- variables $x_1, x_2, x_3, \ldots$ (we may also use $a, b, c, \ldots$)
- logical connectives $\land, \lor, \neg, \rightarrow, \leftrightarrow$
- the equality symbol $=$
- the brackets (and)

The domain-specific symbols are:

- a set of constant symbols
- a set of n-ary predicate symbols, $n \geq 1$
- a set of n-ary function symbols, $n \geq 1$

Due to the domain-specific symbols in the symbol set, predicate logic is always adapted to particular domains. For proving theorems in mathematics, the constants are numbers such as $1, 2, \ldots$, the functions are basic operators such as $+$ and functions such as sin, and the predicates express relations such as $\geq$. For proving theorems in robotics, the symbol set may capture concepts such as positions, directions, landmarks etc.

The domain-specific symbol set is not necessarily finite. For example, there is an infinite set of constants representing numbers or there may be an infinite number of positions a spaceship may have.



Syntax of Terms

Definition (terms)

Given a symbol set S, *terms* over S (also called S-*terms*) are defined inductively as follows:

1. Every variable from S is an S-term.
2. Every constant from S is an S-term.
3. If f is an n-ary function symbol from S and $t_0, \ldots, t_{n-1}$ are S-terms, then $f\, t_0\, \ldots\, t_{n-1}$ is an S-term.

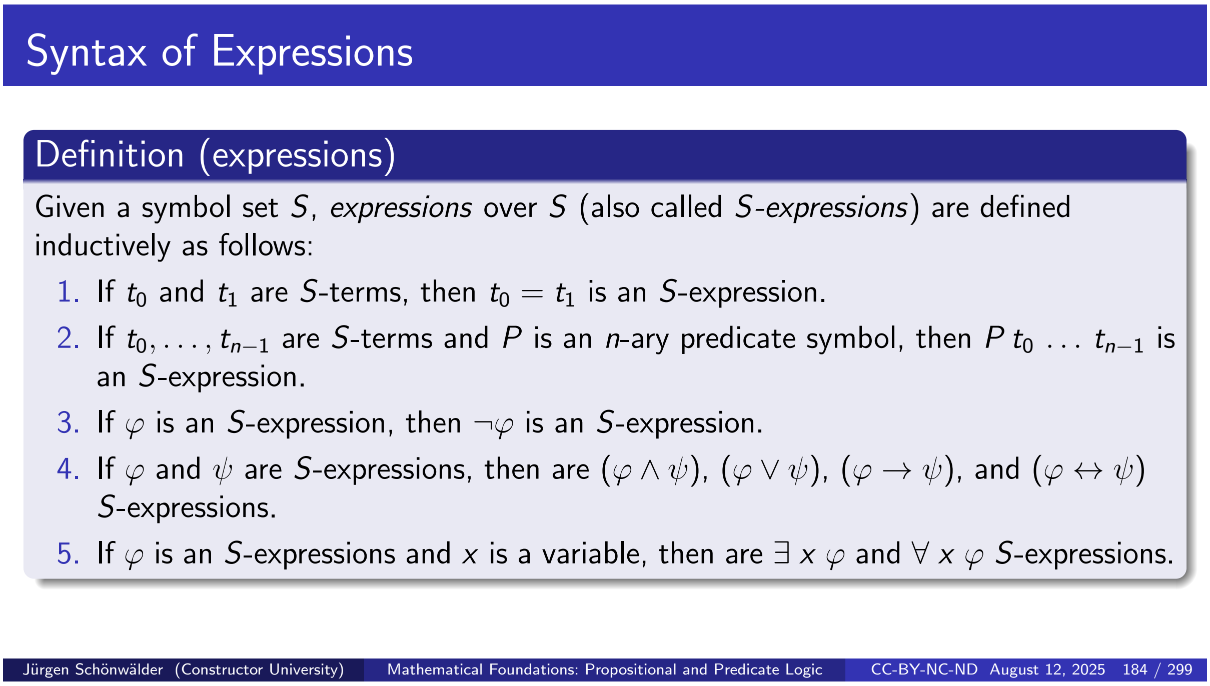
- This definition requires functions to be written in prefix notation.
- Infix notation is often allowed for functions such as addition or multiplication.
- Brackets are often allowed and used to add clarity.

According to the definition, function symbols are used in prefix notation. For example, we would write $+\,2\,3$ to apply the function $+$ to the two constants 2 and 3. Also note that there are no parenthesis involved and evaluation is right to left.

Example: Let S contain the constant symbol 0, the unary successor function σ, and the binary addition function $+$. Here are some ways to denote natural numbers:

0	constant symbol denoting the number 0
$\sigma\, 0$	denotes the successor of 0, that is, 1
$\sigma\,\sigma\,\sigma\, 0$	denotes 3, applying σ three times to 0
$+\ \sigma\, 0 \sigma\,\sigma\,\sigma\, 0$	denotes 4, adding 1 and 3
$+\ \sigma\, 0\, +\ \sigma\,\sigma\,\sigma\, 0\, \sigma\, 0$	denotes 5, adding 1 to the sum of 3 and 1

Of course, writing natural numbers in this way is cumbersome. It is far more convenient if the symbol set contains all natural numbers as constants. Note how the cardinality of the symbol set interacts with practical usability.



Syntax of Expressions

Definition (expressions)

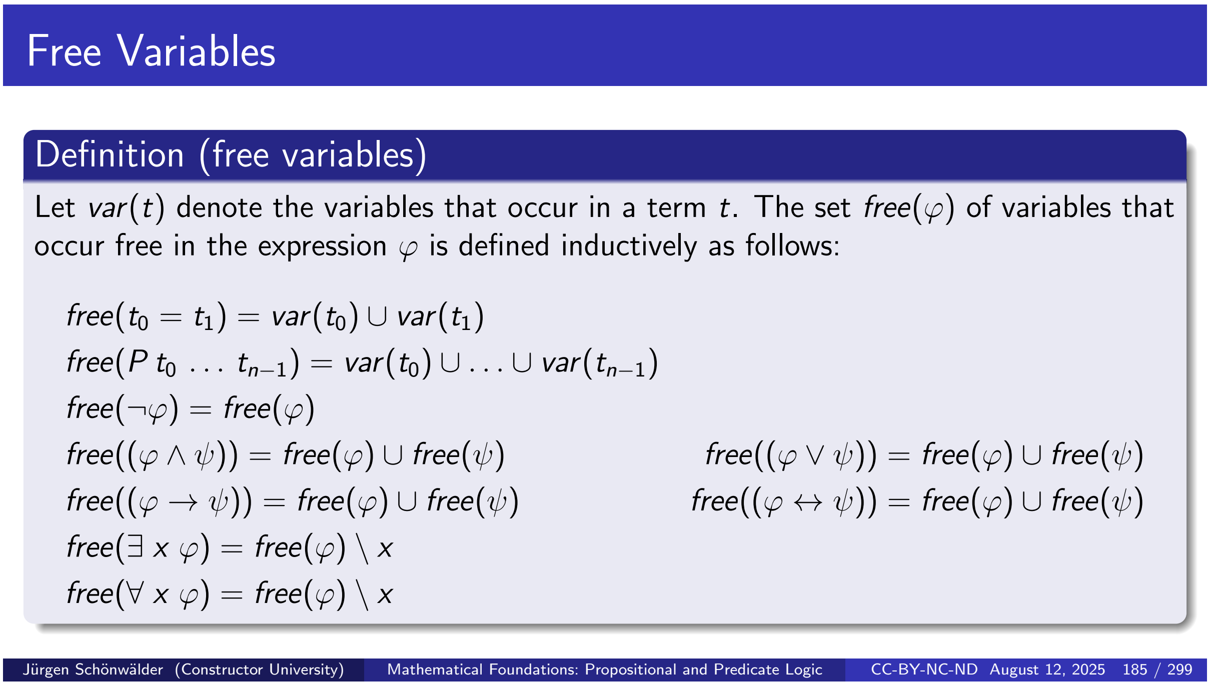
Given a symbol set S, *expressions* over S (also called S-*expressions*) are defined inductively as follows:

1. If t_0 and t_1 are S-terms, then $t_0 = t_1$ is an S-expression.
2. If $t_0, \dots, t_{n-1}$ are S-terms and P is an n-ary predicate symbol, then $P\, t_0 \, \dots \, t_{n-1}$ is an S-expression.
3. If φ is an S-expression, then $\neg\varphi$ is an S-expression.
4. If φ and ψ are S-expressions, then are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, and $(\varphi \leftrightarrow \psi)$ S-expressions.
5. If φ is an S-expressions and x is a variable, then are $\exists\, x\, \varphi$ and $\forall\, x\, \varphi$ S-expressions.

S-expressions define the formal syntax of a predicate logic for a given domain with a specific symbol set S. A formula is well-formed (a so-called well-formed formula), if it follows the rules for S-expressions. In practice, we often allow variations to make it simpler for humans to read and write expressions, such as the usage of infix operators and brackets in terms.

Examples:

$$+\ 1\,1 = 2$$
$$\text{prime}\ 5$$
$$\forall\, x\ (\text{prime}\ x \vee \neg\text{prime}\ x)$$
$$\exists\, x\ (\text{even}\ x \wedge \text{prime}\ x)$$
$$\forall\, x\ \neg(x = \sigma\, x)$$
$$\forall\, x\ \neg\exists\, y\ ((x = \sigma\, y) \rightarrow (x = 0))$$



Free Variables

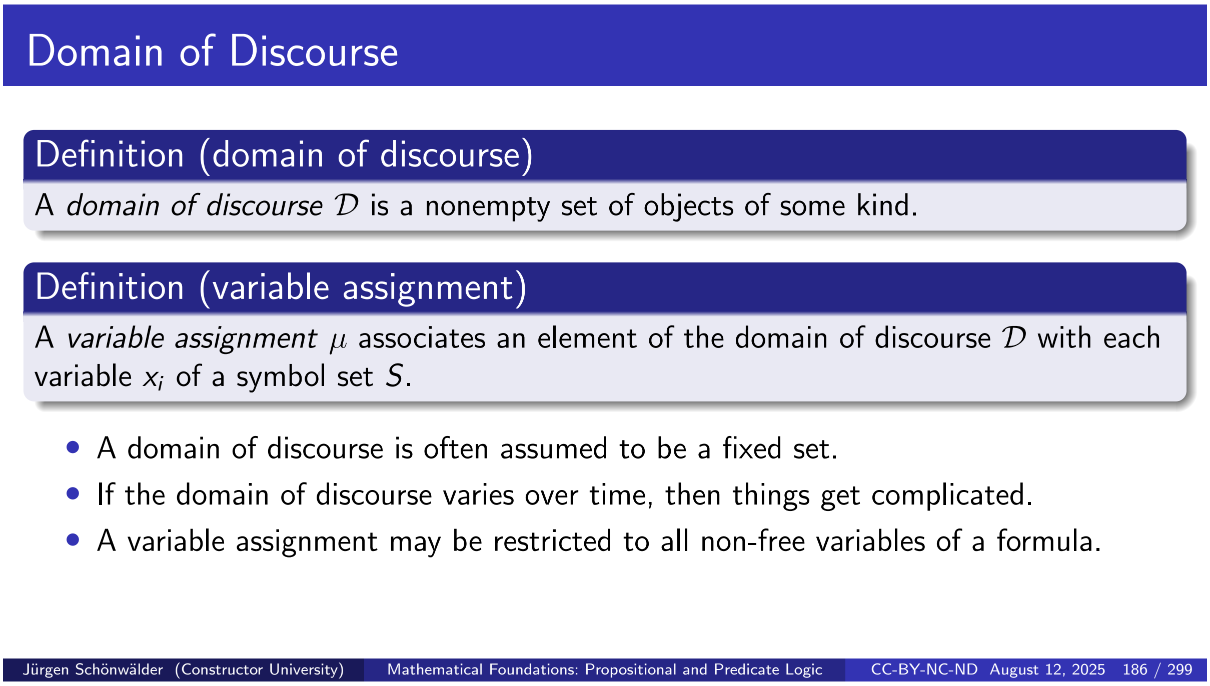
Definition (free variables)

Let $\mathit{var}(t)$ denote the variables that occur in a term t. The set $\mathit{free}(\varphi)$ of variables that occur free in the expression φ is defined inductively as follows:

$$\mathit{free}(t_0 = t_1) = \mathit{var}(t_0) \cup \mathit{var}(t_1)$$
$$\mathit{free}(P\; t_0\; \dots\; t_{n-1}) = \mathit{var}(t_0) \cup \dots \cup \mathit{var}(t_{n-1})$$
$$\mathit{free}(\neg\varphi) = \mathit{free}(\varphi)$$
$$\mathit{free}((\varphi \wedge \psi)) = \mathit{free}(\varphi) \cup \mathit{free}(\psi) \qquad \mathit{free}((\varphi \vee \psi)) = \mathit{free}(\varphi) \cup \mathit{free}(\psi)$$
$$\mathit{free}((\varphi \rightarrow \psi)) = \mathit{free}(\varphi) \cup \mathit{free}(\psi) \qquad \mathit{free}((\varphi \leftrightarrow \psi)) = \mathit{free}(\varphi) \cup \mathit{free}(\psi)$$
$$\mathit{free}(\exists\; x\; \varphi) = \mathit{free}(\varphi) \setminus x$$
$$\mathit{free}(\forall\; x\; \varphi) = \mathit{free}(\varphi) \setminus x$$

Examples:

- The expression $\leq\; 1\, 2$ has the set of free variables $\emptyset$.
- The expression $+\, x\, 1$ has the set of free variables $\{x\}$.
- The expression $\forall\, x\, (\geq\; x\, z)$ has the set of free variables $\{z\}$.
- The expression $\exists\, z\, \forall\, x\, (\geq\; x\, z)$ has the set of free variables $\emptyset$.



Domain of Discourse

Definition (domain of discourse)

A *domain of discourse* $\mathcal{D}$ is a nonempty set of objects of some kind.

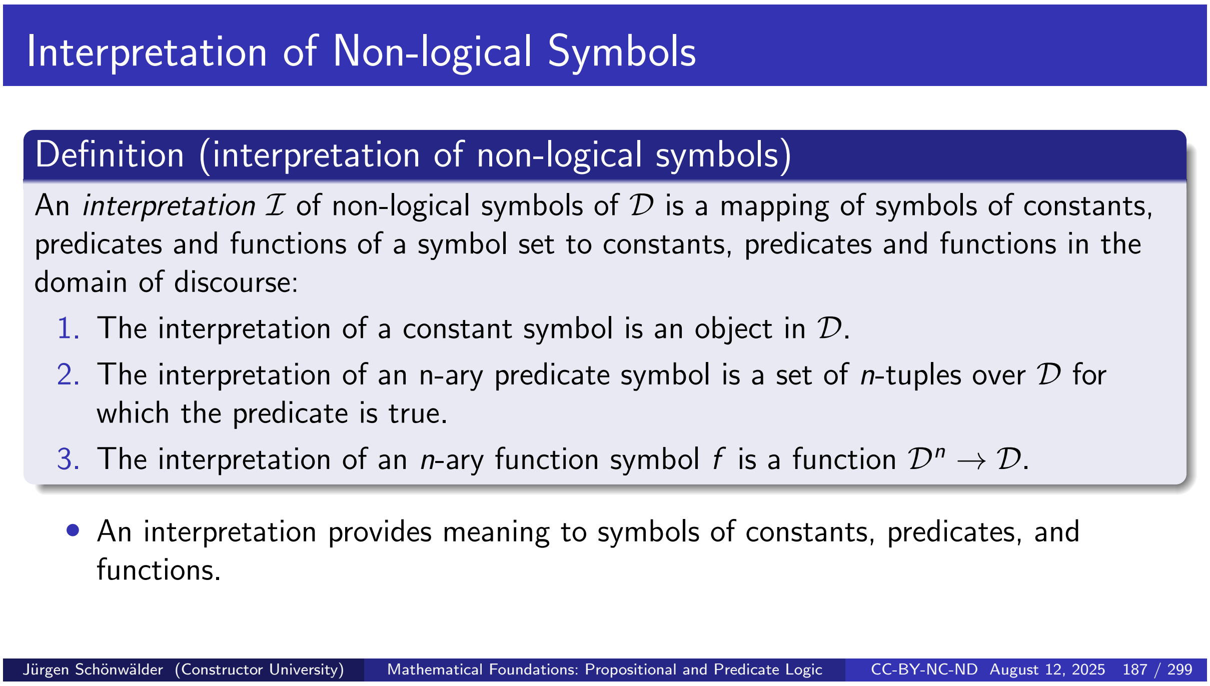
Definition (variable assignment)

A *variable assignment* μ associates an element of the domain of discourse $\mathcal{D}$ with each variable x_i of a symbol set S.

- A domain of discourse is often assumed to be a fixed set.
- If the domain of discourse varies over time, then things get complicated.
- A variable assignment may be restricted to all non-free variables of a formula.

The truth of a formula in predicate logic apparently depends on the domain of discourse. In practice, the domain of discourse is often not explicitly defined. However, if we want to write computer programs that can operate on statements in predicate logic, we will have to povide the domain of discourse.

In practice, we often use an element hood relation $\in$ to denote to which sets a variable is bound, i.e., we write $\forall x \in \mathbb{N}\,(x \geq 0)$. If we would have stated that our domain of discourse is $\mathbb{N}$, we could have just written $\forall x\,(x \geq 0)$. Note that even in the first style, there is an implicit domain of discourse.



Interpretation of Non-logical Symbols

Definition (interpretation of non-logical symbols)

An *interpretation* $\mathcal{I}$ of non-logical symbols of $\mathcal{D}$ is a mapping of symbols of constants, predicates and functions of a symbol set to constants, predicates and functions in the domain of discourse:

1. The interpretation of a constant symbol is an object in $\mathcal{D}$.
2. The interpretation of an n-ary predicate symbol is a set of n-tuples over $\mathcal{D}$ for which the predicate is true.
3. The interpretation of an n-ary function symbol f is a function $\mathcal{D}^n \to \mathcal{D}$.

- An interpretation provides meaning to symbols of constants, predicates, and functions.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Propositional and Predicate Logic CC-BY-NC-ND August 12, 2025 187 / 299

Note again the difference between a symbol such as 42 and the number fourty-two. We can have many symbols, such as 42, $0x2a$, or $0b101010$ mapping to the number fourty-two.

The usual interpretation of the symbol 10 is that it maps to the number ten. But for computer scientists, 10 could also map to the number two. It is therefore essential to know the domain of discourse. The same holds for functions and predicates. For some symbols, like $\leq$, we have a good chance to guess the correct meaning. But in math, to be fully precise, we should not have to guess the meaning of symbols.

While math is (so far) mostly read by humans who (presumably) do a good job in filling in the missing details, this usually does not work well in computer science since computers (so far) do not do a good enough job in filling in the missing details. While computer science may make progress by further developing AI systems to do better guesses, we also see a movement in modern math towards formally specified proofs that can be understood without any ambiguity by computer programs, such as proof assistants and theorem provers. At the end, checking math for correctness is as important as checking programs for correctness and we are meanwhile able to build tools to help with that.

Further online information:

- **Wikipedia**: Lean (proof assistant)

Interpretation of Logical Symbols

Definition (interpretation of logical symbols)

The *interpretation* $\mathcal{I}^*$ of an expression using the domain $\mathcal{D}$ and the interpretation $\mathcal{I}$ of non-logical symbols is defined inductively:

1. $P\, t_0 \dots t_{n-1}$ is true if and only if the tuple $(v_0, \dots, v_n)$ is in the interpretation $\mathcal{I}$ of P and $v_0, \dots, v_{n-1}$ are the interpretations of $t_0, \dots, t_{n-1}$.
2. $t_0 = t_1$ is true if and only if t_0 and t_1 evaluate to the same object in $\mathcal{D}$.
3. $\neg\varphi$ is true if and only if the interpretation of φ is false.

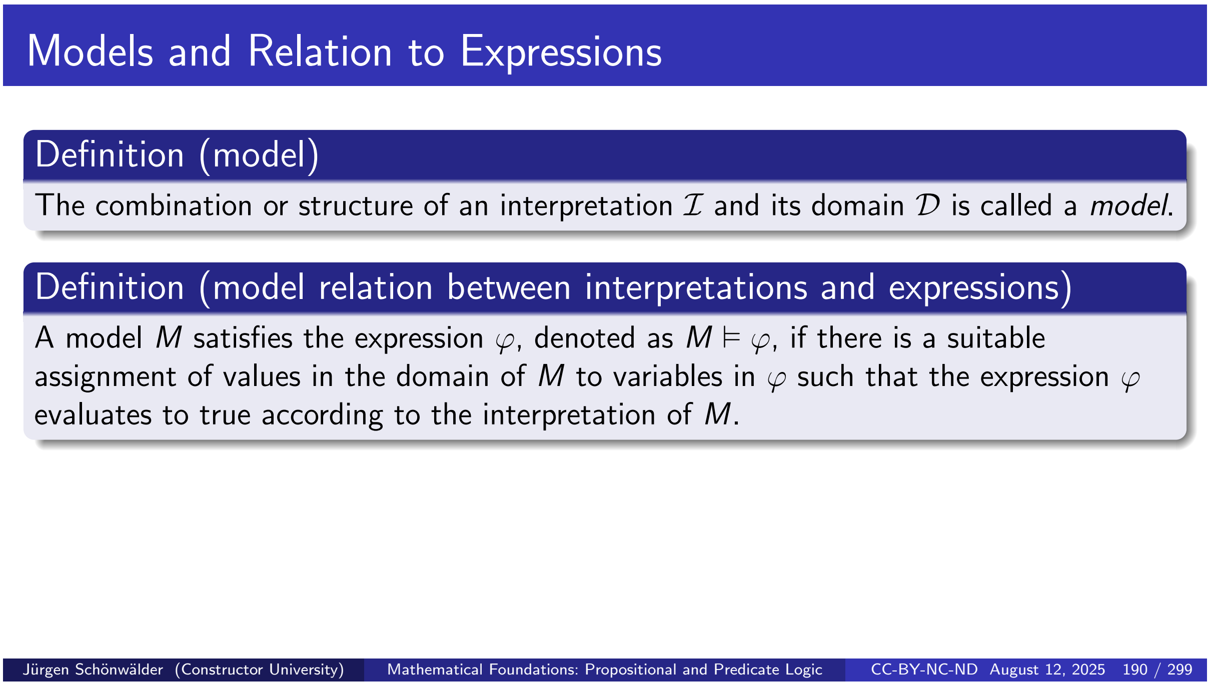
Interpretation of Logical Symbols (cont.)

Definition (interpretation of logical symbols)

The *interpretation* $\mathcal{I}^*$ of an expression using the domain $\mathcal{D}$ and the interpretation $\mathcal{I}$ of non-logical symbols is defined inductively:

4. $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, and $(\varphi \leftrightarrow \psi)$ are true if and only if the interpretations of φ and ψ satisfy the truth tables of the logical connectives.
5. $\exists x\, \varphi$ is true for a variable assignment μ if and only if there exists *at least one* interpretation for a variable assignment μ' that differs from μ at most regarding x.
6. $\forall x\, \varphi$ is true for a variable assignment μ if and only if it is true for *all* interpretations for a variable assignment μ' that differs from μ at most regarding x.

The interpretation $\mathcal{I}^*$ is defined as an extension of the interpretation $\mathcal{I}$. For convenience, we will write just $\mathcal{I}$ to refer to both of them if the distinction between them is not relevant.



Models and Relation to Expressions

Definition (model)

The combination or structure of an interpretation $\mathcal{I}$ and its domain $\mathcal{D}$ is called a *model*.

Definition (model relation between interpretations and expressions)

A model M satisfies the expression φ, denoted as $M \models \varphi$, if there is a suitable assignment of values in the domain of M to variables in φ such that the expression φ evaluates to true according to the interpretation of M.

It follows that for a given expression φ, there can be models satisfying φ and models not satisfying φ. Hence, the understanding and agreement on models is crucial for interpreting expressions in predicate logic.

Here are some examples:

1. Let $\mathcal{D} = \{a, b\}$ and $\mathcal{I}(p) = \{(a, b), (b, a)\}$. Let M denote the model obtained from the combination of the domain $\mathcal{D}$ and the interpretation $\mathcal{I}$.

   Then the interpretation of the formula $\phi = \forall x\, \exists y\, p\, x\, y$ holds since x ranges over $\mathcal{D}$ and there is an element (a, b) and an element (b, a) in $\mathcal{I}(p)$. Hence, $\mathcal{M} \models \phi$.

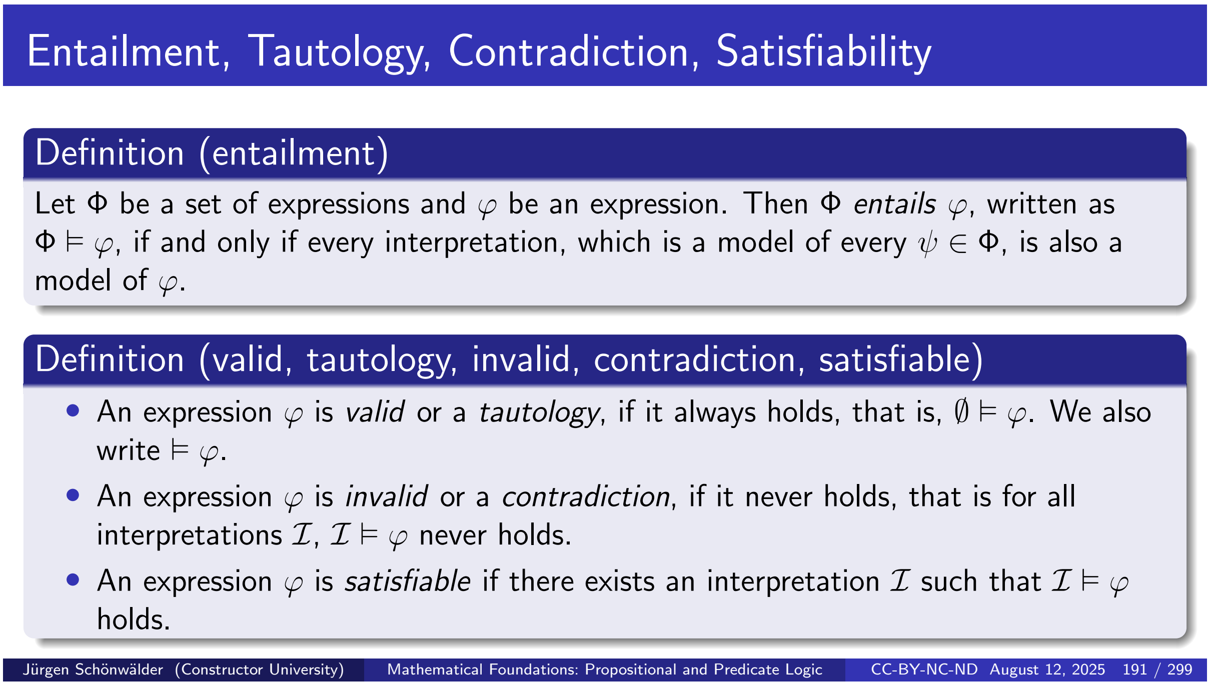
2. Let $\mathcal{D} = \{a, b\}$ and $\mathcal{I}(p) = \{(a, b), (b, a)\}$. Let M denote the model obtained from the combination of the domain $\mathcal{D}$ and the interpretation $\mathcal{I}$.

   Then the formula $\phi = \exists x\, \forall y\, p\, x\, y$ is not satisfiable since there are no elements (a, a) and (b, b) in $\mathcal{I}(P)$. Hence, $\mathcal{M} \not\models \phi$.

3. Let $\mathcal{D} = \mathbb{N}$ and $\mathcal{I}(p) = \{(x, y) | x \leq y\}$. Let M denote the model obtained from the combination of the domain $\mathcal{D}$ and the interpretation $\mathcal{I}$.

   Then the formula $\phi = \exists x\, \forall y\, p\, x\, y$ is satisfied by choosing the smallest natural number for x. Hence, $\mathcal{M} \models \phi$.

As these examples demonstrate, it is crucial to know the model in order to reason whether a formula is satisfied.



Entailment, Tautology, Contradiction, Satisfiability

Definition (entailment)

Let Φ be a set of expressions and φ be an expression. Then Φ *entails* φ, written as $\Phi \models \varphi$, if and only if every interpretation, which is a model of every $\psi \in \Phi$, is also a model of φ.

Definition (valid, tautology, invalid, contradiction, satisfiable)

- An expression φ is *valid* or a *tautology*, if it always holds, that is, $\emptyset \models \varphi$. We also write $\models \varphi$.
- An expression φ is *invalid* or a *contradiction*, if it never holds, that is for all interpretations $\mathcal{I}$, $\mathcal{I} \models \varphi$ never holds.
- An expression φ is *satisfiable* if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \varphi$ holds.

The validity problem, namely determining whether an arbitrary expression φ is valid, is undecidable for first-order logic.

Soundness and Completeness

Definition (soundness)

The *soundness* of a calculus over first-order logic means that if we can carry out a mechanical proof in the syntactical sense, then it is actually a sound proof in the semantic sense.

Definition (completeness)

The *completeness* of a calculus over first-order logic states that if an expression is entailed by some other expressions, it can also be derived by carrying out a mechanical proof in the syntactical sense.

- A calculus over first-order logic can have both soundness and completeness.

Equivalence of Predicate Logic Formulas

Definition (predicate logic formula equivalence)

Two predicate logic formulas F and G are logically equivalent if and only if $F \models G$ and $G \models F$.

- F and G are equivalent iff the truth values of F and G are the same under all possible interpretations.

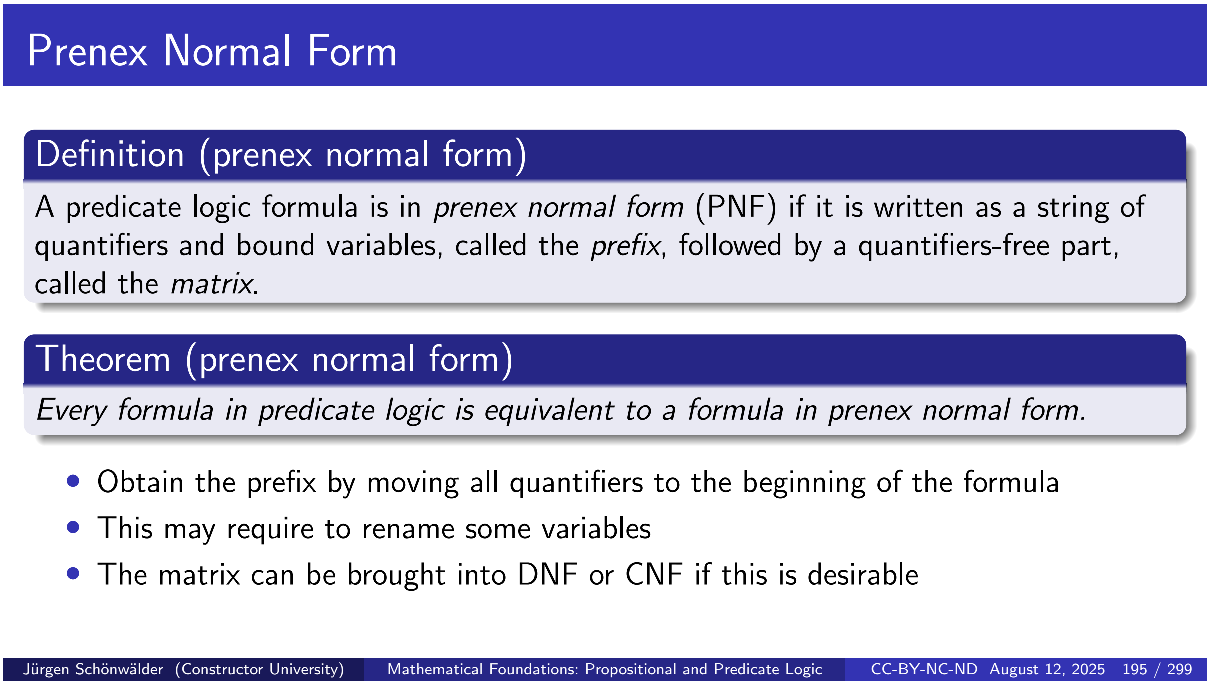
Equivalence Laws for Quantifiers

Theorem (quantifier equivalence laws)

Let p, q, r be predicates. Then the following equivalence laws hold:

1. $\neg(\forall x\, p\, x) \equiv \exists x\, \neg p\, x$ *(quantifier negation)*
2. $\neg(\exists x\, p\, x) \equiv \forall x\, \neg p\, x$ *(quantifier negation)*
3. $\forall x\, (p\, x \wedge q\, x) \equiv (\forall x\, p\, x) \wedge (\forall x\, q\, x)$ *(quantifier distribution)*
4. $\exists x\, (p\, x \vee q\, x) \equiv (\exists x\, p\, x) \vee (\exists x\, q\, x)$ *(quantifier distribution)*
5. $\forall x\, \forall y\, r\, x\, y \equiv \forall y\, \forall x\, r\, x\, y$ *(quantifier independence)*
6. $\exists x\, \exists y\, r\, x\, y \equiv \exists y\, \exists x\, r\, x\, y$ *(quantifier independence)*

We can extend the equivalence rule we already know from Boolean algebra to logical connectives. But since quantifiers are a new construct in predicate logic, we can determine equivalence laws for quantifiers. Here are some of them.



Prenex Normal Form

Definition (prenex normal form)

A predicate logic formula is in *prenex normal form* (PNF) if it is written as a string of quantifiers and bound variables, called the *prefix*, followed by a quantifiers-free part, called the *matrix*.

Theorem (prenex normal form)

Every formula in predicate logic is equivalent to a formula in prenex normal form.

- Obtain the prefix by moving all quantifiers to the beginning of the formula
- This may require to rename some variables
- The matrix can be brought into DNF or CNF if this is desirable

The conversion of a predicate logic formula into PNF generally follows the following steps:

- Resolve all logical connectives into $\land$, $\lor$, $\neg$
- Replace negated quantifiers where necessary
- Rename bound variables where necessary
- Move all quantifiers in order to the left

Example:

$$
\begin{aligned}
\varphi &= \neg((\neg\forall x\,(p\,x) \lor \forall x\,(q\,x)) \land ((r\,x) \rightarrow \forall x\,(q\,x))) && \text{resolving the implication}\\
&= \neg((\neg\forall x\,(p\,x) \lor \forall x\,(q\,x)) \land (\neg(r\,x) \lor \forall x\,(q\,x))) && \text{applying de Morgan outside}\\
&= (\neg(\neg\forall x\,(p\,x)) \lor (\forall x\,(q\,x))) \lor (\neg(\neg(r\,x) \lor \forall x\,(q\,x))) && \text{applying de Morgan inside}\\
&= ((\neg\neg\forall x\,(p\,x)) \land (\neg\forall x\,(q\,x))) \lor (\neg\neg(r\,x) \land (\neg\forall x\,(q\,x))) && \text{resolve double negation}\\
&= ((\forall x\,(p\,x)) \land (\neg\forall x\,(q\,x))) \lor ((r\,x) \land (\neg\forall x\,(q\,x))) && \text{resolve negation of quantifiers}\\
&= ((\forall x\,(p\,x)) \land (\exists x\,\neg(q\,x))) \lor ((r\,x) \land (\exists x\,\neg(q\,x))) && \text{rename variables}\\
&= ((\forall a\,(p\,a)) \land (\exists b\,\neg(q\,b))) \lor ((r\,x) \land (\exists c\,\neg(q\,c))) && \text{move quantifiers}\\
&= \forall a\,\exists b\,\exists c\,((p\,a) \land \neg(q\,b)) \lor ((r\,x) \land \neg(q\,c))
\end{aligned}
$$

Note that the formula in PNF still has the free variable x. Do not rename free variables since they have a meaning in the context where the formula appears.

Part VI

Abstract Algebra

This part introduces basic elements of abstract algebra, which is the study of algebraic structures. An algebraic structure is essentially a set with a collection of operations acting on the elements of the set. Abstract algebra has a strong relation to various areas of computer science, such as cryptography, the theory of automata and formal languages, and complexity theory. Furthermore, the application of concepts of abstract algebra has led to the discovery of more efficient algorithms.

By the end of this part, students should be able to

- explain the purpose and role of abstract algebra;
- define algebraic structures such as monoids, groups, rings, or fields;
- relate algebraic structures to each other;
- conclude whether a given algebraic structure is a group or not;
- illustrate the notion of subgroups and how their cosets partition a group;
- summarize Lagrange's theorem and explain its practical value;
- decide whether an algebraic structure is a ring or a field;
- describe different kinds of homomorphisms;
- sketch properties of group homomorphisms;
- summarize Cayley's theorem;
- recite the definition of lattices and distributive lattices.

Algebraic Structures

Definition (algebraic structure)

An *algebraic structure* consists of a nonempty set S (called the underlying set, carrier set or domain), a collection of operations on S (typically binary operations such as addition and multiplication), and a finite set of axioms that these operations must satisfy.

- A branch of mathematics known as *universal algebra* studies algebraic structures.
- A set S is a degenerate algebraic structure having no operations.
- We are interested in algebraic structures with one or multiple defined operations.

Abstract algebra enables us to prove theorems for several similar algebraic structures very efficiently. As we will see, basic set theory and boolean logic have many similarities. By identifying an algebraic structure underlying both basic set theory and boolean logic, we can define and prove theorems once that apply to both basic set theory and boolean logic.

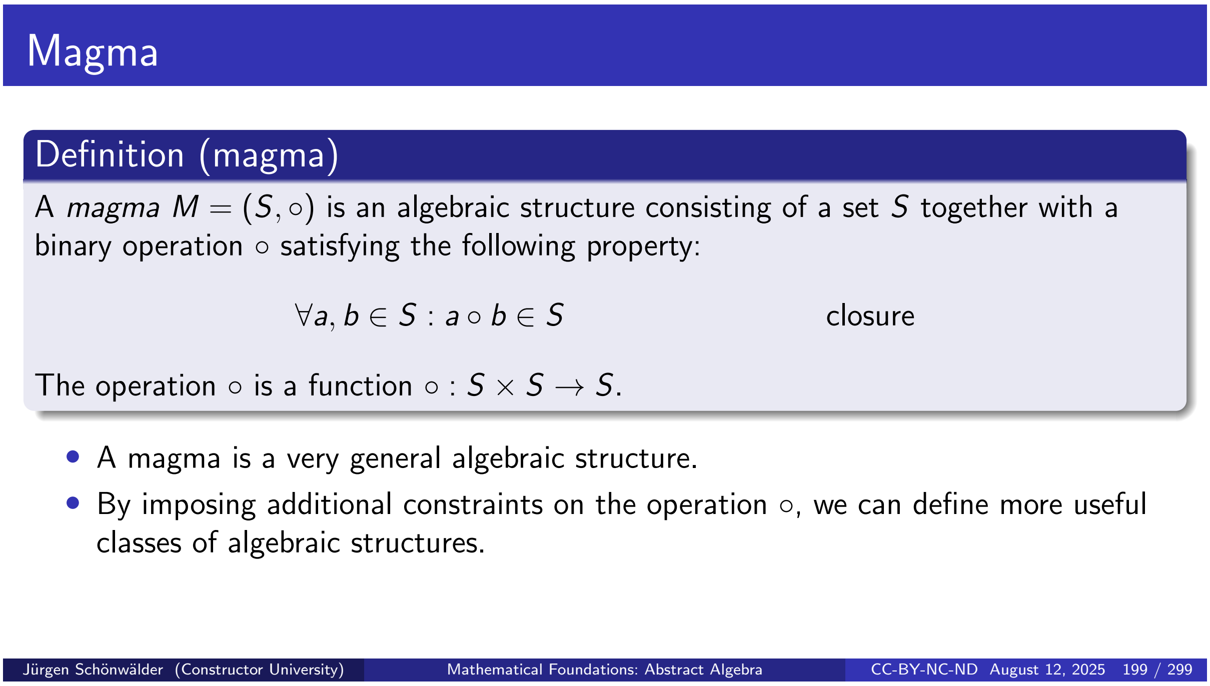
Section 23: Magmas, Semigroups, Monoids, Groups

23 Magmas, Semigroups, Monoids, Groups

24 Rings and Fields

25 Homomorphisms

26 Lattices



Magma

Definition (magma)

A *magma* $M = (S, \circ)$ is an algebraic structure consisting of a set S together with a binary operation $\circ$ satisfying the following property:

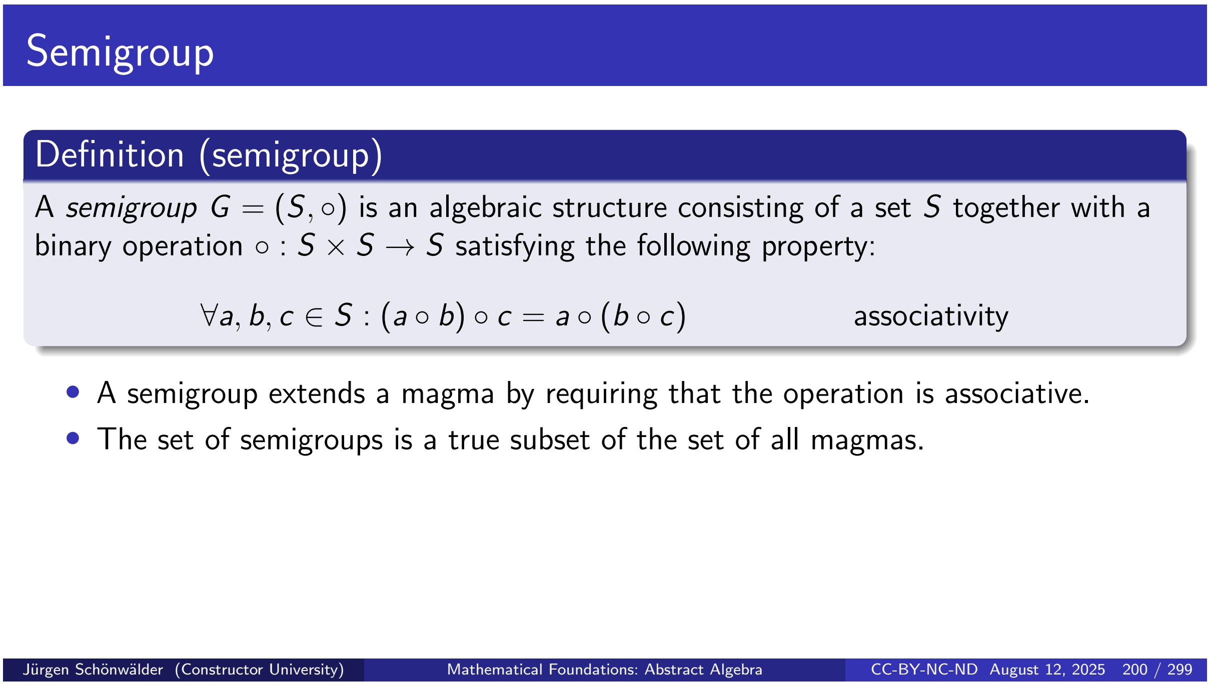
$$\forall a, b \in S : a \circ b \in S \qquad\qquad \text{closure}$$

The operation $\circ$ is a function $\circ : S \times S \to S$.

- A magma is a very general algebraic structure.
- By imposing additional constraints on the operation $\circ$, we can define more useful classes of algebraic structures.

Some examples for magmas:

1. The magma $M = (\mathbb{N}, +)$ represents addition over natural numbers.
2. The magma $M = (\mathbb{N}, \cdot)$ represents multiplication over natural numbers.
3. The magma $M = (\mathbb{R}^3, *)$ with the operation $*$ defined as $(a, b, c) * (x, y, z) = (bz - cy, cx - az, ay - bx)$ is an interesting magma since it is not any of the algebraic structures we define next.



Semigroup

Definition (semigroup)

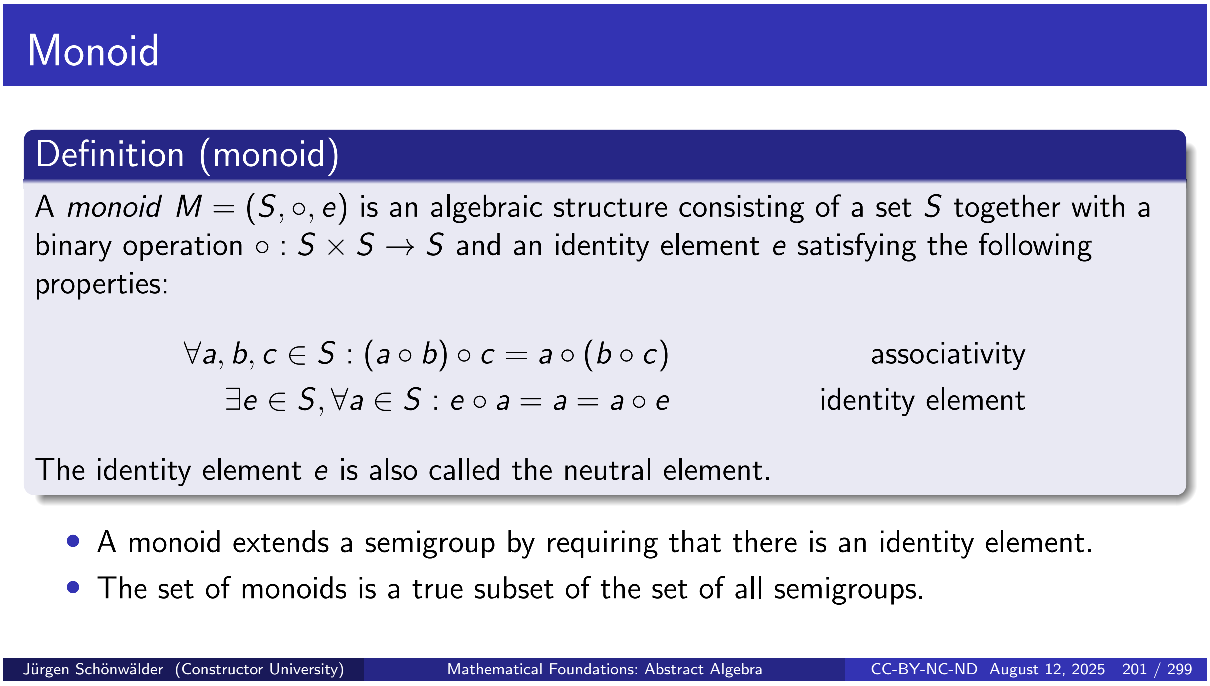
A *semigroup* $G = (S, \circ)$ is an algebraic structure consisting of a set S together with a binary operation $\circ : S \times S \to S$ satisfying the following property:

$$\forall a, b, c \in S : (a \circ b) \circ c = a \circ (b \circ c) \qquad \text{associativity}$$

- A semigroup extends a magma by requiring that the operation is associative.
- The set of semigroups is a true subset of the set of all magmas.

Some examples for semigroups:

1. $(\mathbb{N}, +)$ (addition over natural numbers)
2. $(\mathbb{N}, \cdot)$ (multiplication over natural numbers)
3. $(\mathbb{Z}, +)$ (addition over integer numbers)
4. $(\mathbb{Z}, \cdot)$ (multiplication over integer numbers)
5. $(\mathbb{Q}, +)$ (addition over rational numbers)
6. $(\mathbb{Q}, \cdot)$ (multiplication over rational numbers)
7. $(\mathbb{R}, +)$ (addition over real numbers)
8. $(\mathbb{R}, \cdot)$ (multiplication over real numbers)



Monoid

Definition (monoid)

A *monoid* $M = (S, \circ, e)$ is an algebraic structure consisting of a set S together with a binary operation $\circ : S \times S \to S$ and an identity element e satisfying the following properties:

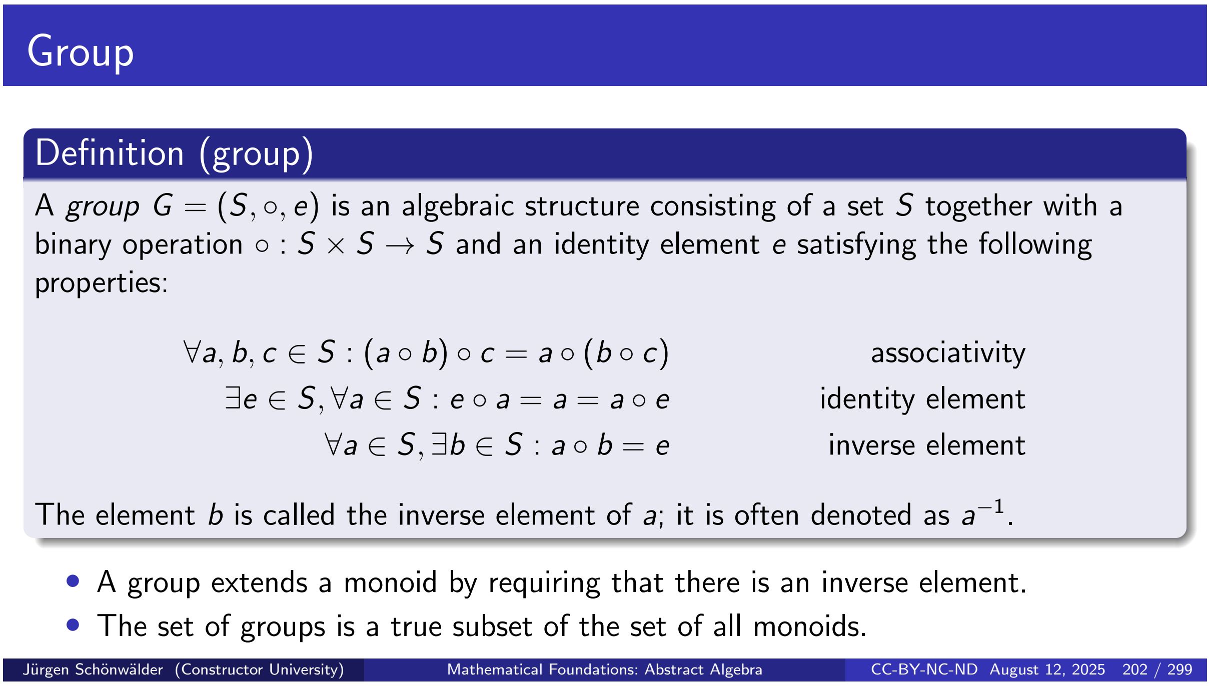
$$\begin{aligned} \forall a, b, c \in S : (a \circ b) \circ c = a \circ (b \circ c) & \qquad \text{associativity} \\ \exists e \in S, \forall a \in S : e \circ a = a = a \circ e & \qquad \text{identity element} \end{aligned}$$

The identity element e is also called the neutral element.

- A monoid extends a semigroup by requiring that there is an identity element.
- The set of monoids is a true subset of the set of all semigroups.

Some examples of monoids:

1. The monoid $M = (\mathbb{N}_0, +, 0)$ represents addition over natural numbers with identity 0.
2. The monoid $M = (\mathbb{R}, \cdot, 1)$ represents multiplication over real numbers with identity 1.
3. The monoid $M = (\mathbb{Z}, \cdot, 1)$ represents multiplication over integer numbers with identity 1.
4. The monoid $M = (\Sigma^*, +, \epsilon)$ with $a + b = ab$ for $a, b \in \Sigma^*$ and a finite alphabet Σ represents the string concatenation.



Group

Definition (group)

A *group* $G = (S, \circ, e)$ is an algebraic structure consisting of a set S together with a binary operation $\circ : S \times S \rightarrow S$ and an identity element e satisfying the following properties:

$$\begin{aligned} \forall a, b, c \in S : (a \circ b) \circ c = a \circ (b \circ c) && \text{associativity} \\ \exists e \in S, \forall a \in S : e \circ a = a = a \circ e && \text{identity element} \\ \forall a \in S, \exists b \in S : a \circ b = e && \text{inverse element} \end{aligned}$$

The element b is called the inverse element of a; it is often denoted as a^{-1}.

- A group extends a monoid by requiring that there is an inverse element.
- The set of groups is a true subset of the set of all monoids.

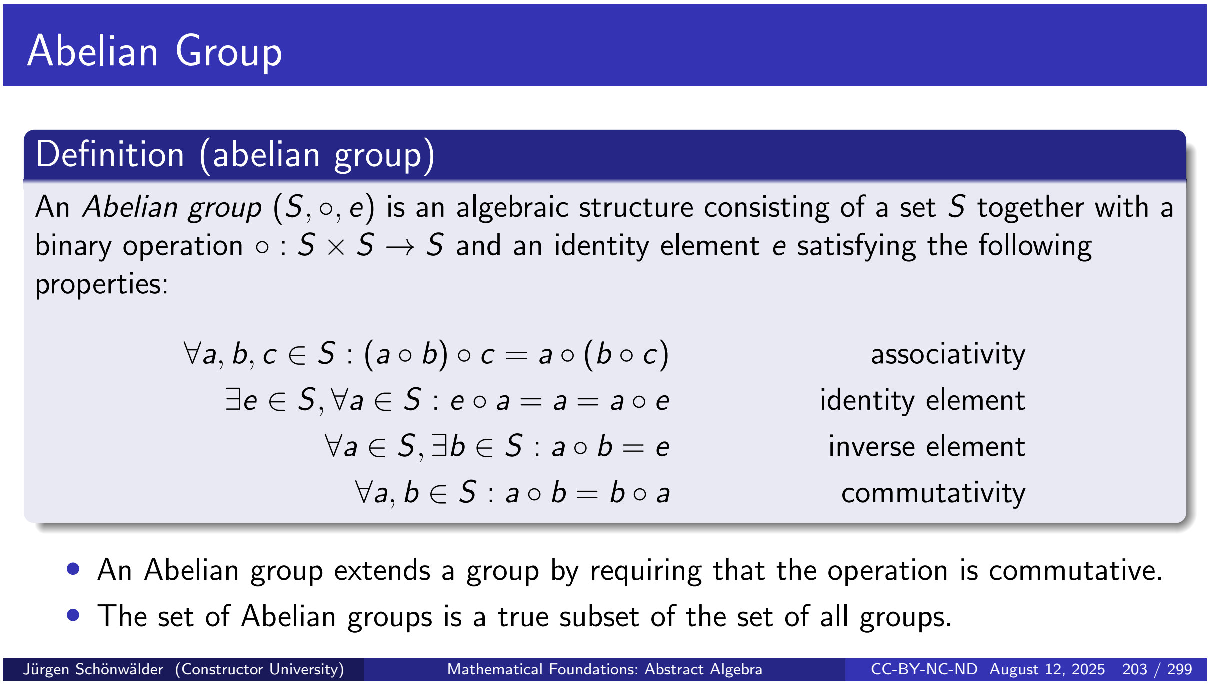
Some examples of groups:

1. The group $G = (\mathbb{Z}, +, 0)$ represents addition over integer numbers with identity 0.
2. The group $G = (\mathbb{Q} \setminus \{0\}, \cdot, 1)$ represents multiplication over rational numbers without zero and with identity 1. Note that we exclude zero since zero has no inverse element.
3. The group $G = (\mathbb{Z}_n, +, 0)$ represents addition modulo n over the integers $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$.
4. Dihedral groups capture the notion of symmetry of a regular polygon. A polygon with n sides has $2n$ different symmetries: n rotational symmetries $(r_0, r_1, \ldots, r_{n-1})$ and n reflection symmetries $(l_0, l_1, \ldots, l_{n-1})$. The rotations and reflections make up the dihedral group D_n (we usually assume $n \geq 3$). The group operation is the composition of rotations and reflections.

   Example D_3: The three rotations of an equilateral triangle are the rotation by 0 degrees (r_0), 120 degrees (r_1), and 240 degrees (r_2). The reflections are the reflection along the angle bisector of the corner A (l_0), the corner B (l_1), and the corner C (l_2). This gives us the dihedral group $D_3 = (\{r_0, r_1, r_2, l_0, l_1, l_2\}, \circ, r_0)$ with the composition $\circ$ defined by the following Cayley table:

$\circ$	r_0	r_1	r_2	l_0	l_1	l_2
r_0	r_0	r_1	r_2	l_0	l_1	l_2
r_1	r_1	r_2	r_0	l_2	l_0	l_1
r_2	r_2	r_0	r_1	l_1	l_2	l_0
l_0	l_0	l_1	l_2	r_0	r_1	r_2
l_1	l_1	l_2	l_0	r_2	r_0	r_1
l_2	l_2	l_0	l_1	r_1	r_2	r_0

   For $a, b \in G$, the value of $a \circ b$ is given by the cell where the row is determined by a and the column by b. (This detail is important since Cayley tables are not necessarily symmetric along the diagonal.)



Abelian Group

Definition (abelian group)

An *Abelian group* $(S, \circ, e)$ is an algebraic structure consisting of a set S together with a binary operation $\circ : S \times S \to S$ and an identity element e satisfying the following properties:

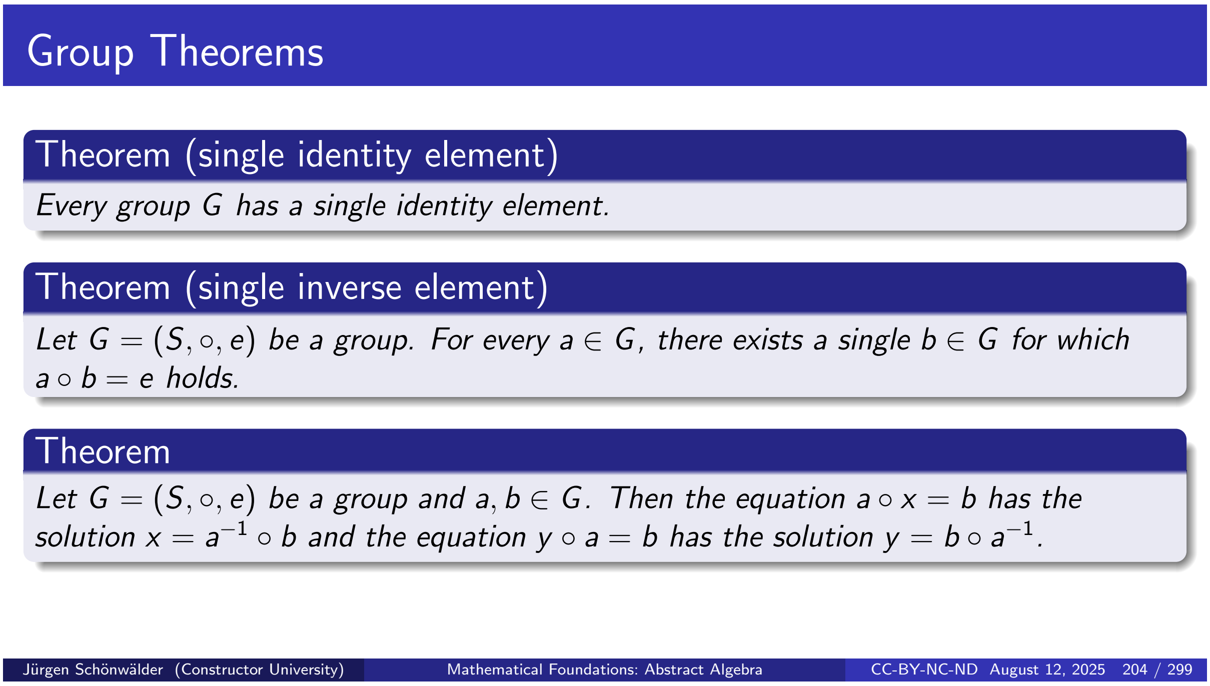
$$\begin{aligned} \forall a, b, c \in S : (a \circ b) \circ c &= a \circ (b \circ c) && \text{associativity} \\ \exists e \in S, \forall a \in S : e \circ a = a &= a \circ e && \text{identity element} \\ \forall a \in S, \exists b \in S : a \circ b &= e && \text{inverse element} \\ \forall a, b \in S : a \circ b &= b \circ a && \text{commutativity} \end{aligned}$$

- An Abelian group extends a group by requiring that the operation is commutative.
- The set of Abelian groups is a true subset of the set of all groups.

Some examples of Abelian groups:

1. The group $G = (\mathbb{Z}, +, 0)$ represents addition over integer numbers with identity 0.
2. The group $G = (\mathbb{Q} \setminus \{0\}, \cdot, 1)$ represents multiplication over rational numbers without zero and with identity 1.

The Dihedral group D_3 is not an Abelian group since the composition operation is not commutative. For example, $r_1 \circ l_1 = l_0$ while $l_1 \circ r_1 = l_2$. Note that the Cayley table of an Abelian group is symmetric along the diagonal (the Cayley table of D_3 is not symmetric).



Group Theorems

Theorem (single identity element)

Every group G has a single identity element.

Theorem (single inverse element)

Let $G = (S, \circ, e)$ be a group. For every $a \in G$, there exists a single $b \in G$ for which $a \circ b = e$ holds.

Theorem

Let $G = (S, \circ, e)$ be a group and $a, b \in G$. Then the equation $a \circ x = b$ has the solution $x = a^{-1} \circ b$ and the equation $y \circ a = b$ has the solution $y = b \circ a^{-1}$.

Proof. Let G be a group. Lets further assume that there are two identities $e, e' \in G$. We show that they must be the same.

By the definition of a group, we know that $g \circ e = g$ and $g \circ e' = g$ for every $g \in G$. Since $e, e' \in G$, we also know that $e' \circ e = e'$ and $e \circ e' = e$. This implies that $e = e'$ and $e' = e$. □

Proof. Let $G = (S, \circ, e)$ be a group and let $b \in G$ and $b' \in G$ be both inverse to $a \in G$. Then the following holds:

$$\begin{aligned} b &= e \circ b \\ &= (b' \circ a) \circ b \\ &= b' \circ (a \circ b) \\ &= b' \circ e \\ &= b' \end{aligned}$$

□

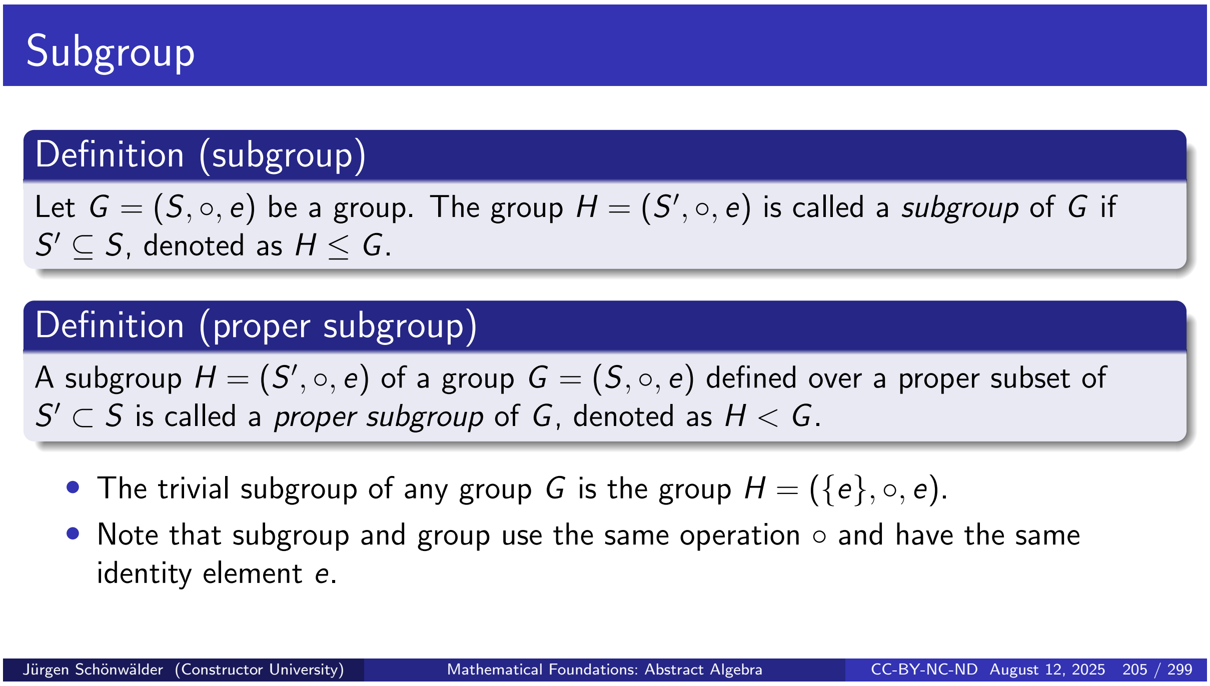
Proof. We first show that $x = a^{-1} \circ b$ solves $a \circ x = b$ and that $y = b \circ a^{-1}$ solves $y \circ a = b$.

$$\begin{aligned} a \circ x &= a \circ (a^{-1} \circ b) \\ &= (a \circ a^{-1}) \circ b \\ &= e \circ b \\ &= b \end{aligned} \qquad \begin{aligned} y \circ a &= (b \circ a^{-1}) \circ a \\ &= b \circ (a^{-1} \circ a) \\ &= b \circ e \\ &= b \end{aligned}$$

Next, we show that there is only a single solution. Note that a group is not necessarily commutative.

$$\begin{aligned} a \circ x &= b \\ a^{-1} \circ (a \circ x) &= a^{-1} \circ b \\ (a^{-1} \circ a) \circ x &= a^{-1} \circ b \\ e \circ x &= a^{-1} \circ b \\ x &= a^{-1} \circ b \end{aligned} \qquad \begin{aligned} y \circ a &= b \\ (y \circ a) \circ a^{-1} &= b \circ a^{-1} \\ y \circ (a \circ a^{-1}) &= b \circ a^{-1} \\ y \circ e &= b \circ a^{-1} \\ y &= b \circ a^{-1} \end{aligned}$$

□



Subgroup

Definition (subgroup)

Let $G = (S, \circ, e)$ be a group. The group $H = (S', \circ, e)$ is called a *subgroup* of G if $S' \subseteq S$, denoted as $H \leq G$.

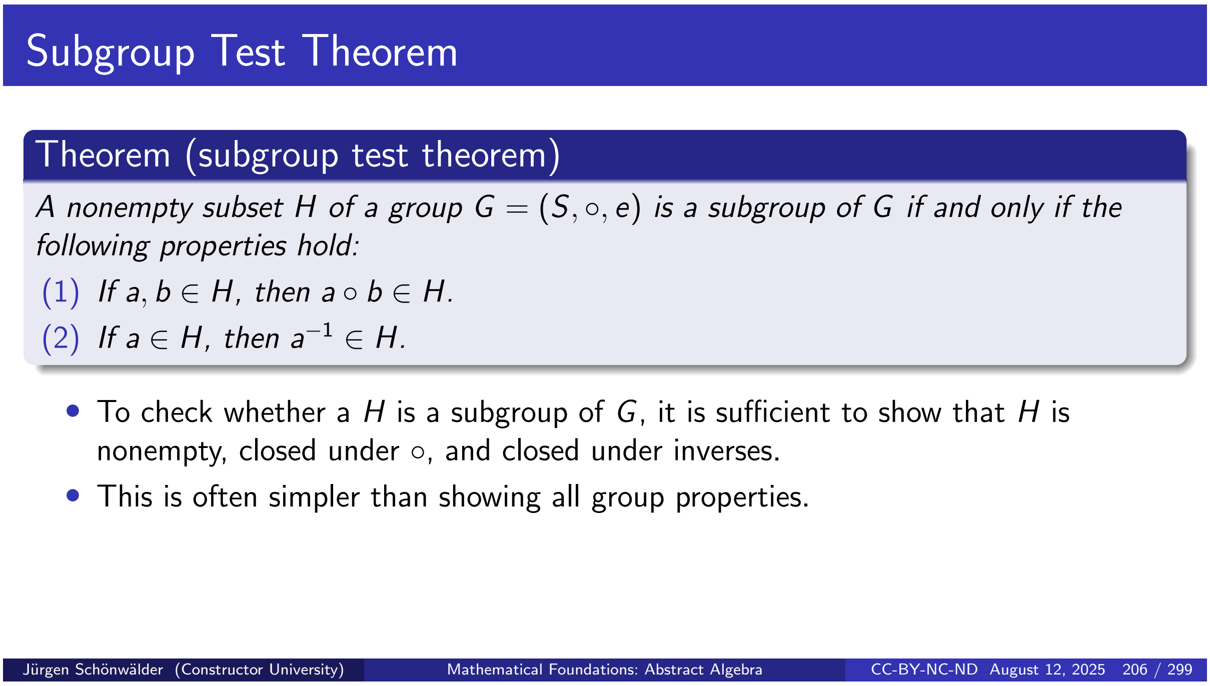
Definition (proper subgroup)

A subgroup $H = (S', \circ, e)$ of a group $G = (S, \circ, e)$ defined over a proper subset of $S' \subset S$ is called a *proper subgroup* of G, denoted as $H < G$.

- The trivial subgroup of any group G is the group $H = (\{e\}, \circ, e)$.
- Note that subgroup and group use the same operation $\circ$ and have the same identity element e.

Some examples of subgroups:

1. Consider the group $G = (\mathbb{Z}, +, 0)$. Let $E = \{\, 2x \,|\, x \in \mathbb{Z} \,\}$ be the set of all even integers. Then $H = (E, +, 0)$ is a proper subgroup of G. (Note that the set of all odd integers do not form a subgroup.)
2. The dihedral group D_3 has the proper subgroup $H_1 = (\{r_0, r_1, r_2\}, \circ, r_0)$, as can be seen from the Cayley table. Does D_3 have more proper non-trivial subgroups?



Subgroup Test Theorem

Theorem (subgroup test theorem)

A nonempty subset H of a group $G = (S, \circ, e)$ is a subgroup of G if and only if the following properties hold:

(1) *If $a, b \in H$, then $a \circ b \in H$.*

(2) *If $a \in H$, then $a^{-1} \in H$.*

- To check whether a H is a subgroup of G, it is sufficient to show that H is nonempty, closed under $\circ$, and closed under inverses.
- This is often simpler than showing all group properties.

Proof. We prove the equivalence by proving both implications.

⇒ Let H be a subgroup of the group G. We show that it is closed under $\circ$ and under inverse.

Since H is a group, we know that H is closed under $\circ$ and we know that every element $a \in H$ has an inverse $a^{-1} \in H$.

⇐ Let H be a nonempty subset of the group G that is closed under $\circ$ and under inverse. We show that H is a subgroup of G.

We have to check the requirements for a group. We already know that H is closed under $\circ$ and that every element in H has an inverse. So we need to show associativity and the existence of an identity element.

Let $a, b, c \in H$. Since H is a subset of G, $a, b, c \in G$ as well. Since G is a group, we know that $(a \circ b) \circ c = a \circ (b \circ c)$ holds. (And since H is closed under $\circ$, we know that the result is an element of H.)

We know that for every $a \in H$, $a^{-1} \in H$. Furthermore, we know that H is closed under $\circ$, hence $a \circ a^{-1} = e \in H$.

□

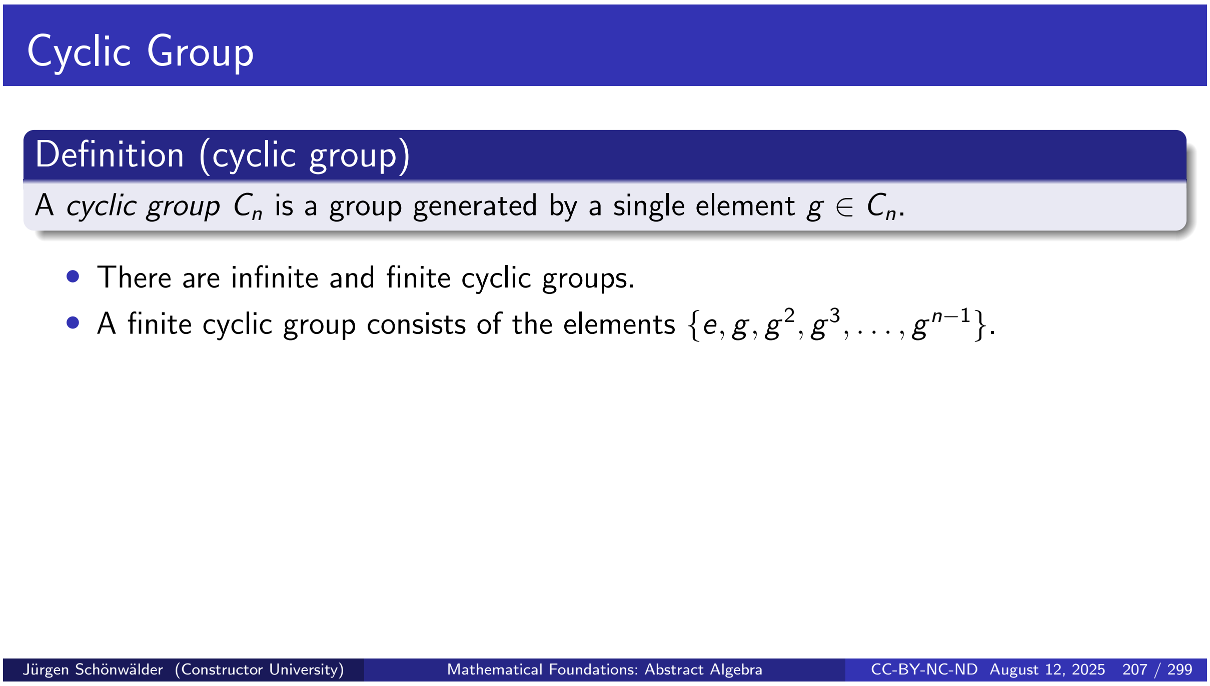
Example: Consider the group $G = (\mathbb{Z}_4, +, 0)$ where $+$ denotes addition modulo 4. We show that $H = (\{0, 2\}, +, 0)$ is a proper subgroup of G using the subgroup test theorem.

(1) Obviously, the group H is non-empty.

(2) We show $a, b \in H \rightarrow a \circ b \in H$ by simply considering all four possible cases:

$$\begin{aligned}
0 + 0 &= 0 \in H \\
0 + 2 &= 2 \in H \\
2 + 0 &= 2 \in H \\
2 + 2 &= 0 \in H
\end{aligned}$$

(3) We show $a \in H \rightarrow a^{-1} \in H$ by considering the two possible cases. The inverse of 0 is 0 (since $0 + 0 = 0$) and the inverse of 2 is 2 (since $2 + 2 = 0$). Both inverses are an element of H.



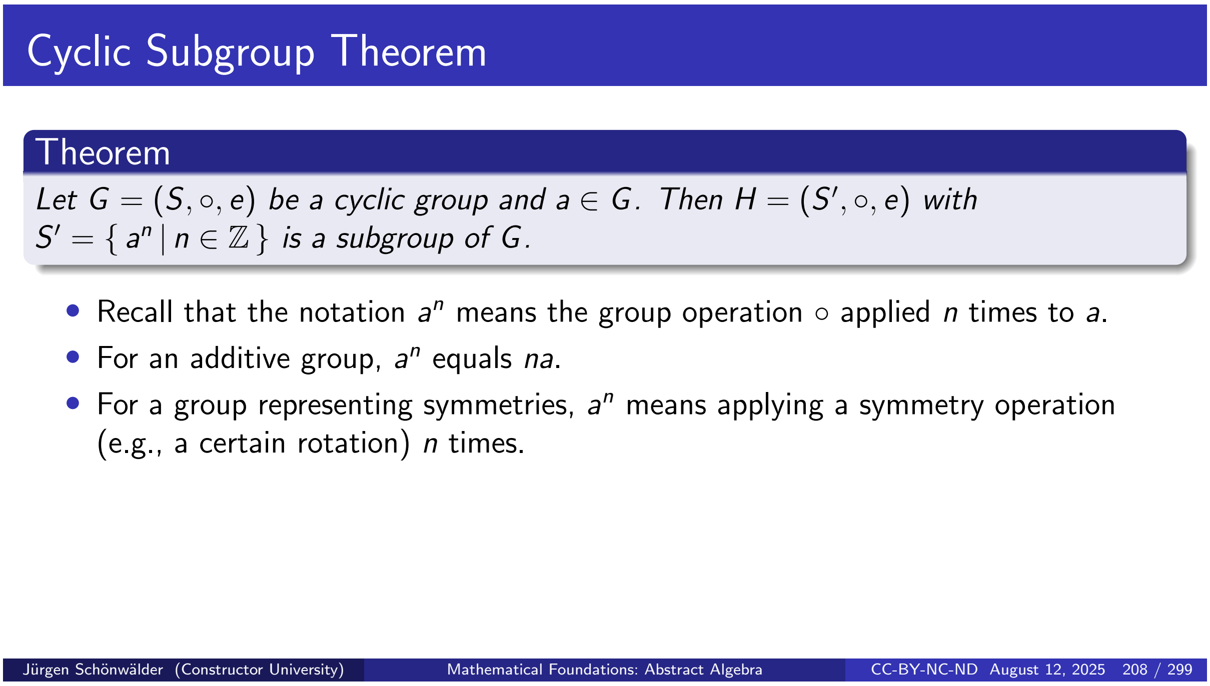
Cyclic Group

Definition (cyclic group)

A *cyclic group* C_n is a group generated by a single element $g \in C_n$.

- There are infinite and finite cyclic groups.
- A finite cyclic group consists of the elements $\{e, g, g^2, g^3, \ldots, g^{n-1}\}$.

Example: The group $(\mathbb{Z}_n, +, 0)$ with the addition module n is a cyclic group.



Cyclic Subgroup Theorem

Theorem

Let $G = (S, \circ, e)$ be a cyclic group and $a \in G$. Then $H = (S', \circ, e)$ with $S' = \{\, a^n \mid n \in \mathbb{Z} \,\}$ is a subgroup of G.

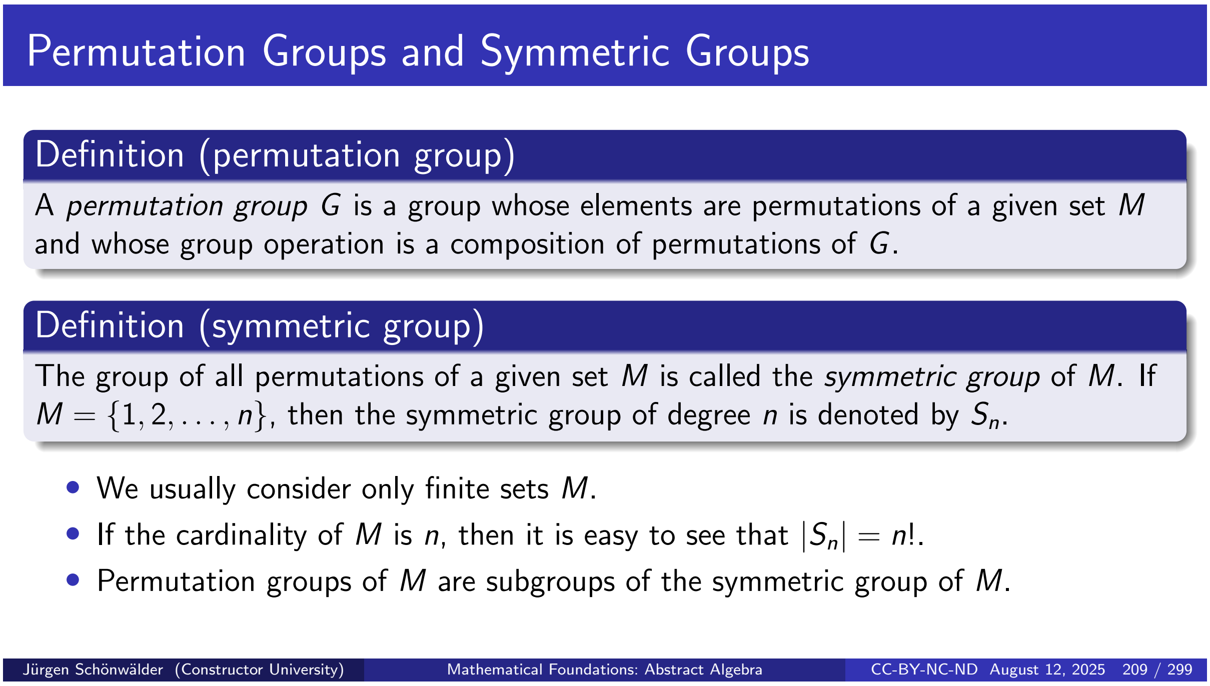
- Recall that the notation a^n means the group operation $\circ$ applied n times to a.
- For an additive group, a^n equals na.
- For a group representing symmetries, a^n means applying a symmetry operation (e.g., a certain rotation) n times.

Proof. We need to show that S' is closed under $\circ$ and under inverse.

Let $x, y \in S'$. This means that we can write $x = a^n$ and $y = a^m$ for some $n, m \in \mathbb{Z}$. It follows that $x \circ y = a^n \circ a^m = a^{n+m} \in S'$ since $n + m \in \mathbb{Z}$. Hence S' is closed under $\circ$.

Let $x \in S'$. We can assume $x = a^n$ for some $n \in \mathbb{Z}$. The inverse $x^{-1} = (a^n)^{-1} = a^{-n}$. Since $-n \in \mathbb{Z}$, the inverse of x is an element of S' and S' is closed under inverse.

Since S' is nonempty, we can apply the subgroup test theorem and hence $H = (S', \circ, e)$ is a subgroup of G. □



Permutation Groups and Symmetric Groups

Definition (permutation group)

A *permutation group* G is a group whose elements are permutations of a given set M and whose group operation is a composition of permutations of G.

Definition (symmetric group)

The group of all permutations of a given set M is called the *symmetric group* of M. If $M = \{1, 2, \ldots, n\}$, then the symmetric group of degree n is denoted by S_n.

- We usually consider only finite sets M.
- If the cardinality of M is n, then it is easy to see that $|S_n| = n!$.
- Permutation groups of M are subgroups of the symmetric group of M.

A permutation of a set M is a bijective function from M to M. The permutation function can be viewed as a rearrangement the elements of M. For a small finite set M, we can write the permutation by listing the initial order and the resulting order. For example, the permutations of the set $M = \{1, 2, 3\}$ are:

$$f_0 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \qquad f_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \qquad f_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

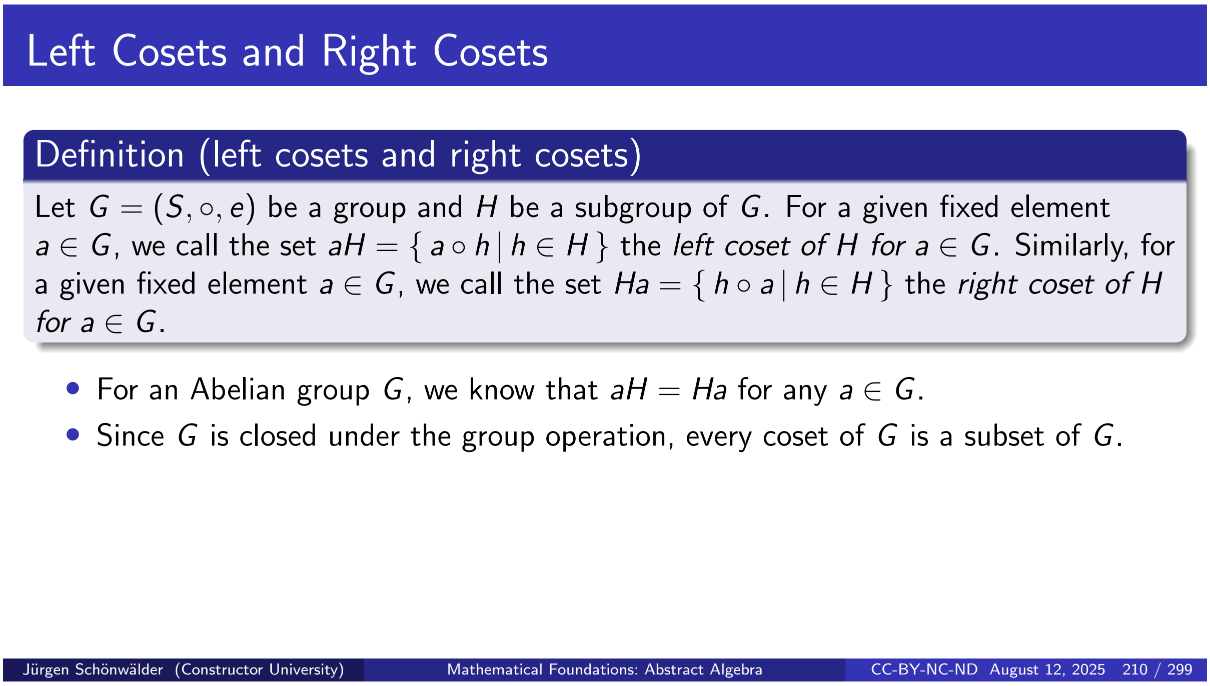
$$f_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \qquad f_4 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \qquad f_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

The permutation group is formed as $G = (\{f_0, f_1, f_2, f_3, f_4, f_5\}, \circ, f_0)$ where $\circ$ denotes function composition. Obviously, f_0 is the identity element since it does not change anything and f_1 is inverse to f_1. Furthermore, one can easily see that $f_2 \circ f_1 = f_5$.

Lets check why S_n is indeed a group. The elements of S_n are permutations $f_i : M \to M$ and permutations are bijective.

- Closure: Since the permutations f and g are bijections, so is the composition $f \circ g$.
- Associativity: Function composition is known to be associative, hence $f \circ (g \circ h) = (f \circ g) \circ h$.
- Identity: The permutation not changing any elements is the identity element.
- Inverses: Since a permutation f is bijective, an inverse permutation f^{-1} exists.

Example: The Dihedral group D_3 is structurally the same as the symmetric group S_3. Note that this is not generally true. The Dihedral Groups D_4 has eight elements while the symmetric group S_4 has $4! = 24$ elements.



Left Cosets and Right Cosets

Definition (left cosets and right cosets)

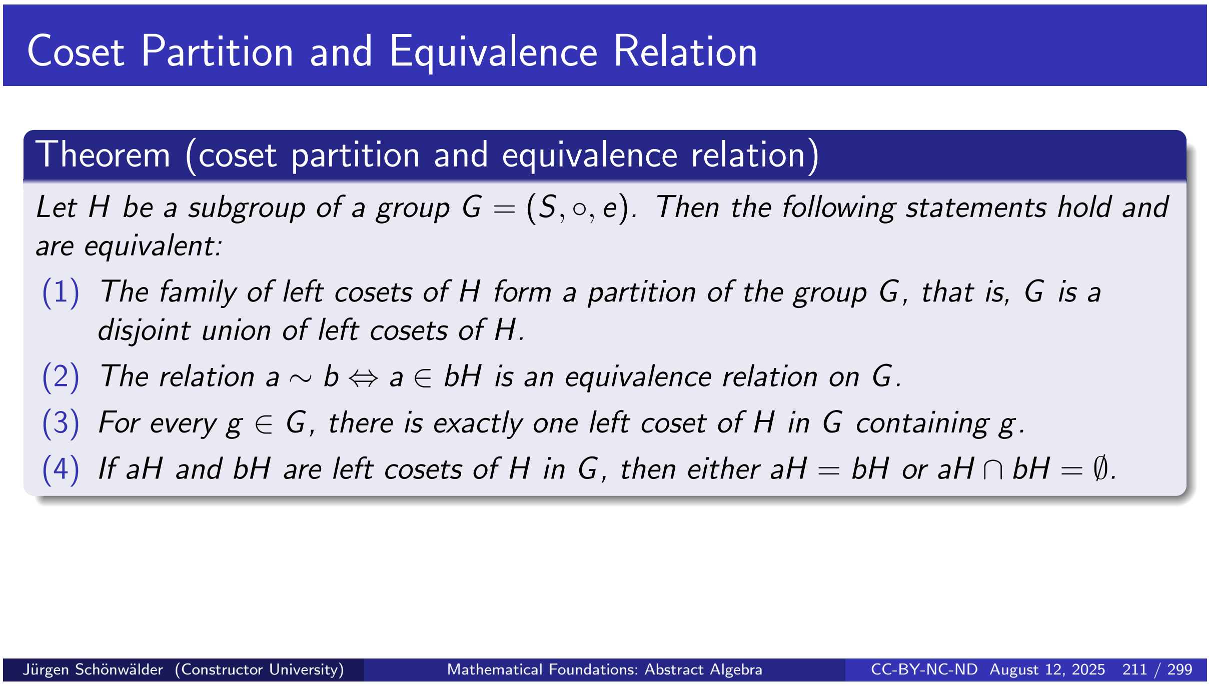
Let $G = (S, \circ, e)$ be a group and H be a subgroup of G. For a given fixed element $a \in G$, we call the set $aH = \{\, a \circ h \mid h \in H \,\}$ the *left coset of H for $a \in G$*. Similarly, for a given fixed element $a \in G$, we call the set $Ha = \{\, h \circ a \mid h \in H \,\}$ the *right coset of H for $a \in G$*.

- For an Abelian group G, we know that $aH = Ha$ for any $a \in G$.
- Since G is closed under the group operation, every coset of G is a subset of G.

Lets consider the group $G = (\mathbb{Z}_4, +, 0)$ where $+$ denotes addition modulo 4. We already know that G has the subgroup $H = (\{0, 2\}, +, 0)$. Lets consider all possible left cosets of H:

$$\begin{aligned}
0H &= \{0+0, 0+2\} = \{0, 2\}\\
1H &= \{1+0, 1+2\} = \{1, 3\}\\
2H &= \{2+0, 2+2\} = \{2, 0\}\\
3H &= \{3+0, 3+2\} = \{3, 1\}
\end{aligned}$$

Obviously, $0H = 2H = H$ and $1H = 3H$. Note that the cosets partition the set of the group into distinct subsets. We will see that this is generally true.



Coset Partition and Equivalence Relation

Theorem (coset partition and equivalence relation)

Let H be a subgroup of a group $G = (S, \circ, e)$. Then the following statements hold and are equivalent:

(1) *The family of left cosets of H form a partition of the group G, that is, G is a disjoint union of left cosets of H.*

(2) *The relation $a \sim b \Leftrightarrow a \in bH$ is an equivalence relation on G.*

(3) *For every $g \in G$, there is exactly one left coset of H in G containing g.*

(4) *If aH and bH are left cosets of H in G, then either $aH = bH$ or $aH \cap bH = \emptyset$.*

Proof. We have to show the equivalence of the four statements and we have to show that at least one of them is true.

Lets first show the equivalence of the statements. It is obvious that the first two statements are equivalent. Statement (1) follows from (2) by the definition of an equivalence relation. Conversely, statement (2) follows from statement (1) since a partition induces an equivalence relation. Statement (4) follows from (1) since two cosets of the partition are either the same or they are disjoint. Conversely, if all pairs of cosets are either the same or disjoint, the cosets form a partition, hence (1) also follows from (4). Finally, statement (1) implies statement (3) since every element of $g \in G$ can only be in one coset if the cosets form a partition. Conversely, if every element $g \in G$ is in exactly one coset, then the cosets must form a partition.

We show that statement (2) holds by showing that the relation $a \sim b \Leftrightarrow a \in bH$ is an equivalence relation, that is, it is reflexive, symmetric, and transitive.

- Reflexivity: For any $a \in G$ we have to show that $a \sim a$ holds.

  Since H is a subgroup, we know that $e \in H$. Hence, for any $a \in G$, we know that $a = a \circ e$ holds and hence $a \in aH$. Hence $a \sim a$ holds and the relation is reflexive.

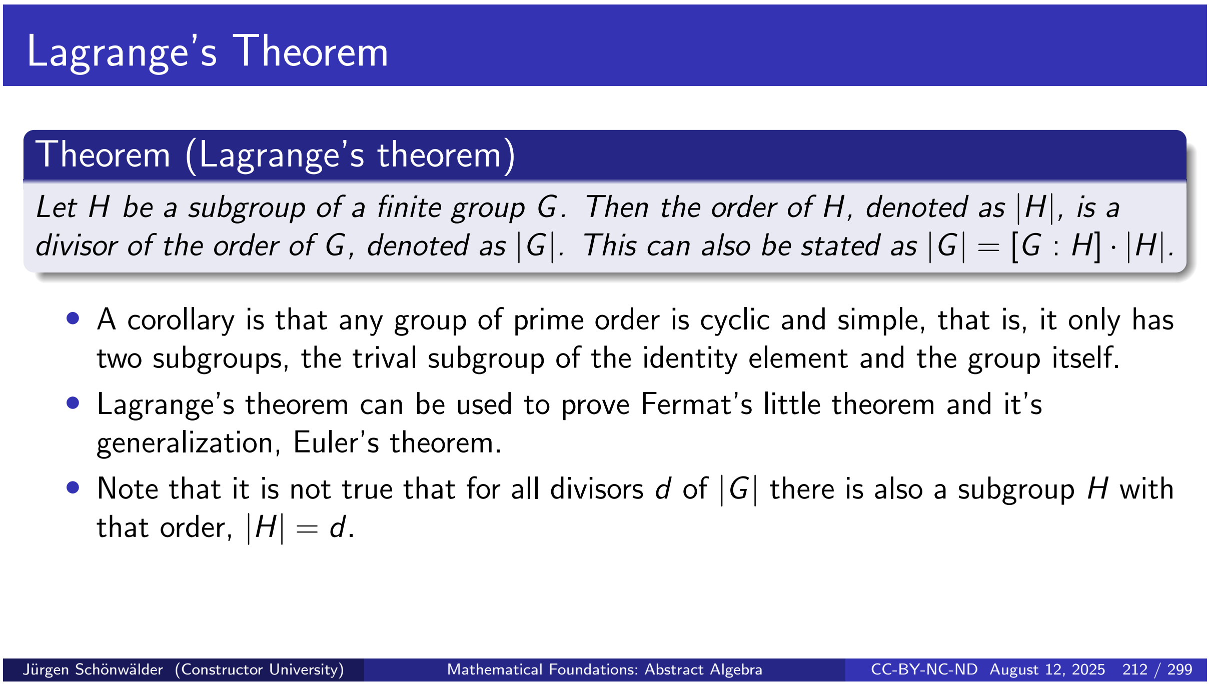
- Symmetry: For any $a, b \in G$ we have to show that $a \sim b$ implies $b \sim a$.

  If $a \sim b$, then $a = b \circ h$ for some $h \in H$. Since $h \in H$, there is also an inverse $h^{-1} \in H$. By right composing both sides of $a = b \circ h$ with h^{-1}, we get $b = a \circ h^{-1}$, which means $b \sim a$ and hence the relation is symmetric.

- Transitivity: For any $a, b, c \in G$, we have to show that if $a \sim b$ and $b \sim c$ implies $a \sim c$.

  $a \sim b$ means that $a = b \circ h_1$ for some $h_1 \in H$. Similarly, $b \sim c$ means that $b = c \circ h_2$ for some $h_2 \in H$. By substitution and associativity, we obtain $a = (c \circ h_2) \circ h_1 = c \circ (h_2 \circ h_1)$. Since $h_1, h_2 \in H$, we know that $(h_2 \circ h_1) \in H$. Hence, we have $a \sim c$ and the relation is transitive.

□



Lagrange's Theorem

Theorem (Lagrange's theorem)

Let H be a subgroup of a finite group G. Then the order of H, denoted as $|H|$, is a divisor of the order of G, denoted as $|G|$. This can also be stated as $|G| = [G : H] \cdot |H|$.

- A corollary is that any group of prime order is cyclic and simple, that is, it only has two subgroups, the trival subgroup of the identity element and the group itself.
- Lagrange's theorem can be used to prove Fermat's little theorem and it's generalization, Euler's theorem.
- Note that it is not true that for all divisors d of $|G|$ there is also a subgroup H with that order, $|H| = d$.

Proof. We already know that the left cosets of H form a partition of G. Hence, to prove Lagrange's theorem, we need to show that any coset aH of H has the same number of elements as H. This then implies that all cosets of H in G have the same number of elements and hence $|H|$ divides $|G|$.

To show that a coset aH has the same number of elements as H, we define function $f : H \to aH$ with $f(h) = a \circ h$ mapping from H to aH and we show that this function is a bijection, i.e., it is injective and surjective.

- Injective: We show that $f(h_1) = f(h_2)$ implies $h_1 = h_2$.

$$\begin{aligned} f(h_1) &= f(h_2) & \\ a \circ h_1 &= a \circ h_2 & \text{left compose with } a^{-1} \\ a^{-1} \circ (a \circ h1) &= a^{-1} \circ (a \circ h_2) & \text{associativity} \\ (a^{-1} \circ a) \circ h1 &= (a^{-1} \circ a) \circ h_2 & \\ h_1 &= h_2 & \end{aligned}$$

- Surjective: We show that for any element of aH there is an element in H mapping to it.

  Let $b \in aH$. This means that $b = a \circ h$ for some $h \in H$. Hence, we have found $f(h) = a \circ h$.

□

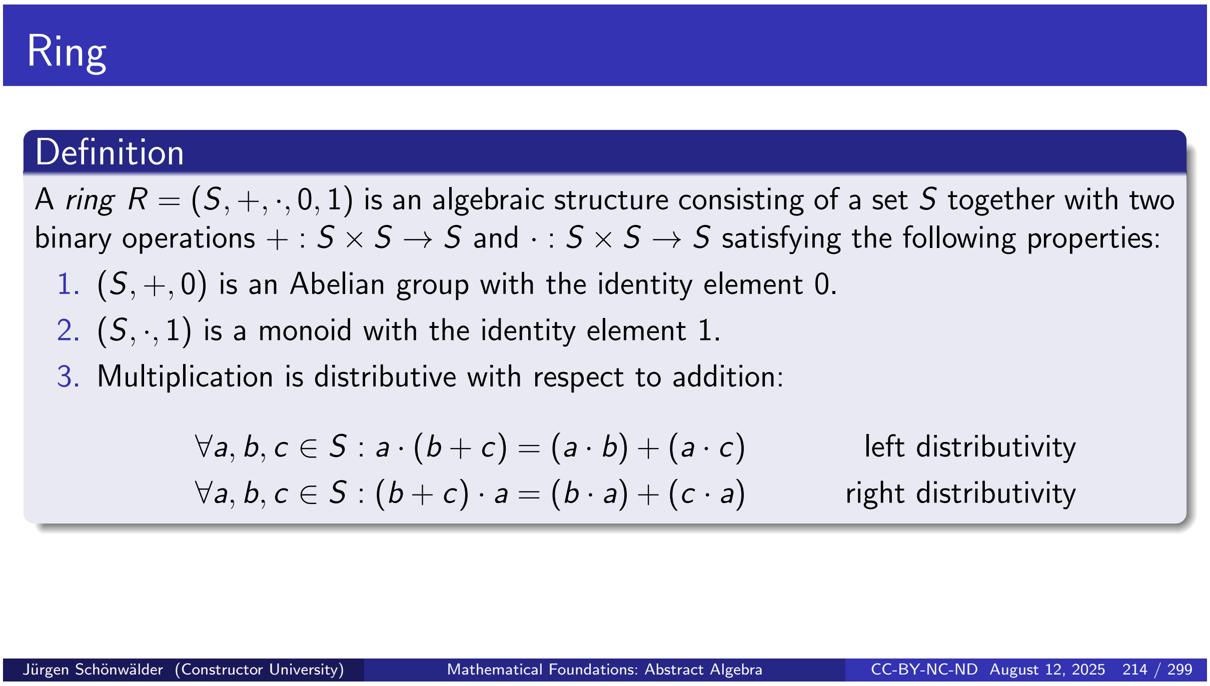
Section 24: Rings and Fields

23 Magmas, Semigroups, Monoids, Groups

24 Rings and Fields

25 Homomorphisms

26 Lattices



Ring

Definition

A *ring* $R = (S, +, \cdot, 0, 1)$ is an algebraic structure consisting of a set S together with two binary operations $+ : S \times S \to S$ and $\cdot : S \times S \to S$ satisfying the following properties:

1. $(S, +, 0)$ is an Abelian group with the identity element 0.
2. $(S, \cdot, 1)$ is a monoid with the identity element 1.
3. Multiplication is distributive with respect to addition:

$$\begin{array}{ll} \forall a, b, c \in S : a \cdot (b + c) = (a \cdot b) + (a \cdot c) & \text{left distributivity} \\ \forall a, b, c \in S : (b + c) \cdot a = (b \cdot a) + (c \cdot a) & \text{right distributivity} \end{array}$$

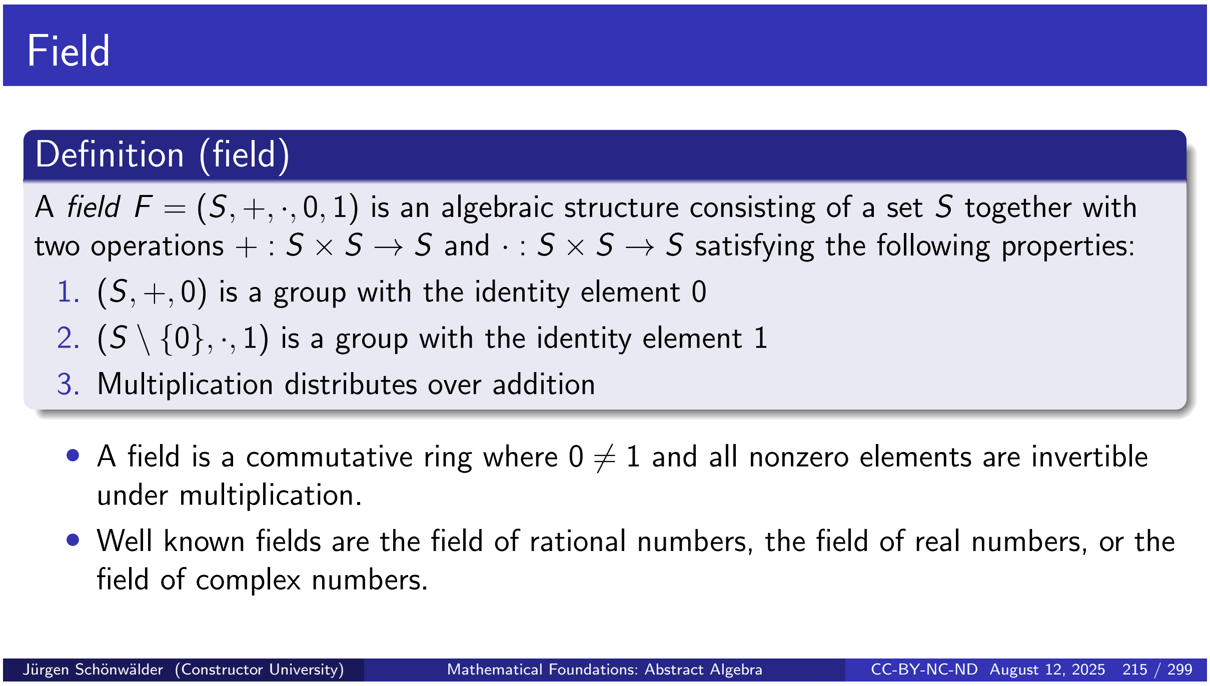
The operations are called $+$ and $\cdot$ and the neutral elements 0 and 1 since this makes it easier to think about rings. The names are, however, placeholders. The set S can have a complex structure and hence the neutral elements will also have a complex structure. In other words, the number 0 and the identity element 0 are two very separate concepts and the context tells you what is meant in a concrete situation. The same holds for the number 1 and the identity element 1.

Note that some authors define a ring differently, i.e., they only require $(S, \cdot)$ to be a semigroup.

Some examples of rings:

1. The algebraic structure $(\mathbb{Z}, +, \cdot, 0, 1)$ is a ring.
2. The finite sets of integer numbers $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ are rings for addition modulo n and multiplication modulo n.
3. The set of all 2×2 matrices is a ring for matrix addition and matrix multiplication operations. The zero elements are:

$$0 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \qquad 1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$



Field

Definition (field)

A *field* $F = (S, +, \cdot, 0, 1)$ is an algebraic structure consisting of a set S together with two operations $+ : S \times S \to S$ and $\cdot : S \times S \to S$ satisfying the following properties:

1. $(S, +, 0)$ is a group with the identity element 0
2. $(S \setminus \{0\}, \cdot, 1)$ is a group with the identity element 1
3. Multiplication distributes over addition

- A field is a commutative ring where $0 \neq 1$ and all nonzero elements are invertible under multiplication.
- Well known fields are the field of rational numbers, the field of real numbers, or the field of complex numbers.

Like before, operations $+$ and $\cdot$ are placeholders and the same is true for the identity elements 0 and 1.

Some examples of fields:

1. $(\mathbb{Q}, +, \cdot, 0, 1)$ (addition and multiplication over rational numbers)
2. $(\mathbb{R}, +, \cdot, 0, 1)$ (addition and multiplication over real numbers)
3. $(\mathbb{Z}_p, +, \cdot, 0, 1)$ (addition and multiplication modulo p over $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ with p prime)

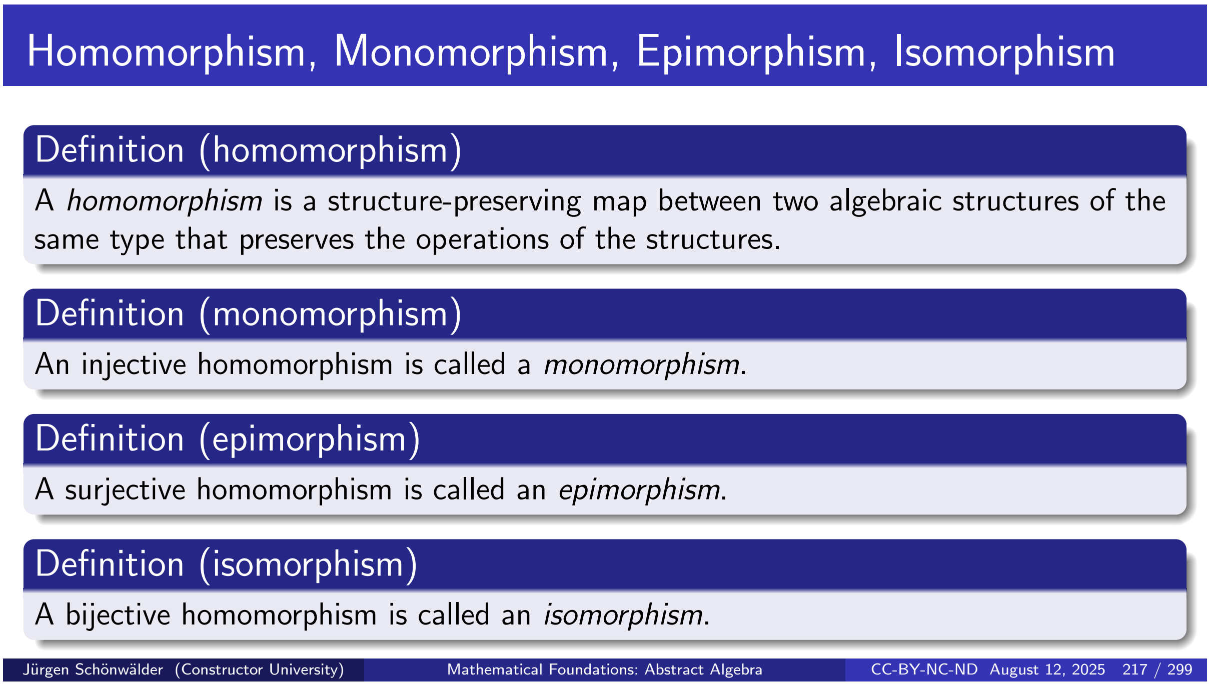
Section 25: Homomorphisms

23 Magmas, Semigroups, Monoids, Groups

24 Rings and Fields

25 Homomorphisms

26 Lattices



Homomorphism, Monomorphism, Epimorphism, Isomorphism

Definition (homomorphism)

A *homomorphism* is a structure-preserving map between two algebraic structures of the same type that preserves the operations of the structures.

Definition (monomorphism)

An injective homomorphism is called a *monomorphism*.

Definition (epimorphism)

A surjective homomorphism is called an *epimorphism*.

Definition (isomorphism)

A bijective homomorphism is called an *isomorphism*.

Given two groups, $G = (S_G, *, e_G)$ and $H = (S_H, \star, e_H)$, a *group homomorphism* from G to H is a mapping $h : G \to H$ such that for all u and v in G the following holds:

$$h(u * v) = h(u) \star h(v)$$

Note that the group operation on the left side of the equation is that of G and on the right side that of H. The group homomorphism h maps the identity element e_G of G to the identity element e_H of H and it also maps the inverses of G to inverses of H:

$$h(u^{-1}) = (h(u))^{-1}$$

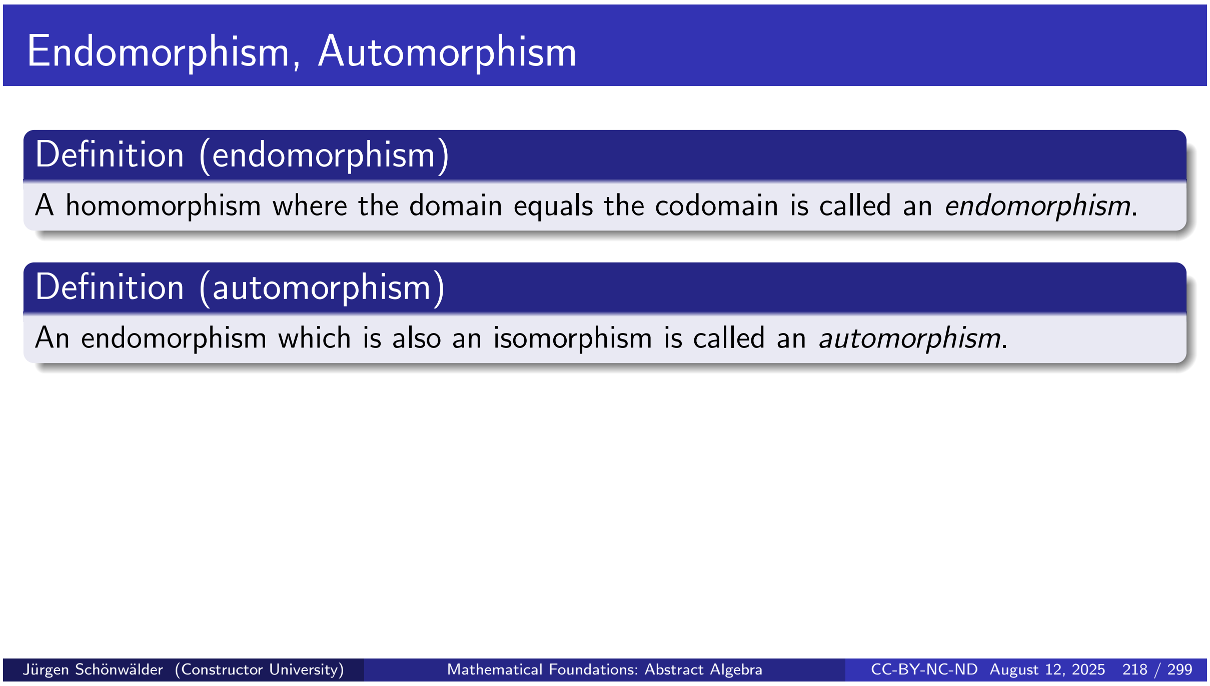
Example: Given the groups $G = (\mathbb{R}, +, 0)$ and $H = (\mathbb{R}, \cdot, 1)$, the map $h : \mathbb{R} \to \mathbb{R}$ with $x \mapsto 2^x$ is a group homomorphism.

$$\begin{aligned} h(u + v) &= 2^{(u+v)} \\ &= 2^u \cdot 2^v \\ &= h(u) \cdot h(v) \end{aligned}$$

Note that h maps the identity element 0 of G to the identity element 1 of H and that it also maps inverses:

$$\begin{aligned} h(u^{-1}) &= h(-u) \\ &= 2^{-u} \\ &= (2^u)^{-1} \\ &= (h(u))^{-1} \end{aligned}$$

Let G_1 and G_2 be groups. A bijective homomorphism $f : G_1 \to G_2$ is called an isomorphism from G_1 to G_2. If such a homomorphism exist from G_1 to G_2, we call G_1 isomorphic to G_2. Isomorphic groups essentially have the same form even though the sets of the groups may be very different.



Endomorphism, Automorphism

Definition (endomorphism)

A homomorphism where the domain equals the codomain is called an *endomorphism*.

Definition (automorphism)

An endomorphism which is also an isomorphism is called an *automorphism*.

Example: Lets show that $G = (\mathbb{R}, +, 0)$ is isomorphic to $H = (\mathbb{R}^+, \cdot, 1)$. To show this, we need to find an isomorphism mapping from G to H. Lets try with the function $f : \mathbb{R} \to \mathbb{R}^+$ with $x \mapsto e^x$.

- f is injective: Let $x, y \in \mathbb{R}$ with $f(x) = f(y)$. We show that this implies that $x = y$:

$$\begin{aligned} f(x) &= f(y) \\ e^x &= e^y \\ ln(e^x) &= ln(e^y) \\ x &= y \end{aligned}$$

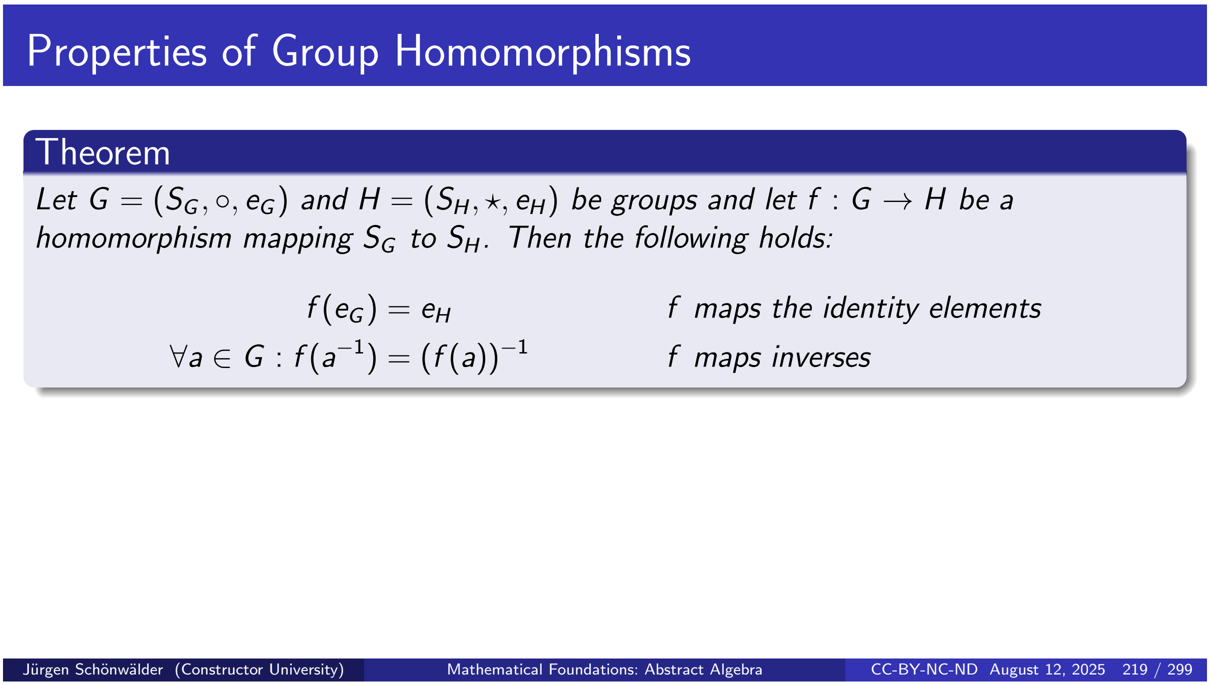
- f is surjective: Let $y \in \mathbb{R}^+$ be arbitrary. We show that there will be an $x \in \mathbb{R}$ such that $y = f(x)$. For an arbitrary y, lets take $x = ln(y)$:

$$\begin{aligned} f(x) &= f(ln(y)) \\ &= e^{ln(y)} \\ &= y \end{aligned}$$

- f is a homomorphism: Let $x, y \in \mathbb{R}$. We have to show that $f(x + y) = f(x) \cdot f(y)$:

$$\begin{aligned} f(x+y) &= e^{x+y} \\ &= e^x \cdot e^y \\ &= f(x) \cdot f(y) \end{aligned}$$

Note that there can be other isomorphisms. To show that two groups are isomorphic, it is sufficient to find one of possibly many isomorphisms.



Properties of Group Homomorphisms

Theorem

Let $G = (S_G, \circ, e_G)$ and $H = (S_H, \star, e_H)$ be groups and let $f : G \to H$ be a homomorphism mapping S_G to S_H. Then the following holds:

$$\begin{array}{rl} f(e_G) = e_H & \textit{f maps the identity elements} \\ \forall a \in G : f(a^{-1}) = (f(a))^{-1} & \textit{f maps inverses} \end{array}$$

Proof. We prove the two statements separately.

- We first prove $f(e_G) = e_H$. Lets pick an arbitrary $a \in G$. Since f is a homomorphims, the following must hold:

  $$f(a \circ e_G) = f(a) \star f(e_G)$$

  In addition, we know that $f(a \circ e_G) = f(a)$ since e_G is the identity of G. Hence we get:

  $$f(a) = f(a) \star f(e_G)$$

  Since H is a group, we know that the inverse of $f(a)$ exists. Hence, we can left multiply the above with $(f(a))^{-1}$:

  $$(f(a))^{-1} \star f(a) = (f(a))^{-1} \star f(a) \star f(e_G)$$

  Due to associativity, this can be reduced to the following:

  $$e_H = e_H \star f(e_G) = f(e_G)$$

- We now prove $f(a^{-1}) = (f(a))^{-1}$ for all $a \in G$. Lets pick an arbitrary $a \in G$. Since G is a group, a has the inverse a^{-1}. Since f is a homomorphism, we know that the following holds:

  $$f(a) \star f(a^{-1}) = f(a \circ a^{-1}) = f(e_G) = e_H$$

  Since H is a group, we know that the inverse of $f(a)$ exists. Hence, we can left multiply the above with $(f(a))^{-1}$:

  $$(f(a))^{-1} \star f(a) \star f(a^{-1}) = (f(a))^{-1} \star e_H$$

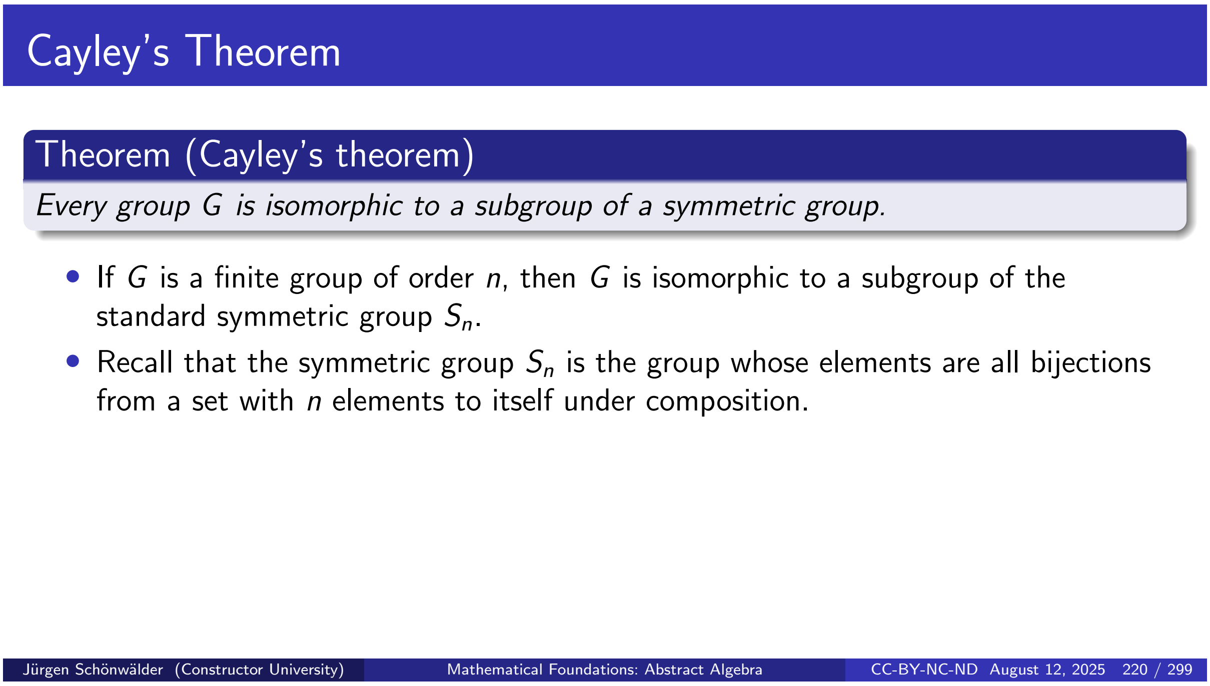
  This reduces to

  $$e_H \star f(a^{-1}) = (f(a))^{-1} \star e_H$$

  and finally we get:

  $$f(a^{-1}) = (f(a))^{-1}$$

□



Cayley's Theorem

Theorem (Cayley's theorem)

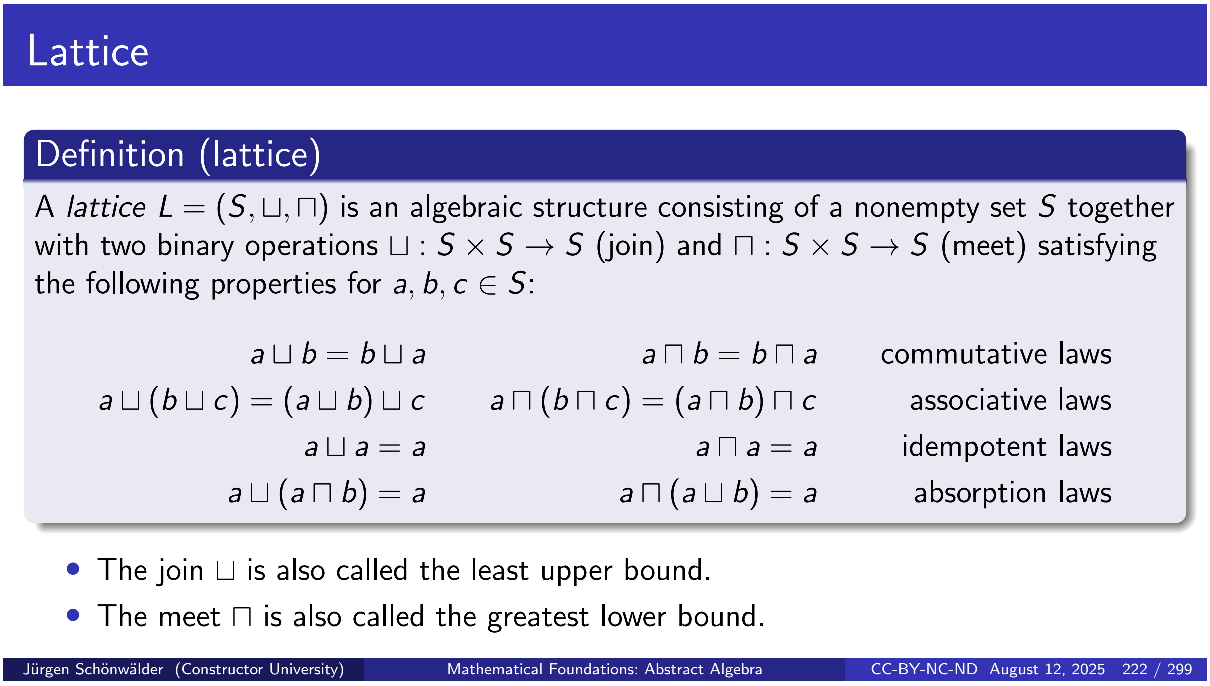
Every group G is isomorphic to a subgroup of a symmetric group.

- If G is a finite group of order n, then G is isomorphic to a subgroup of the standard symmetric group S_n.
- Recall that the symmetric group S_n is the group whose elements are all bijections from a set with n elements to itself under composition.

Further online information:

- **Wikipedia**: Cayley's theorem

Section 26: Lattices



Lattice

Definition (lattice)

A *lattice* $L = (S, \sqcup, \sqcap)$ is an algebraic structure consisting of a nonempty set S together with two binary operations $\sqcup : S \times S \to S$ (join) and $\sqcap : S \times S \to S$ (meet) satisfying the following properties for $a, b, c \in S$:

$$\begin{array}{rrr}
a \sqcup b = b \sqcup a & a \sqcap b = b \sqcap a & \text{commutative laws} \\
a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c & a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c & \text{associative laws} \\
a \sqcup a = a & a \sqcap a = a & \text{idempotent laws} \\
a \sqcup (a \sqcap b) = a & a \sqcap (a \sqcup b) = a & \text{absorption laws}
\end{array}$$

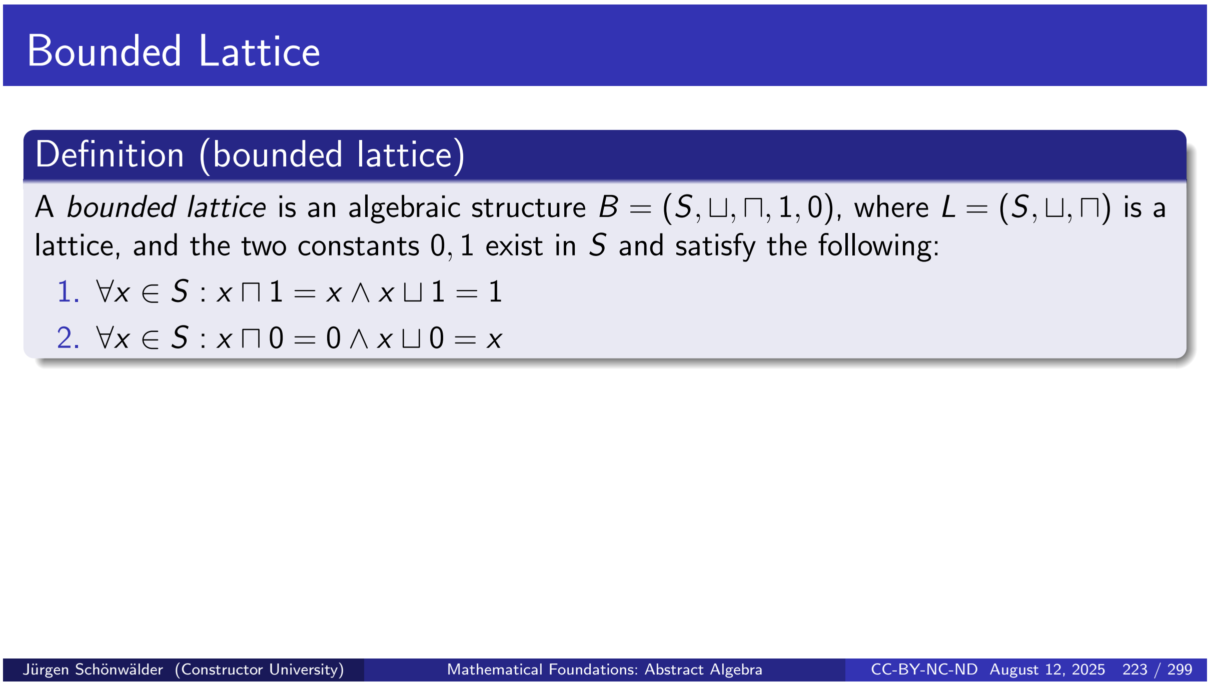
- The join $\sqcup$ is also called the least upper bound.
- The meet $\sqcap$ is also called the greatest lower bound.

Some examples of lattices:

1. $L = (\mathbb{N}, \sqcup, \sqcap)$ is a lattice over the natural numbers where $\sqcup$ is the least common multiple of two numbers and $\sqcap$ is the greatest common divisor of two numbers.
2. $L = (\mathbb{B}, \vee, \wedge)$ is a lattice we already know from Boolean algebra.
3. The set of all subsets of a given set forms a lattice.

Further online information:

- **Wikipedia**: Lattice (order)



Bounded Lattice

Definition (bounded lattice)

A *bounded lattice* is an algebraic structure $B = (S, \sqcup, \sqcap, 1, 0)$, where $L = (S, \sqcup, \sqcap)$ is a lattice, and the two constants $0, 1$ exist in S and satisfy the following:

1. $\forall x \in S : x \sqcap 1 = x \land x \sqcup 1 = 1$
2. $\forall x \in S : x \sqcap 0 = 0 \land x \sqcup 0 = x$

Boolean algebra is a bounded lattice: $B = (\mathbb{B}, \lor, \land, 0, 1)$.

Because the meet and join operations both commute and associate, a lattice can be viewed as consisting of two commutative semigroups having the same domain. For a bounded lattice, these semigroups are in fact commutative monoids. The

Lattice and Partial Orders

Theorem

A nonempty partially ordered set $(S, \preceq)$ forms the lattice $L = (S, \sup, \inf)$ if for every $a, b \in S$ both the suprenum $\sup(a, b)$ and the infimum $\inf(a, b)$ exist.

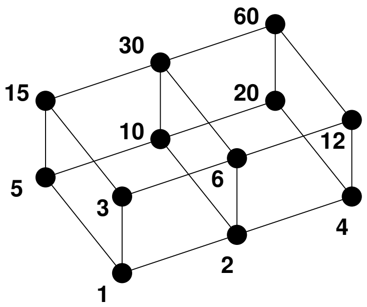
- Recall that a binary relation is a partial order if it is
  - reflexive,
  - antisymmetric, and
  - transitive.
- The suprenum $\sup(a, b)$ of $a, b \in S$ is the smallest upper bound of a and b.
- The infimum $\inf(a, b)$ of $a, b \in S$ is the greatest lower bound of a and b.
- Some authors define lattices via partially ordered sets.

Proof. Lets assume we have a partially ordered set S such that for all $a, b \in S$ the suprenum $\sup(a, b)$ and the infimum $\inf(a, b)$ exist in S. We show that this gives us the lattice $L = (S, \sup, \inf)$ by verifying the properties of a lattice:

- Commutative laws: Apparently $\inf(a, b) = \inf(b, a)$ and $\sup(a, b) = \sup(b, a)$ since the order of the elements does not matter for the infimum and the suprenum.
- Associative laws: It is easy to see that $\inf(\inf(a, b), c) = \inf(a, \inf(b, c))$ and $\sup(\sup(a, b), c) = \sup(a, \sup(b, c))$ hold.
- Idempotent laws: Apparently $a = \inf(a, a)$ and $a = \sup(a, a)$.
- Absorption laws: $a = \sup(a, \inf(a, b))$ is true since $\inf(a, b)$ is less than or equal to a. Similarly, $a = \inf(a, \sup(a, b))$ is true since $\sup(a, b)$ is greater or equal to a.

□

Hasse diagrams are a way to visualize finite partially ordered sets. Let S be a set and $\preceq$ a partial order on S. Construct a graph where all elements of S are represented as nodes. A link between a and b denotes that $a \prec b$ and there is no c such that $a \prec c \prec b$. Furthermore, if $a \prec b$, then b is placed above a.



The Hasse diagram above shows the partially ordered set $\{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$ with the order relation *divides by*. This is actually a lattice since all pairs of elements have a suprenum (a join) and an infimum (a meet).

Further online information:

- **Wikipedia**: Hasse diagram

Distributive Lattice

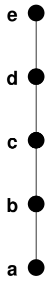
Definition (distributive lattice)

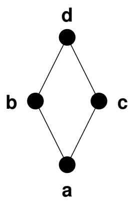
A *distributive lattice* is a lattice $L = (S, \sqcup, \sqcap)$ which satisfies the distributive laws for $a, b, c \in S$:

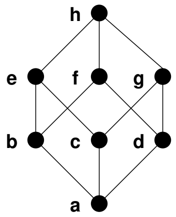
$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$
$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$

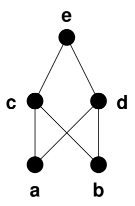
- It can be shown that one distributive law implies the other.

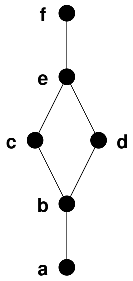
With Hasse diagrams, it is relatively easy to check visually whether a resonably small finite partially ordered set is a lattice or not: In a lattice, all pairs of "unrelated" nodes must have a suprenum and an infimum, i.e., a unique node "above" the pair of nodes where edges join and a unique node "below" the pair of nodes where edges meet.

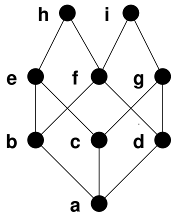


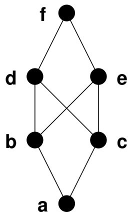












Some of the the Hasse diagrams above are lattices, some are not. Which ones are not lattices and why?

Complements and Distributive Lattices

Definition

The elements x, y of a bounded lattice $B = (S, \sqcup, \sqcap, 1, 0)$ are complements if $x \sqcup y = 1$ and $x \sqcap y = 0$.

Theorem

In a bounded distributive lattice, every element has at most one complement.

- This theorem makes it relatively easy the decide whether a given lattice is distributive.

Part VII

Graphs and Graph Algorithms

This part introduces basic elements of graph theory. Many classic computer science problems, such as finding shortest paths, creating spanning trees, calculating maximum flows, can be formulated as problems on graphs. Trees are of special importance in computer science since trees enable efficient algorithms for finding information.

By the end of this part, students should be able to

- distinguish simple graphs, multigraphs, directed graphs, and directed multigraphs;
- recognize walks, trails, paths, closed walks, and cycles;
- identify components and determine whether a graph is connected;
- split graphs into subgraphs and components;
- create graphs using disjoint unions;
- transform graphs simple graphs, directed graphs, and multigraphs into directed multigraphs
- represent graphs using adjacency list and adjacency matrices;
- recall and use basic properties of trees;
- outline and execute graph traversal algorithms;
- describe and solve maximum flow problems.

Section 27: Graphs and Multigraphs

27 Graphs and Multigraphs

28 Directed Graphs and Directed Multigraphs

29 Trees

30 Graph Traversals

31 Shortest Paths

32 Maximum Flows

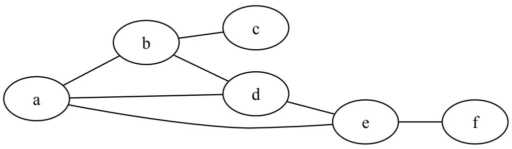
Simple Graphs

Definition (simple graphs)

A *simple graph* is a pair $G = (V, E)$, where V is a set of *vertices*, and E is a set of unordered pairs $\{u, v\}$ of *edges* with $u, v \in V$ and $u \neq v$. The vertices u and v of an edge $\{u, v\}$ are called the edge's *endpoints*. When an edge $\{u, v\}$ exists, then the vertices u and v are called *adjacent*. The set of vertices adjacent to a given vertex v are called the *neighbors* of v.

- A simple graph has no links connecting a vertex to itself (so called self-loops).
- There can only be a single edge connecting two vertices in a simple graph.
- Sometimes it is convenient to restrict the vertices to finite sets.

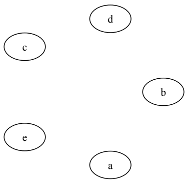
Consider the simple graph below. Mathematically, the graph is defined by $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ and $E = \{\{a, b\}, \{a, d\}, \{a, e\}, \{b, c\}, \{b, d\}, \{d, e\}, \{e, f\}\}$.



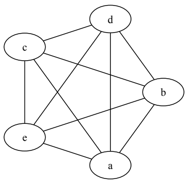
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Computer programs use different techniques to represent graphs internally. Two common approaches are adjacency lists and adjacency matrices: In the first approach, a program stores for each vertex the set of adjacent vertices, e.g. $a \mapsto \{b, d, e\}, b \mapsto \{a, c, d\}, c \mapsto \{b\}, d \mapsto \{a, b, e\}, e \mapsto \{d, f\}, f \mapsto \{e\}$. The other approach is to create a matrix with a row and a column for each vertex. The value in a cell is 1 if there is an edge between two vertices and 0 otherwise. For a simple graph, the diagonal will always be zero. For the graph G, we have the adjacency matrix A.

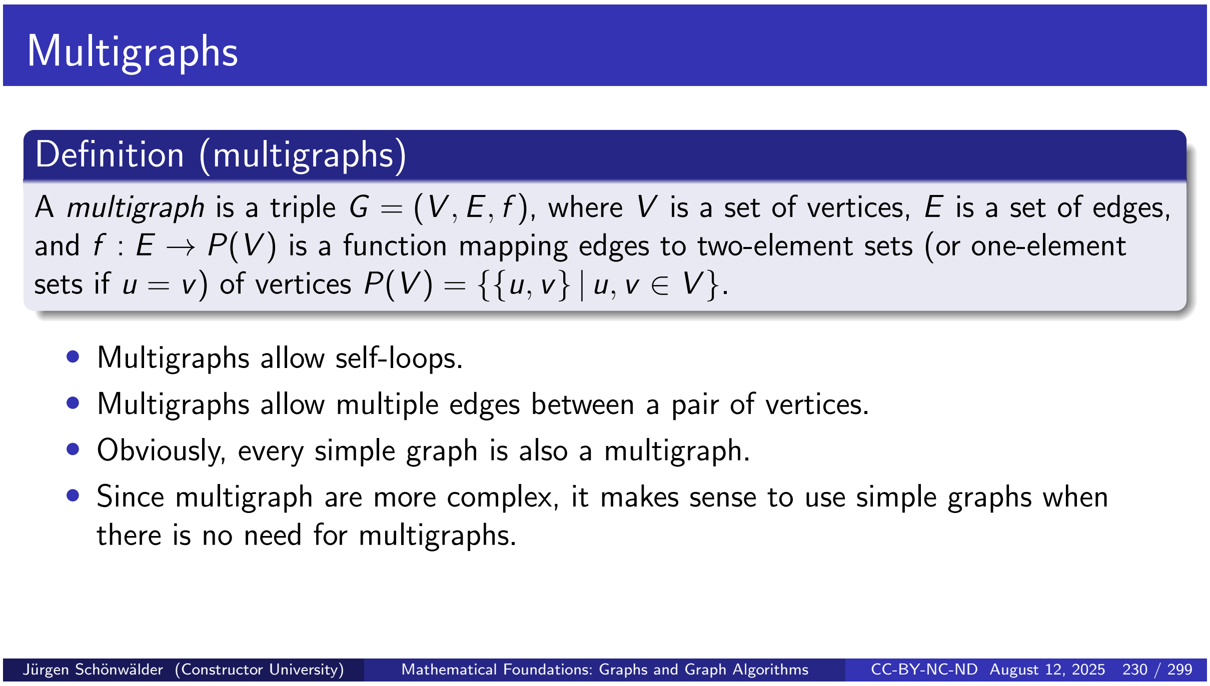
An graph without any edges, $G = (V, \emptyset)$, is called an empty graph. Note that an empty graph still has a set of vertices. A simple graph $G = (V, E)$, where E includes all possible edges of V, is called a complete graph. Complete graphs with $n = |V|$ vertices are called K_n (since all complete graphs with n vertices are isomorphic to each other).



$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$



Multigraphs

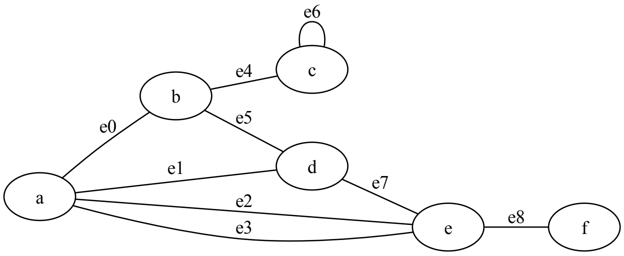
Definition (multigraphs)

A *multigraph* is a triple $G = (V, E, f)$, where V is a set of vertices, E is a set of edges, and $f : E \to P(V)$ is a function mapping edges to two-element sets (or one-element sets if $u = v$) of vertices $P(V) = \{\{u, v\} \mid u, v \in V\}$.

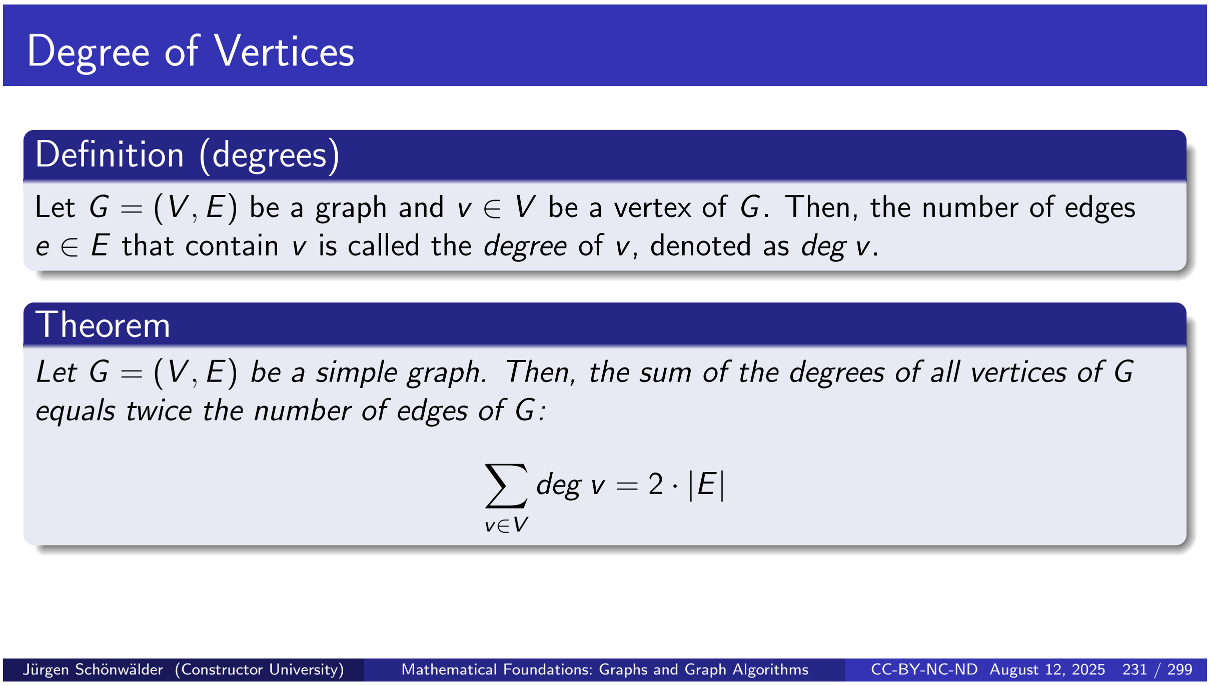
- Multigraphs allow self-loops.
- Multigraphs allow multiple edges between a pair of vertices.
- Obviously, every simple graph is also a multigraph.
- Since multigraph are more complex, it makes sense to use simple graphs when there is no need for multigraphs.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Graphs and Graph Algorithms CC-BY-NC-ND August 12, 2025 230 / 299

The following graph is a multigraph since it contains a self-loop and two edges between a and e. Note that the representation using adjacency matrices can be extended to represent multigraphs as well.



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 2 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



Degree of Vertices

Definition (degrees)

Let $G = (V, E)$ be a graph and $v \in V$ be a vertex of G. Then, the number of edges $e \in E$ that contain v is called the *degree* of v, denoted as *deg* v.

Theorem

Let $G = (V, E)$ be a simple graph. Then, the sum of the degrees of all vertices of G equals twice the number of edges of G:

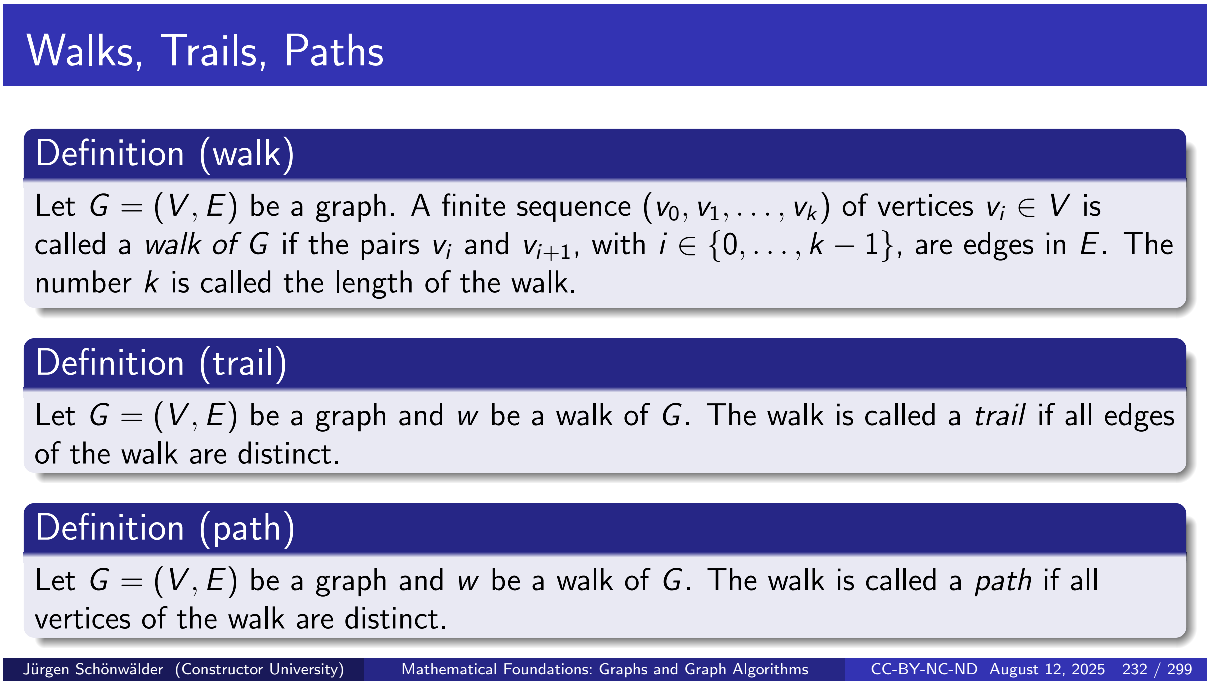
$$\sum_{v \in V} deg\ v = 2 \cdot |E|$$

Proof. We can proof this via induction over the number of edges. Let G be a graph. If G is empty, then the degree of all vertices is 0 and hence $\sum_{v \in V} 0 = 2 \cdot 0$ holds. Lets assume that $\sum_{v \in V} deg\, v = 2 \cdot |E|$ holds for $|E| = n$ edges. If we add an additional edge to E, then the size of E increases to $n + 1$. Since the new edge has two endpoints, the degree of these two endpoints will increase by one each. Since we add two on the left side of the equation and two to the right side of the equation, it still holds for $|E| = n + 1$. □

In many natural large graphs, the degree distribution, that is the fraction of nodes in the network with a degree K, follows certain pattern where a large majority of nodes have low degree but a small number have high degree. This tends to make the connectivity of graphs robust against single independent random failures. In some graphs, it was noted that their degree distribution follows a power law. These graphs got known as scale-free networks. Scale-free networks have a number of interesting properties.

Further online information:

- **Wikipedia**: Scale-free network



Walks, Trails, Paths

Definition (walk)

Let $G = (V, E)$ be a graph. A finite sequence $(v_0, v_1, \dots, v_k)$ of vertices $v_i \in V$ is called a *walk of G* if the pairs v_i and v_{i+1}, with $i \in \{0, \dots, k-1\}$, are edges in E. The number k is called the length of the walk.

Definition (trail)

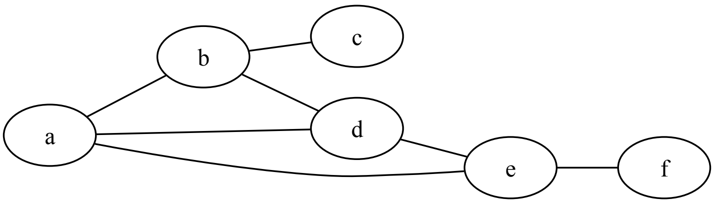
Let $G = (V, E)$ be a graph and w be a walk of G. The walk is called a *trail* if all edges of the walk are distinct.

Definition (path)

Let $G = (V, E)$ be a graph and w be a walk of G. The walk is called a *path* if all vertices of the walk are distinct.

A walk does not put any constraints on the vertices visited and the edges crossed. It is perfectly fine to walk in circles. A trail requires that edges are only crossed once. Of course, it is not required to cross all edges of a graph or to visit all vertices. A path requires that vertices are only crossed once. This implies that also edges are only crossed once. Hence, a path is a special kind of a trail.

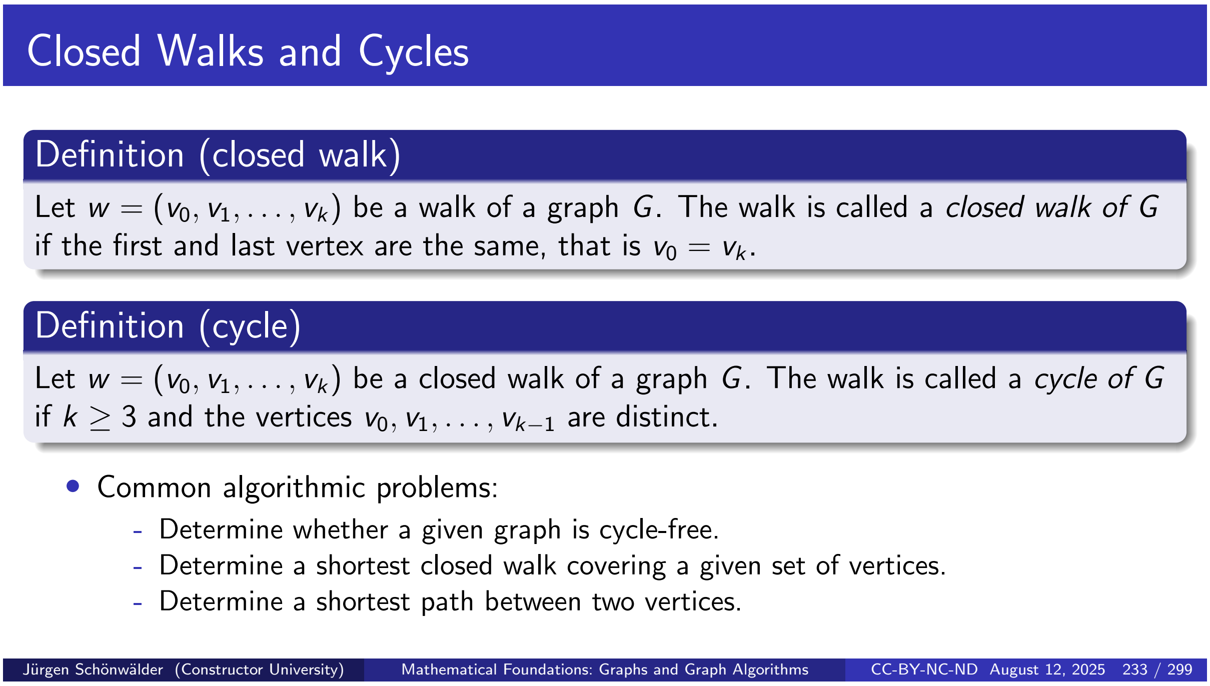
Lets consider the following simple graph:



The walk (a, b, d, a, b, c) is neither a trail nor a path. The walk (a, d, e, a, b, c) is a trail but not a path. The walk (a, e, d, b, c) is a path and hence also a trail. The sequence (a, d, e, c) is not a walk.

We can define simple operations on walks:

- Two walks can be concatenated into a single walk if the first walk ends at a vertex that is the beginning of the second walk.
- A walk can be split into two shorter walks.
- A walk can be reversed by reversing the vertices in the sequence.
- A walk can be reduced to a path by removing any loops from the sequence.



Closed Walks and Cycles

Definition (closed walk)

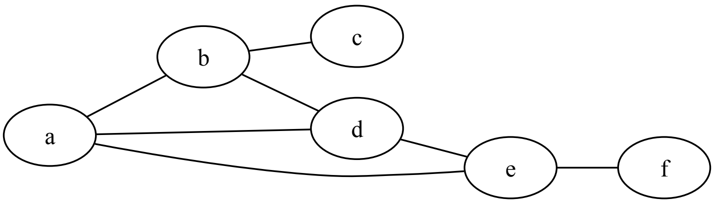
Let $w = (v_0, v_1, \ldots, v_k)$ be a walk of a graph G. The walk is called a *closed walk of G* if the first and last vertex are the same, that is $v_0 = v_k$.

Definition (cycle)

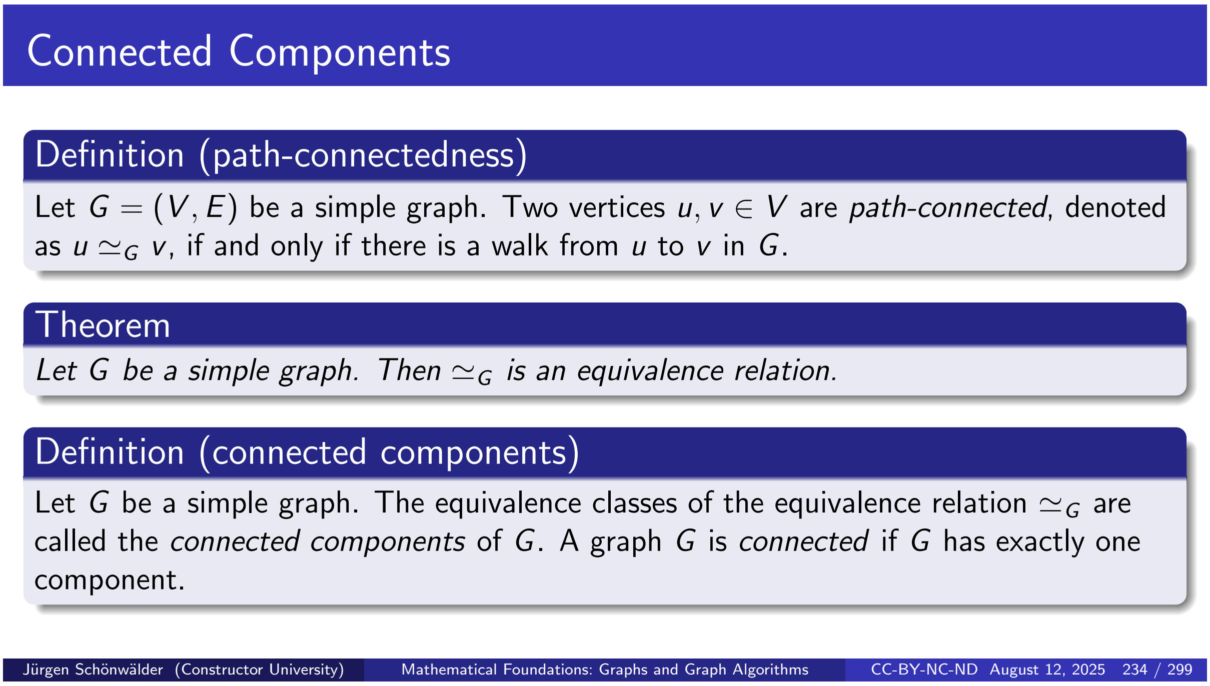
Let $w = (v_0, v_1, \ldots, v_k)$ be a closed walk of a graph G. The walk is called a *cycle of G* if $k \geq 3$ and the vertices $v_0, v_1, \ldots, v_{k-1}$ are distinct.

- Common algorithmic problems:
  - Determine whether a given graph is cycle-free.
  - Determine a shortest closed walk covering a given set of vertices.
  - Determine a shortest path between two vertices.

Lets consider the following simple graph:



The sequence (e, d, a, b, d, e) is a closed walk but not a cycle. The walks (a, b, d, a) and (a, d, e, a) and (a, b, d, e, a) are all cycles.



Connected Components

Definition (path-connectedness)

Let $G = (V, E)$ be a simple graph. Two vertices $u, v \in V$ are *path-connected*, denoted as $u \simeq_G v$, if and only if there is a walk from u to v in G.

Theorem

Let G be a simple graph. Then $\simeq_G$ is an equivalence relation.

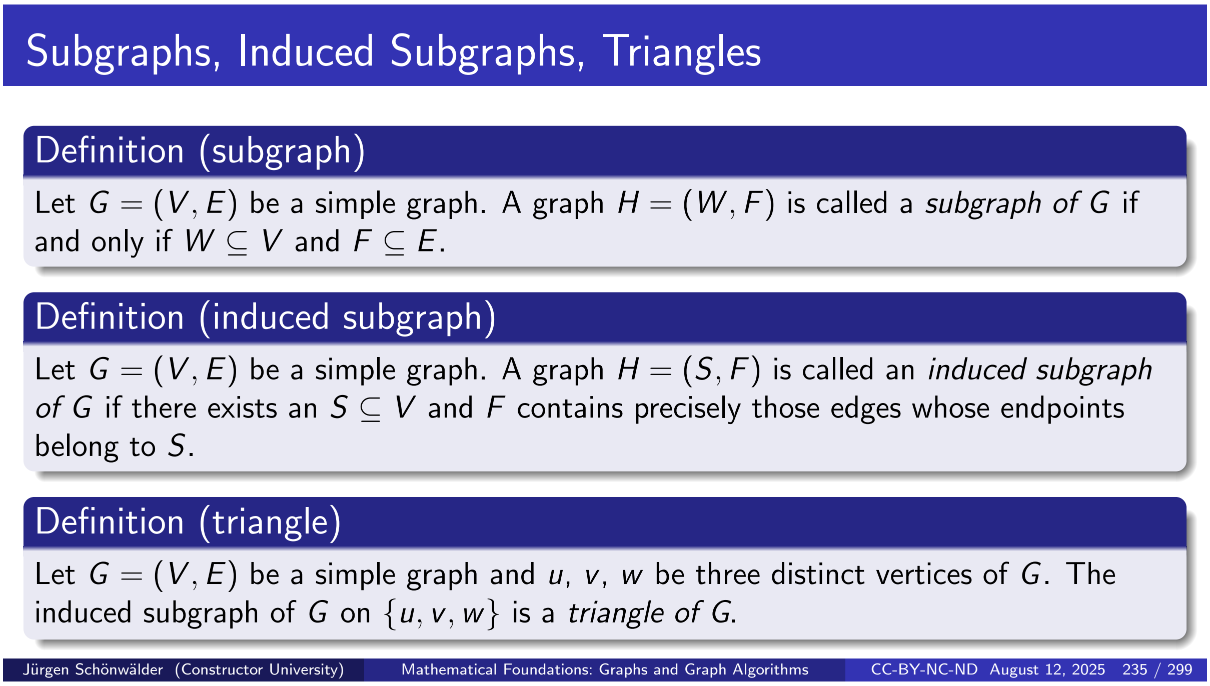
Definition (connected components)

Let G be a simple graph. The equivalence classes of the equivalence relation $\simeq_G$ are called the *connected components* of G. A graph G is *connected* if G has exactly one component.

Proof. We need to show that $\simeq_G$ is symmetric, reflexive, and transitive:

- Symmetry: If $u \simeq_G v$, the $v \simeq_G u$ since we can reverse the walk from u to v.
- Reflexivity: Since the trivial walk (u) is a walk from u to u, we know that $u \simeq_G u$.
- Transitivity: If $u \simeq_G v$ and $v \simeq_G w$, then $u \simeq_G w$ since we can concatenate the walk from u to v with the walk from v to w.

□



Subgraphs, Induced Subgraphs, Triangles

Definition (subgraph)

Let $G = (V, E)$ be a simple graph. A graph $H = (W, F)$ is called a *subgraph of G* if and only if $W \subseteq V$ and $F \subseteq E$.

Definition (induced subgraph)

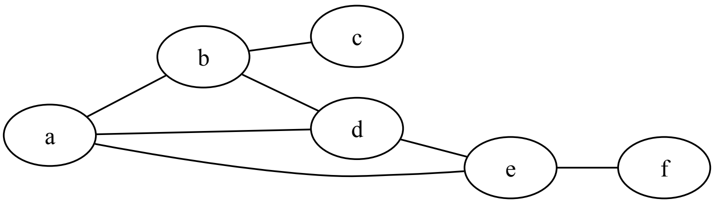
Let $G = (V, E)$ be a simple graph. A graph $H = (S, F)$ is called an *induced subgraph of G* if there exists an $S \subseteq V$ and F contains precisely those edges whose endpoints belong to S.

Definition (triangle)

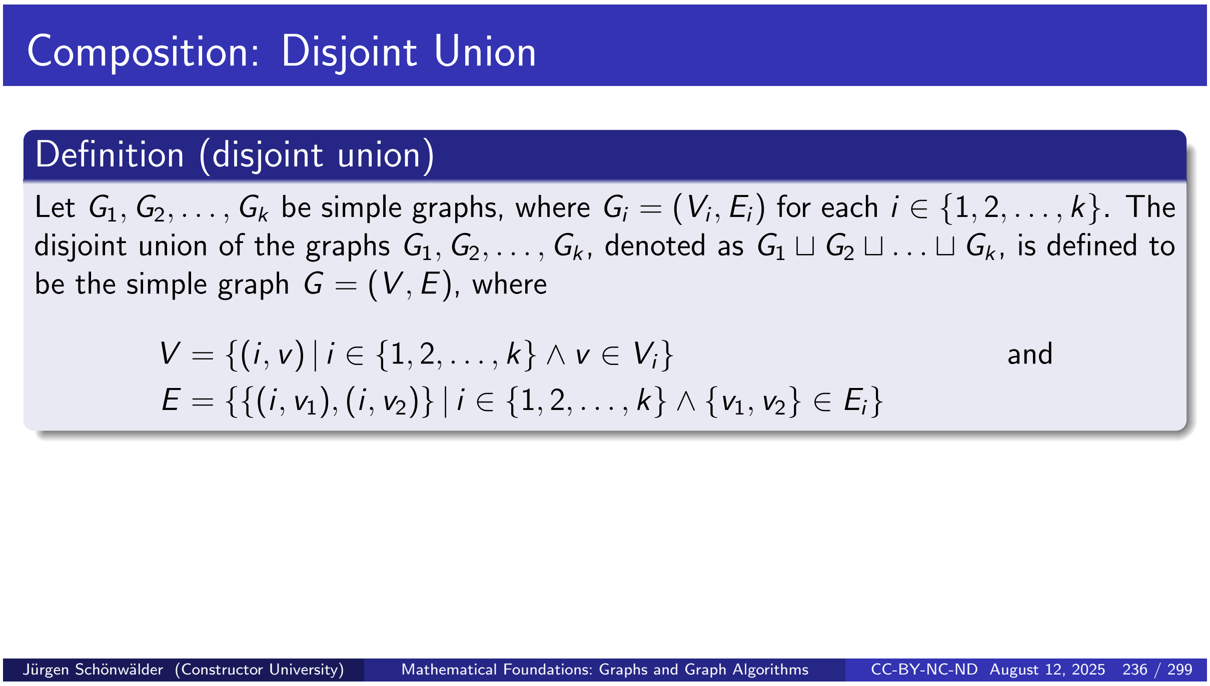
Let $G = (V, E)$ be a simple graph and u, v, w be three distinct vertices of G. The induced subgraph of G on $\{u, v, w\}$ is a *triangle of G*.

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Graphs and Graph Algorithms CC-BY-NC-ND August 12, 2025 235 / 299

Lets consider the following simple graph:



The subgraph $H = (\{a, b, d\}, \{\{a, b\}, \{b, d\}, \{a, d\}\})$ is an induced subgraph of G. It is also a triangle of G. The subgraph $I = (\{a, b, c, d\}, \{\{a, b\}, \{b, c\}, \{b, d\}\})$ is not an induced subgraph of G since the edge $\{a, d\}$ is missing.



Composition: Disjoint Union

Definition (disjoint union)

Let $G_1, G_2, \ldots, G_k$ be simple graphs, where $G_i = (V_i, E_i)$ for each $i \in \{1, 2, \ldots, k\}$. The disjoint union of the graphs $G_1, G_2, \ldots, G_k$, denoted as $G_1 \sqcup G_2 \sqcup \ldots \sqcup G_k$, is defined to be the simple graph $G = (V, E)$, where

$$V = \{(i, v) \mid i \in \{1, 2, \ldots, k\} \land v \in V_i\} \qquad \text{and}$$
$$E = \{\{(i, v_1), (i, v_2)\} \mid i \in \{1, 2, \ldots, k\} \land \{v_1, v_2\} \in E_i\}$$

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Graphs and Graph Algorithms CC-BY-NC-ND August 12, 2025 236 / 299

For the two graphs $G_1 = (\{a, b, c\}, \{\{a, b\}, \{b, c\}, \{a, c\}\})$ and $G_2 = (\{\{a, b\}, \{a, b\}\})$, the disjoint union is $G_1 \sqcup G_2 = (\{(1, a), (1, b), (1, c), (2, a), (2, b)\}, \{\{(1, a), (1, b)\}, \{(1, b), (1, c)\}, \{(1, a), (1, c)\}, \{(2, a), (2, b)\}\})$.

Decompositions: Subgraphs of Components

Theorem

Let G be a simple graph and C be a component of G. Then, the induced subgraph of G on the set C is connected.

Theorem

Let G be a simple graph and let $C_1, C_2, \dots, C_k$ be all components of G. Then G is isomorphic to the disjoint union of subgraphs induced by the components.

Section 28: Directed Graphs and Directed Multigraphs

27 Graphs and Multigraphs

28 Directed Graphs and Directed Multigraphs

29 Trees

30 Graph Traversals

31 Shortest Paths

32 Maximum Flows

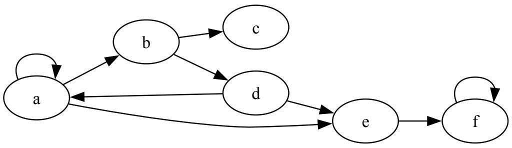
Directed Graphs

Definition (simple directed graphs)

A *simple directed graph* (also called a *digraph*) is a pair $G = (V, E)$, where V is a finite set of *vertices*, and E is a subset of $V \times V$ of *edges*. An edge $(u, v) \in E$ is also called an *arc* of G and u is called the *source* of this arc and v the *target* of this arc.

- We draw edges of digraphs as arrows pointing from the source to the target.
- Note that this definition of simple digraphs allows loops.

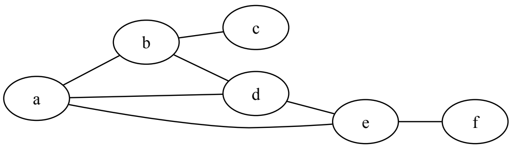
Consider the graph below. Mathematically, the graph is defined by $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ and $E = \{(a, a), (a, b), (a, e), (b, c), (b, d), (d, a), (d, e), (e, f), (f, f)\}$.



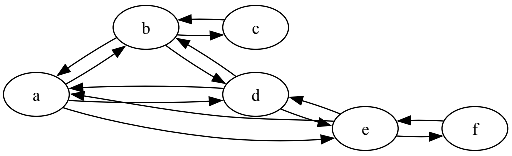
$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that adjacency matrices can still be used to represent simple directed graphs. Compared to simple graphs, the diagonal is not guaranteed anymore to be zero and the matrix is not necessarily symmetric anymore.

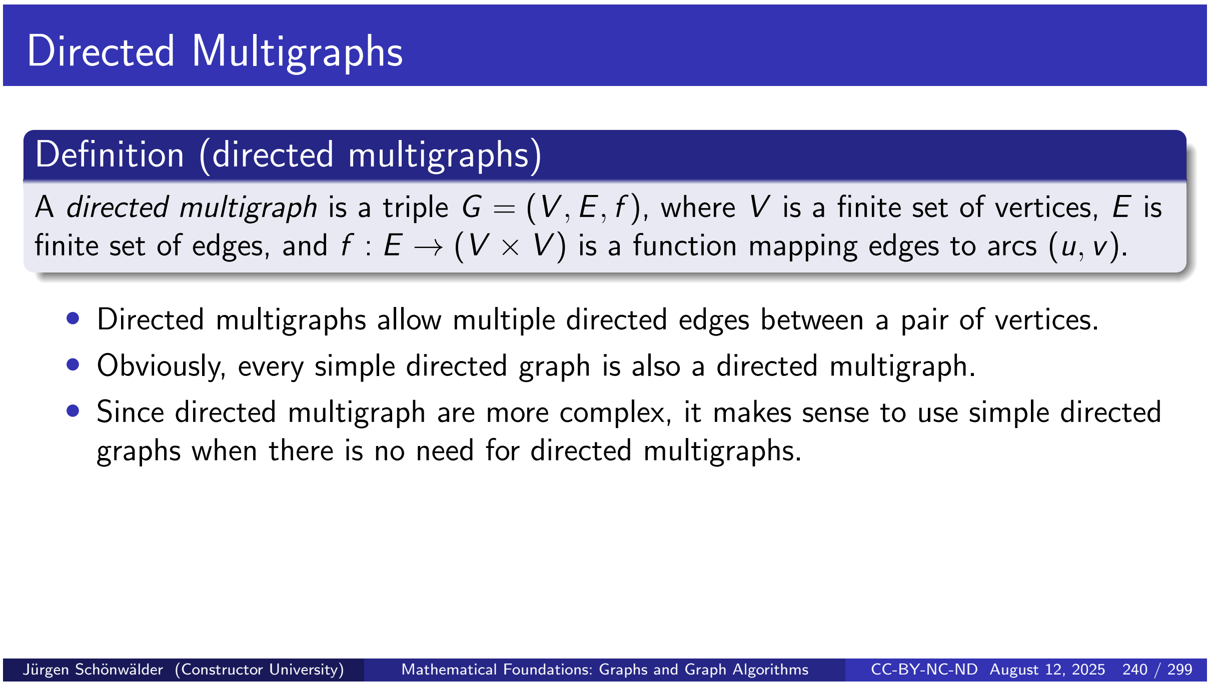
Lets informally discuss how finite simple graphs relate to simple directed graphs. If we look at the adjacency matrix of a simple graph, then this is also a valid adjacency matrix of a directed graph. However, since edges are directed in directed graph, we double the number of edges.



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



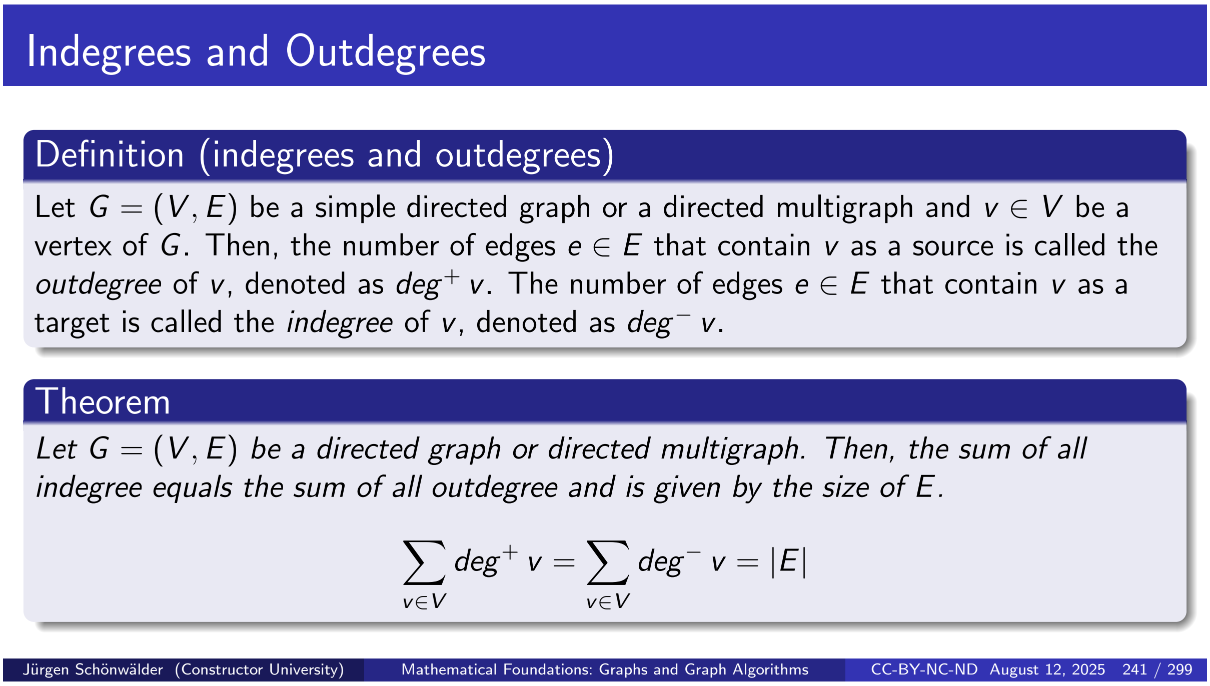
Directed Multigraphs

Definition (directed multigraphs)

A *directed multigraph* is a triple $G = (V, E, f)$, where V is a finite set of vertices, E is finite set of edges, and $f : E \to (V \times V)$ is a function mapping edges to arcs (u, v).

- Directed multigraphs allow multiple directed edges between a pair of vertices.
- Obviously, every simple directed graph is also a directed multigraph.
- Since directed multigraph are more complex, it makes sense to use simple directed graphs when there is no need for directed multigraphs.

Like we can map a finite simple graph into a directed graph by creating two directed edges for each simple edge, we can map a finite multigraph into a directed multigraph. Hence, a directed multigraph is the most general form of a graph. It is, however, also the most complex form of a graph and hence it is desirable to use the graph type necessary to solve a problem and not the most generic graph to solve a problem. Computer programs or libraries for processing graphs usually provide dedicated support for the different types of graphs.



Indegrees and Outdegrees

Definition (indegrees and outdegrees)

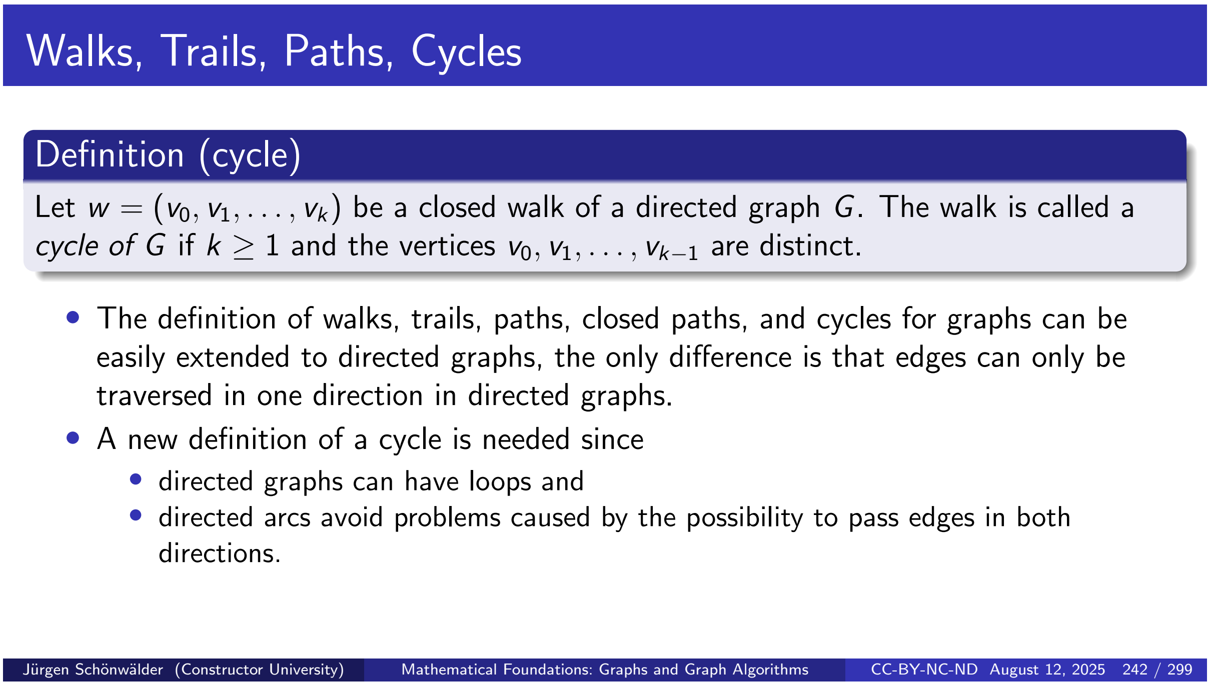
Let $G = (V, E)$ be a simple directed graph or a directed multigraph and $v \in V$ be a vertex of G. Then, the number of edges $e \in E$ that contain v as a source is called the *outdegree* of v, denoted as $deg^+ v$. The number of edges $e \in E$ that contain v as a target is called the *indegree* of v, denoted as $deg^- v$.

Theorem

Let $G = (V, E)$ be a directed graph or directed multigraph. Then, the sum of all indegree equals the sum of all outdegree and is given by the size of E.

$$\sum_{v \in V} deg^+ v = \sum_{v \in V} deg^- v = |E|$$

Proof. Since $deg^+ v$ is the number of arcs leaving v and we sum over all vertices $v \in V$, this gives us the number of all arcs in G, hence $\sum_{v \in V} deg^+ v = |E|$. With a similar argument, we obtain that $\sum_{v \in V} deg^- v = |E|$ and hence the sums are equal. □



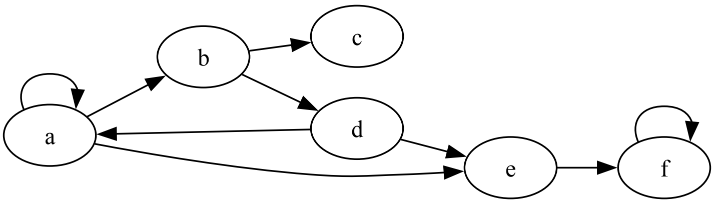
Walks, Trails, Paths, Cycles

Definition (cycle)

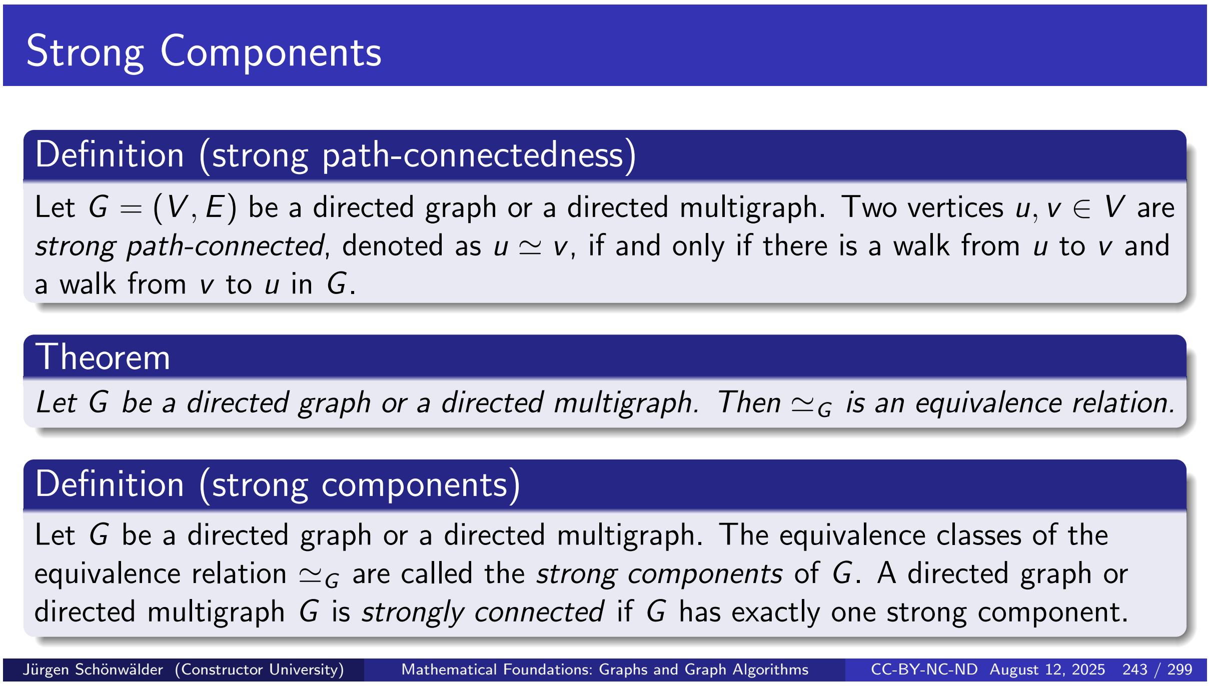
Let $w = (v_0, v_1, \ldots, v_k)$ be a closed walk of a directed graph G. The walk is called a *cycle of G* if $k \geq 1$ and the vertices $v_0, v_1, \ldots, v_{k-1}$ are distinct.

- The definition of walks, trails, paths, closed paths, and cycles for graphs can be easily extended to directed graphs, the only difference is that edges can only be traversed in one direction in directed graphs.
- A new definition of a cycle is needed since
  - directed graphs can have loops and
  - directed arcs avoid problems caused by the possibility to pass edges in both directions.

Lets consider the following simple graph:



The sequence (a, b, d, a, e) is a walk and a trail but not a path. The sequence (a, b, d, a) is a closed walk and a cycle. The sequence (a, b, c) is a path from a to c. Note that there is no reverse path from c to a.



Strong Components

Definition (strong path-connectedness)

Let $G = (V, E)$ be a directed graph or a directed multigraph. Two vertices $u, v \in V$ are *strong path-connected*, denoted as $u \simeq v$, if and only if there is a walk from u to v and a walk from v to u in G.

Theorem

Let G be a directed graph or a directed multigraph. Then $\simeq_G$ is an equivalence relation.

Definition (strong components)

Let G be a directed graph or a directed multigraph. The equivalence classes of the equivalence relation $\simeq_G$ are called the *strong components* of G. A directed graph or directed multigraph G is *strongly connected* if G has exactly one strong component.

The notion of connectedness does not directly extend from graphs to directed graphs since walks and paths are not necessarily reversible in directed graphs. Hence we introduce the notion of strong path connectedness to restore the equivalence relation necessary to define strong components.

Note that the differences here are not just technical. For graphs, the connected components were relatively easy to identify as subgraphs that are disconnected from other subgraphs. With directed graphs, strong components are subgraphs that appear within a graph that visually looks connected and in fact it is possible to have paths between vertices of different strong components.

Section 29: Trees

27 Graphs and Multigraphs

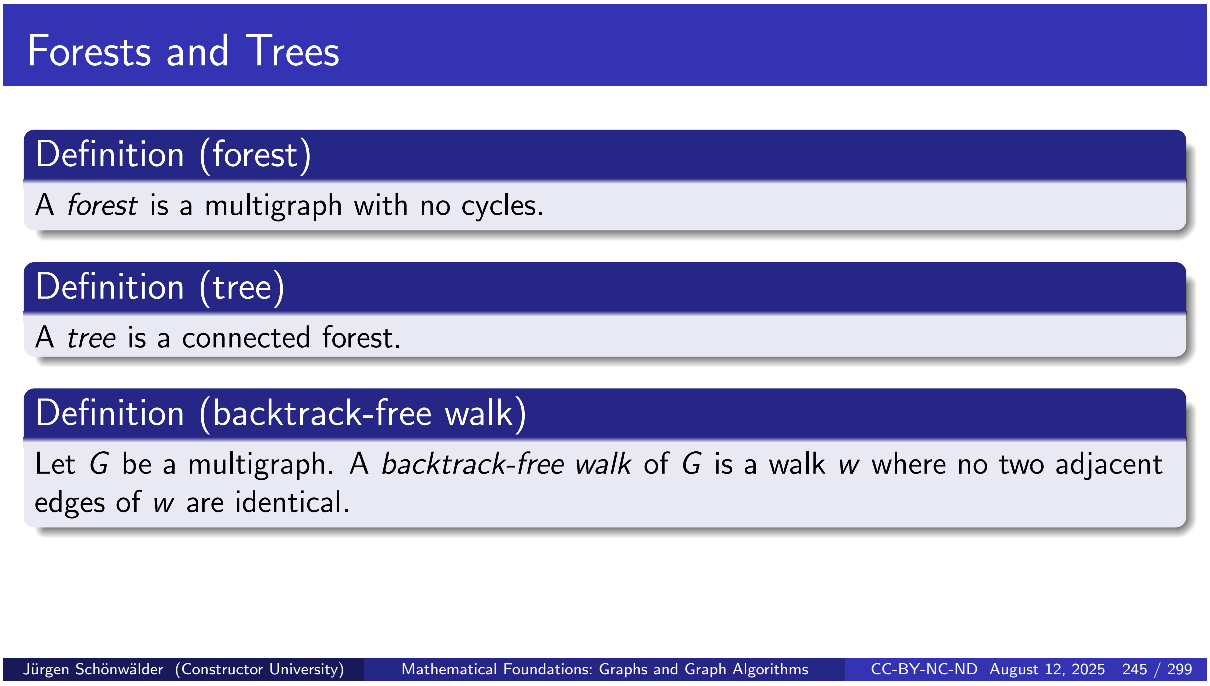
28 Directed Graphs and Directed Multigraphs

29 Trees

30 Graph Traversals

31 Shortest Paths

32 Maximum Flows



Forests and Trees

Definition (forest)

A *forest* is a multigraph with no cycles.

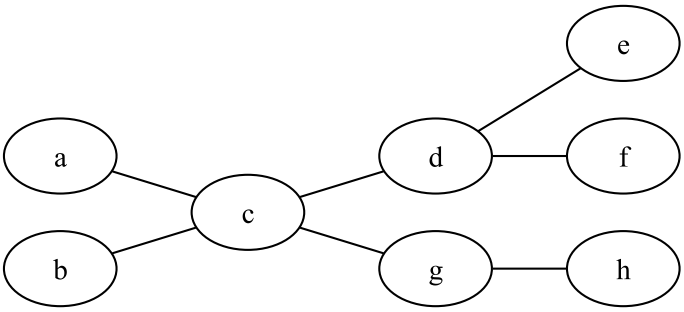
Definition (tree)

A *tree* is a connected forest.

Definition (backtrack-free walk)

Let G be a multigraph. A *backtrack-free walk* of G is a walk w where no two adjacent edges of w are identical.

Let us consider the following graph:

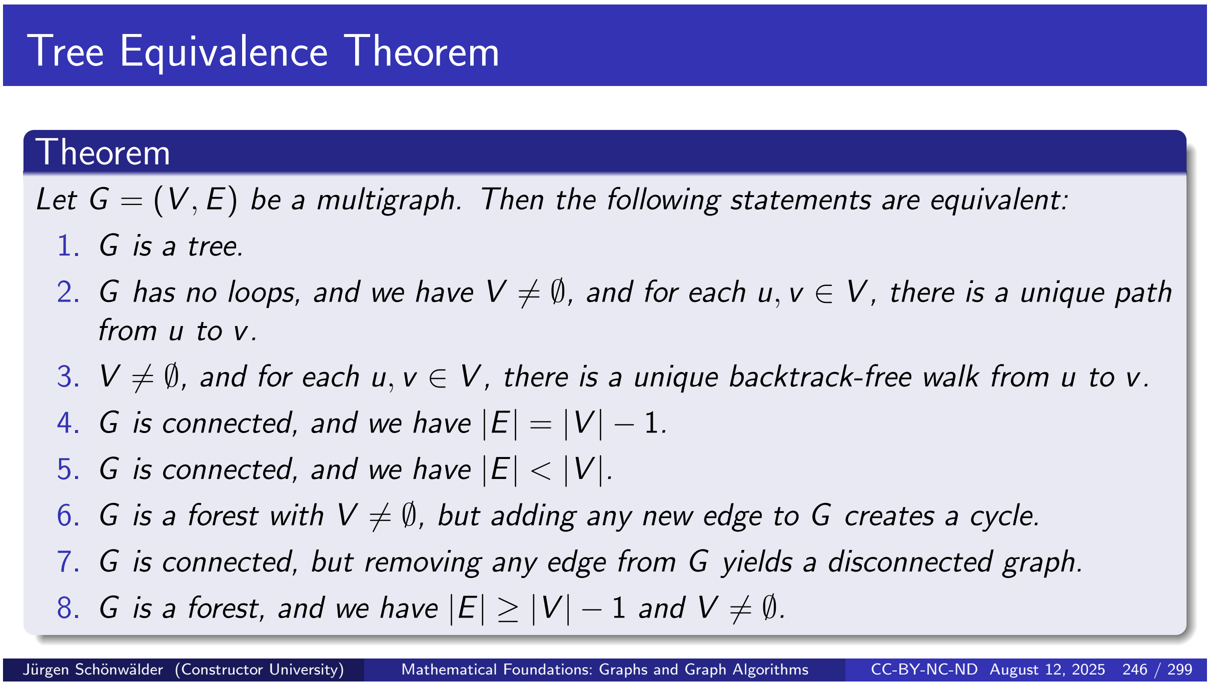


This graph is a tree with 8 nodes and 7 edges. If we would add any additional edge, the graph would have a cycle and hence it is not a tree anymore. Note that $G = (\{a\}, \emptyset)$ is a proper tree, while $F = (\{a, b, c\}, \emptyset)$ is a forest.

The mathematical notion of a tree does not require to identify a root node. In computer science, trees usually have a defined root node. Furthermore, many applications in computer science also impose a total order on the child nodes of a given vertex. In graph theory, additional structure is only imposed if required for the sake of graph theory.

Further online information:

- **Wikipedia**: Tree (graph theory)



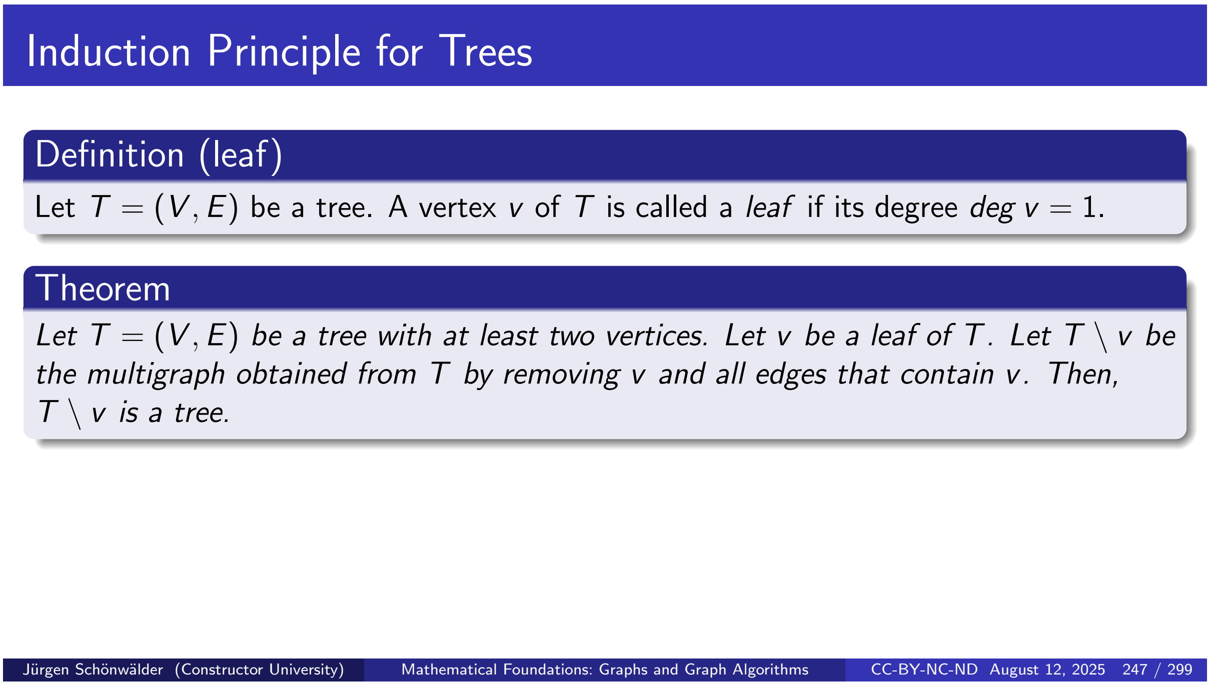
Tree Equivalence Theorem

Theorem

Let $G = (V, E)$ be a multigraph. Then the following statements are equivalent:

1. *G is a tree.*
2. *G has no loops, and we have $V \neq \emptyset$, and for each $u, v \in V$, there is a unique path from u to v.*
3. *$V \neq \emptyset$, and for each $u, v \in V$, there is a unique backtrack-free walk from u to v.*
4. *G is connected, and we have $|E| = |V| - 1$.*
5. *G is connected, and we have $|E| < |V|$.*
6. *G is a forest with $V \neq \emptyset$, but adding any new edge to G creates a cycle.*
7. *G is connected, but removing any edge from G yields a disconnected graph.*
8. *G is a forest, and we have $|E| \geq |V| - 1$ and $V \neq \emptyset$.*

This theorem can be proven by showing many implications that together establish the equivalence of the statements. Many of them are rather trivial to proof, perhaps a good exercise.



Induction Principle for Trees

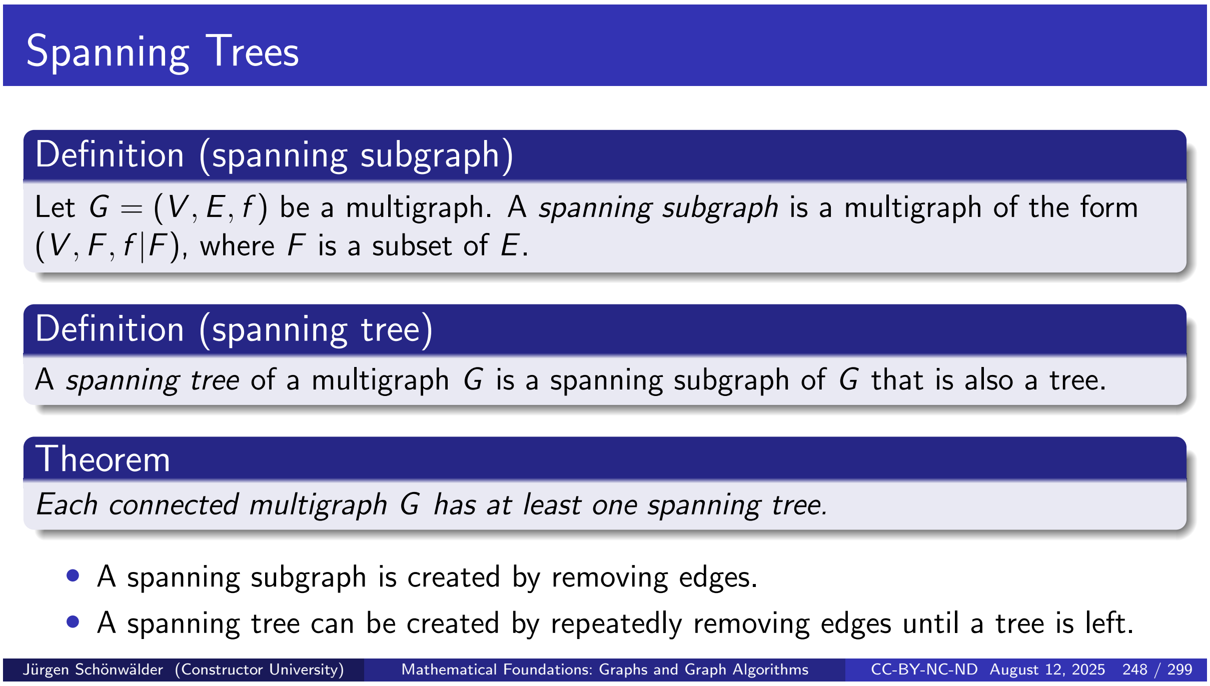
Definition (leaf)

Let $T = (V, E)$ be a tree. A vertex v of T is called a *leaf* if its degree *deg* $v = 1$.

Theorem

Let $T = (V, E)$ be a tree with at least two vertices. Let v be a leaf of T. Let $T \setminus v$ be the multigraph obtained from T by removing v and all edges that contain v. Then, $T \setminus v$ is a tree.

The induction principle for trees says that we iteratively can remove leafs from a tree until eventually a tree with a single vertex and no edges is left. This essentially reduces any tree in a sequence of steps to a degenerated tree of a single vertex. Note that we can turn this around: If we have a tree and we attach a new vertex as a new leaf somewhere to the tree, the result is still a tree.



Spanning Trees

Definition (spanning subgraph)

Let $G = (V, E, f)$ be a multigraph. A *spanning subgraph* is a multigraph of the form $(V, F, f|F)$, where F is a subset of E.

Definition (spanning tree)

A *spanning tree* of a multigraph G is a spanning subgraph of G that is also a tree.

Theorem

Each connected multigraph G has at least one spanning tree.

- A spanning subgraph is created by removing edges.
- A spanning tree can be created by repeatedly removing edges until a tree is left.

Spanning trees are an important building block. Many operations on multigraphs can be simplified by first constructing a spanning tree and then executing the operation on the spanning tree.

Section 30: Graph Traversals

27 Graphs and Multigraphs

28 Directed Graphs and Directed Multigraphs

29 Trees

30 Graph Traversals

31 Shortest Paths

32 Maximum Flows

Graph Traversals

Definition (graph traversal)

Let $G = (V, E)$ be a connected graph. A *graph traversal* is a systematic method for visiting every vertex of G from a given start vertex $s \in V$.

Definition (queue)

A *queue* is a collection of elements maintained in sequence where elements can be added (enqueued) at the end of the queue and be removed (dequeued) from the front of the queue.

Definition (stack)

A *stack* is a collection of elements maintained in sequence where elements can be added (pushed) on the top of the stack and be removed (popped) from the top of the stack.

Graph traversal algorithms play an important role in computer science. They provide a building block for many other interesting algorithms. To describe graph traversal algorithms, we need some data structures to keep track where we are in a traversal. In addition to a simple sets, we need queues and stacks.

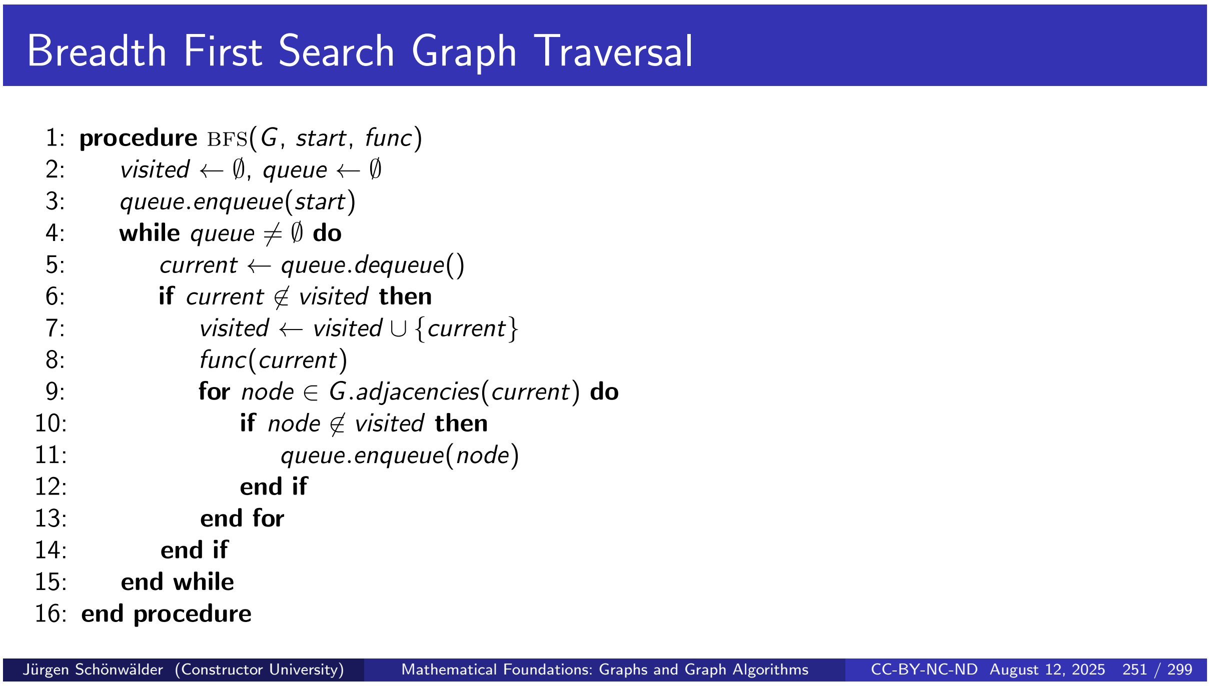
A queue is a collection holding elements. New elements are added to the end of a queue and elements leave the queue only at the front of the queue. Elements do not change their position in the queue. A queue has first-in-first-out behaviour. You all know queues from the coffee bar (except that people occasionally do not enqueue at the end, but lets ignore this behaviour for now).

A stack is another collection holding elements. New elements are added on the top of the stack and elements are removed from the top of the stack. Elements do not change their position in the stack. A stack has last-in-first-out behaviour. You all know stacks from your cupboards where you have a stack of plates. When you need a plate, you take one of the top of the stack of plates. After cleaning a plate, you put it back on the top of the stack of plates.

Graph traversal algorithms use queues and stacks in order to remember that some portions of a graph still need to be explored.

Further online information:

- **Wikipedia**: Graph traversal

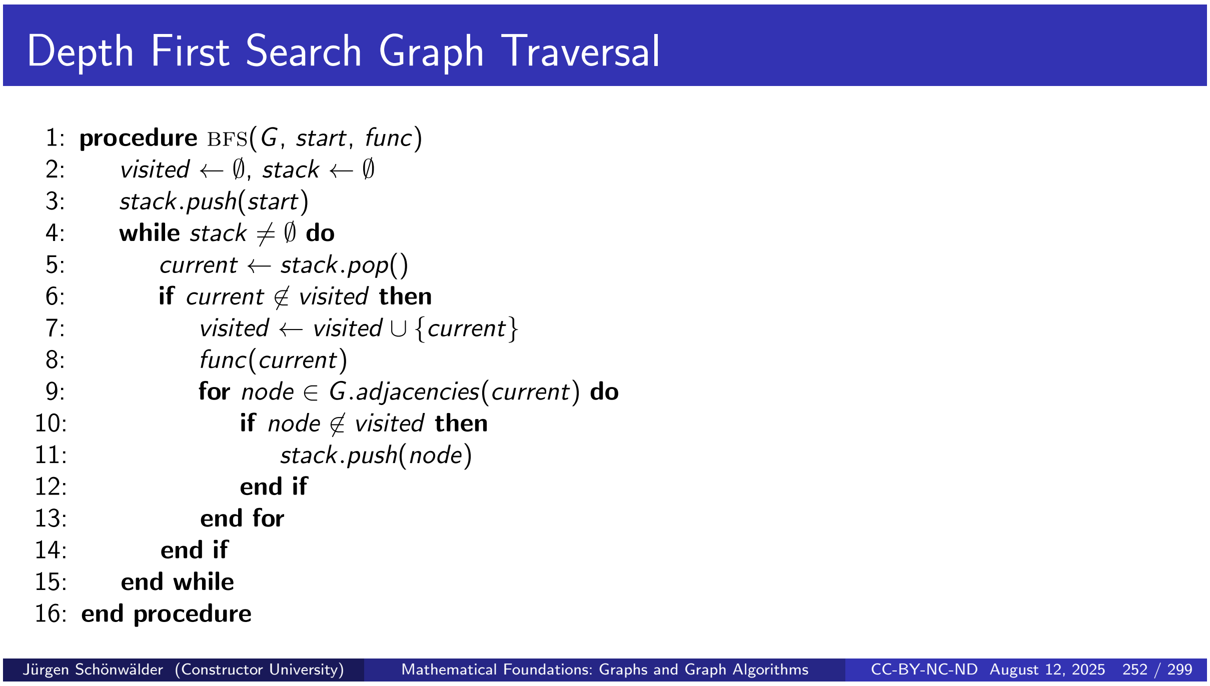


Breadth First Search Graph Traversal

```
procedure BFS($G$, $start$, $func$)
    $visited \leftarrow \emptyset$, $queue \leftarrow \emptyset$
    $queue.enqueue(start)$
    while $queue \neq \emptyset$ do
        $current \leftarrow queue.dequeue()$
        if $current \notin visited$ then
            $visited \leftarrow visited \cup \{current\}$
            $func(current)$
            for $node \in G.adjacencies(current)$ do
                if $node \notin visited$ then
                    $queue.enqueue(node)$
                end if
            end for
        end if
    end while
end procedure
```

Below is an implementation in Python traversing a graph in breadth-first order and returning the sequence of vertices. The code uses the `networkx` Python library for the representation of graphs.

```python
import collections
import networkx as nx

def bfs(graph, start):
    visited = set()
    queue = collections.deque([start])
    order = []

    while queue:
        current = queue.popleft();
        if current not in visited:
            order.append(current)
            visited.add(current)
            for node in graph[current]:
                if node not in visited:
                    queue.append(node)

    return order
```



Depth First Search Graph Traversal

```
procedure BFS($G$, $start$, $func$)
    $visited \leftarrow \emptyset$, $stack \leftarrow \emptyset$
    $stack.push(start)$
    while $stack \neq \emptyset$ do
        $current \leftarrow stack.pop()$
        if $current \notin visited$ then
            $visited \leftarrow visited \cup \{current\}$
            $func(current)$
            for $node \in G.adjacencies(current)$ do
                if $node \notin visited$ then
                    $stack.push(node)$
                end if
            end for
        end if
    end while
end procedure
```

Below is an implementation in Python traversing a graph in depth-first order and returning the sequence of vertices. The code uses the `networkx` Python library for the representation of graphs.

```python
import collections
import networkx as nx

def dfs(graph, start):
    visited = set()
    queue = collections.deque([start])
    order = []

    while queue:
        current = queue.popleft();
        if current not in visited:
            order.append(current)
            visited.add(current)
            for node in graph[current]:
                if node not in visited:
                    queue.appendleft(node)

    return order
```

A depth-first traversal can be implemented using a recursive function, which uses the call stack to keep track of the nodes still to be explored.

Section 31: Shortest Paths

27 Graphs and Multigraphs

28 Directed Graphs and Directed Multigraphs

29 Trees

30 Graph Traversals

31 Shortest Paths

32 Maximum Flows

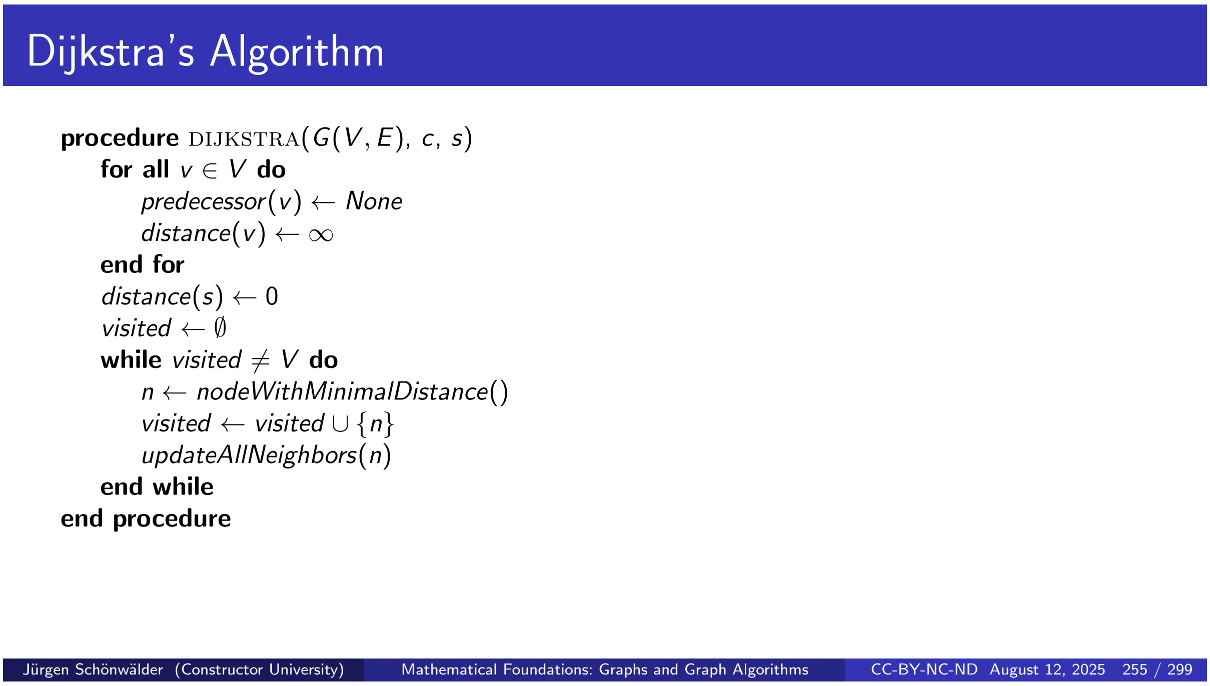
Single Source Shortest Paths

Definition (shortest-path)

Let $G = (V, E)$ be a connected graph and $c : E \to \mathbb{N}$ be a cost function. A *shortest path* from a vertex v_1 to a vertex v_n is a path $p = (v_1, v_2, \ldots, v_n)$ minimizing the path cost $n = \sum_{i=1}^{n-1} c(\{v_i, v_{i+1)}\})$.

- The *single-pair shortest path problem* is the problem of finding a shortest path from a given source vertex to a given destination vertex.
- The *single-source shortest path problem* is the problem of finding the shortest paths from a given source vertex to all possible destinations.
- The *all-pairs shortest path problem* is the problem of finding the shortest paths from all possible sources to all possible destinations.

Shortest-path problems play an important role. When planning a trip using online maps, we often have single-pair shortest path problems solved by the map software. In communication networks, routers often solve the single-source shortest path problem in order to determine how to forward data packets efficiently.

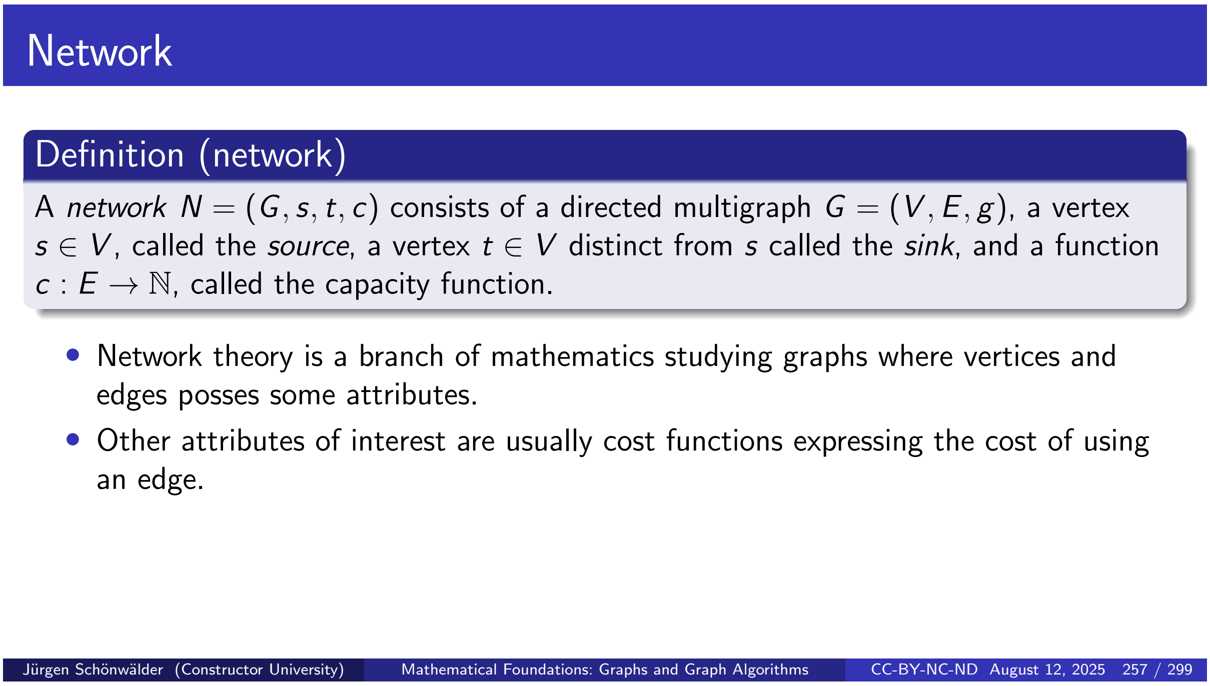


Dijkstra's Algorithm

```
procedure DIJKSTRA($G(V, E)$, $c$, $s$)
    for all $v \in V$ do
        $predecessor(v) \leftarrow None$
        $distance(v) \leftarrow \infty$
    end for
    $distance(s) \leftarrow 0$
    $visited \leftarrow \emptyset$
    while $visited \neq V$ do
        $n \leftarrow nodeWithMinimalDistance()$
        $visited \leftarrow visited \cup \{n\}$
        $updateAllNeighbors(n)$
    end while
end procedure
```

```python
import heapq
import networkx as nx

def dijkstra(graph, start):
    queue = [(0, start)]                    # used as a priority queue
    distances = {node: float('inf') for node in graph.nodes()}
    distances[start] = 0
    predecessors = {}
    visited = set()

    while queue:
        current_distance, current = heapq.heappop(queue)
        if current not in visited:
            visited.add(current)
            for neigh in graph[current]:
                distance = current_distance + graph[current][neigh]["weight"]
                if distance < distances[neigh]:
                    distances[neigh] = distance
                    predecessors[neigh] = current
                    heapq.heappush(queue, (distance, neigh))

    return distances
```

Section 32: Maximum Flows



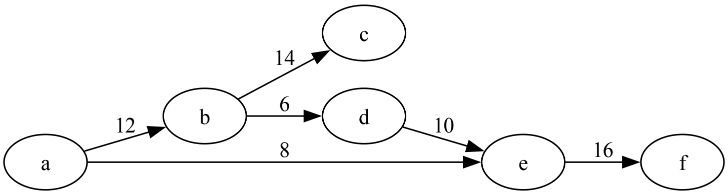
Network

Definition (network)

A *network* $N = (G, s, t, c)$ consists of a directed multigraph $G = (V, E, g)$, a vertex $s \in V$, called the *source*, a vertex $t \in V$ distinct from s called the *sink*, and a function $c : E \to \mathbb{N}$, called the capacity function.

- Network theory is a branch of mathematics studying graphs where vertices and edges posses some attributes.
- Other attributes of interest are usually cost functions expressing the cost of using an edge.

Lets consider the following network with the source a and the sink f. The capacity is shown as labels next to the edges.



From the visualization, it is clear that a can at maximum inject 20 units into the network and it is also clear that f can at maximum only receive 16 units. Given the capacity of the edges, a common question is what is the maximum flow that the sink f can receive? For a small graph, we can easily find the solution: There are two independent paths from a to f, namely (a, b, d, e, f) and (a, e, f). The first subpath to e is limited to 6 units while the second subpath to e is limited to 8 units. This means e can at most receive 14 units and since this is below the capacity of the edge from e to f, the maximum flow that can be received at the sink is 14 units.

Flow

Definition (flow)

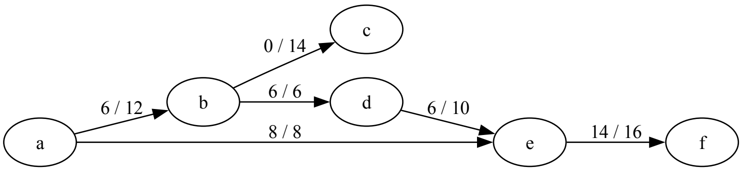
Let $N = (G, s, t, c)$ be a network with $G = (V, E, g)$. A *flow* on the network N is a function $f : E \to \mathbb{N}$ satisfying the following constraints:

$$\begin{array}{lcr} \forall e \in E & 0 \leq f(e) \leq c(e) & \text{capacity constraints} \\ \forall v \in V \setminus \{s, t\} & f^{-}(v) = f^{+}(v) & \text{conversation constraints} \end{array}$$

The *inflow* of f, denoted as $f^{-}(v)$, and the *outflow* of f, denoted as $f^{+}(v)$, are defined as follows:

$$f^{-}(v) = \sum_{e \in E, v \text{ target of } e} f(e) \qquad f^{+}(v) = \sum_{e \in E, v \text{ source of } e} f(e)$$

We can indicate the flow by labeling edges with both the edge's flow and capacity.



The graph shows the maximum flow possible between the source $s = a$ and the sink $t = e$ is 14, given the bottleneck edges (b, d) and (a, e).

Cuts and Capacity of a Cut

Definition ($s - t$ cut)

Let $N = (G, s, t, c)$ be a network consisting of a directed multigraph $G = (V, E, g)$, a source $s \in V$, a sink $t \in V$ and a capacity function $c : E \to \mathbb{N}$.

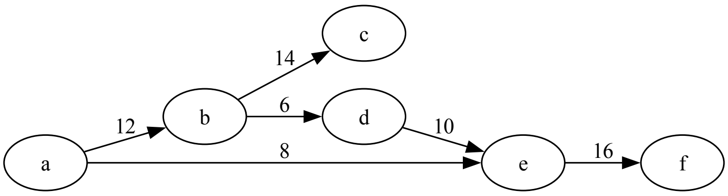
An $s - t$ *cut* of N consists of a set $S \subset V$ and its complement $\bar{S} = V \setminus S$ such that $s \in S$, $t \in \bar{S}$ and $S \cup \bar{S} = V$ and $V \cap \bar{S} = \emptyset$.

The *capacity* of an $s - t$ cut S is given by:

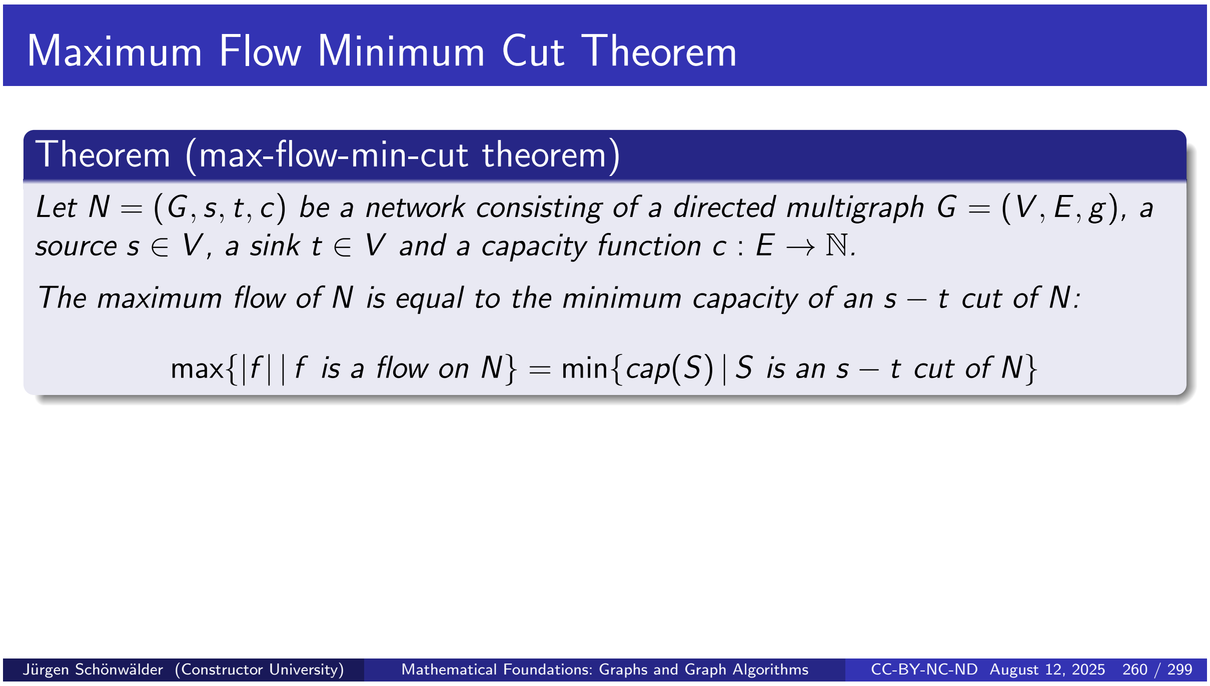
$$cap(S) = \sum_{\substack{e=(u,v)\in E \\ u\in S, v\in\bar{S}}} c(e)$$

A cut separates a part of the graph containing the source node from the part of the graph containing the sink node. The capacity of a cut sums up the capacity of all edges starting in the part containing the source and ending in the part containing the sink. Different cuts can have different capacity.

Lets consider the following network with the source a and the sink f.



The cut $S = \{a, b, c, d, e\}$ and $\bar{S} = \{f\}$ has the capacity $cap(S) = 16$. The cut $S' = \{a, b, c, d\}$ and $\bar{S'} = \{e, f\}$ has the capacity $cap(S') = 18$. Finally, the cut $S'' = \{a, b, c\}$ and $\bar{S''} = \{d, e, f\}$ has the capacity $cap(S'') = 14$.



Maximum Flow Minimum Cut Theorem

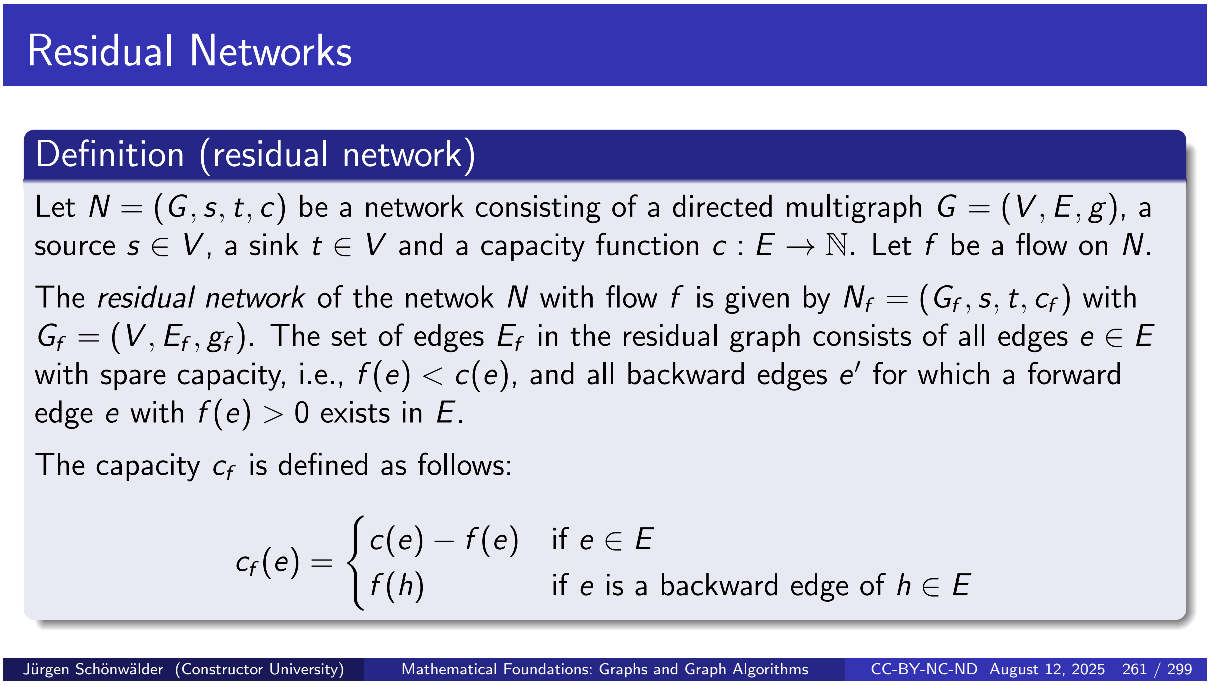
Theorem (max-flow-min-cut theorem)

Let $N = (G, s, t, c)$ be a network consisting of a directed multigraph $G = (V, E, g)$, a source $s \in V$, a sink $t \in V$ and a capacity function $c : E \to \mathbb{N}$.

The maximum flow of N is equal to the minimum capacity of an $s - t$ cut of N:

$$\max\{|f| \mid f \text{ is a flow on } N\} = \min\{cap(S) \mid S \text{ is an } s - t \text{ cut of } N\}$$

Every possible cut defines an upper bound for the maximum flow that is possible between the source and the sink. Hence, finding the cut with the smallest capacity is intuitively the same as finding the maximum flow.



Residual Networks

Definition (residual network)

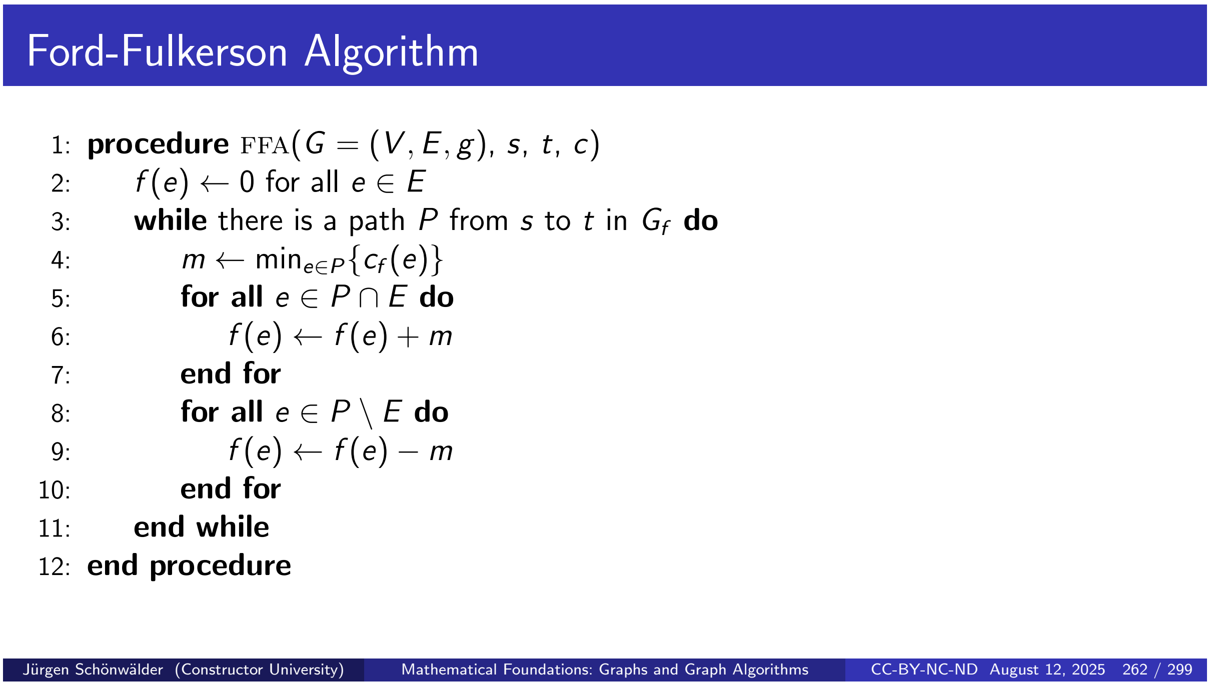
Let $N = (G, s, t, c)$ be a network consisting of a directed multigraph $G = (V, E, g)$, a source $s \in V$, a sink $t \in V$ and a capacity function $c : E \to \mathbb{N}$. Let f be a flow on N.

The *residual network* of the netwok N with flow f is given by $N_f = (G_f, s, t, c_f)$ with $G_f = (V, E_f, g_f)$. The set of edges E_f in the residual graph consists of all edges $e \in E$ with spare capacity, i.e., $f(e) < c(e)$, and all backward edges e' for which a forward edge e with $f(e) > 0$ exists in E.

The capacity c_f is defined as follows:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(h) & \text{if } e \text{ is a backward edge of } h \in E \end{cases}$$

The residual edges capture the idea that a flow in one direction may be pushed back into the other direction.



Ford-Fulkerson Algorithm

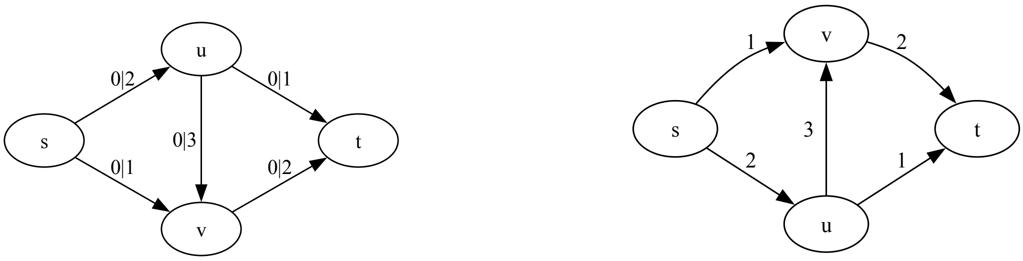
```
procedure FFA($G = (V, E, g)$, $s$, $t$, $c$)
    $f(e) \leftarrow 0$ for all $e \in E$
    while there is a path $P$ from $s$ to $t$ in $G_f$ do
        $m \leftarrow \min_{e \in P}\{c_f(e)\}$
        for all $e \in P \cap E$ do
            $f(e) \leftarrow f(e) + m$
        end for
        for all $e \in P \setminus E$ do
            $f(e) \leftarrow f(e) - m$
        end for
    end while
end procedure
```

Jürgen Schönwälder (Constructor University) Mathematical Foundations: Graphs and Graph Algorithms CC-BY-NC-ND August 12, 2025 262 / 299

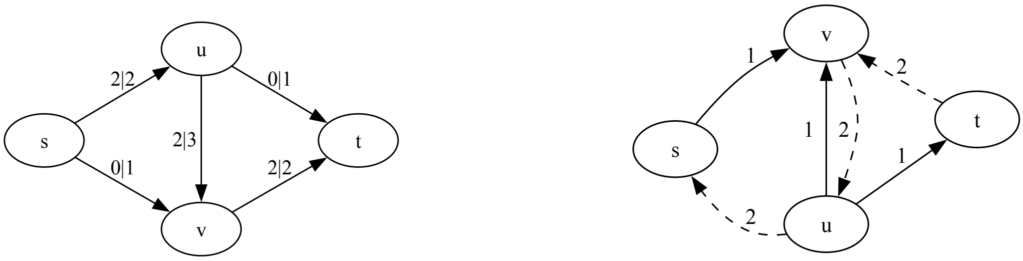
Initially, the flows on all edges are set to zero. The algorithm the tries to find a path from the source s to the sink t in the residual network. As long as such a path exists, the bottleneck capacity is determined and then the flows on all regular edges and all residual edges are updated. The algorithm terminates once there are no more paths in the residual network. The algorithms returns a maximum flow. Note that different flows with the same maximum flow may be possible. The solution returned depends on the choice of the algorithm for finding paths from s to t.

In the following example, the flow network is shown on the left and the corresponding residual network on the right.

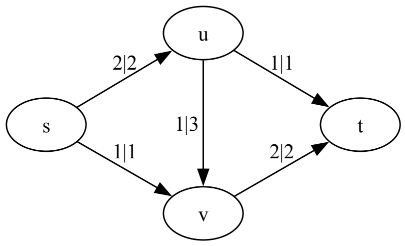
- Initial state with empty flows:

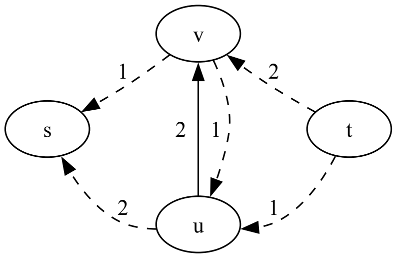


- Selecting the path $s \to u \to v \to t$ with the capacity 2:



- Selecting the path $s \to v \to u \to t$ with the capacity 1:





Part VIII

Software Correctness

We generally want our software to be correct. In this part, we define software correctness and we study a classic technique, called the Floyd-Hoare logic, to prove the correctness of programs written in a simple imperative programming language.

Once we have developed the necessary ideas to prove the correctness of programs, we will investigate ways to automate some of the software verification process. The treatment of Floyd-Hoare triples and Floyd-Hoare logic is largely based on Mike Gordon's exececellent "Background reading on Hoare Logic".

By the end of this part, students should be able to

- recall the difference between partial and total correctness;
- articulate Floyd-Hoare triples and preconditions and postconditions;
- elaborate the role and importance of formal software specifications;
- describe the rules of the Hoare logic for program verification;
- illustrate precondition strengthening and postcondition weakening;
- explain the relevance of the weakest precondition and the strongest postcondition;
- sketch how annotations help to (partially) automate software verification;
- demonstrate how loop invariants are used for partial correctness proofs;
- outline how loop variants are used for total correctness proofs.

Section 33: Software Specification

33 Software Specification

34 Software Verification

35 Automation of Software Verification

Formal Specification and Verification

Definition (formal specification)

A *formal specification* uses a formal (mathematical) notation to provide a precise definition of what a program should do.

Definition (formal verification)

A *formal verification* uses logical rules to mathematically prove that a program satisfies a formal specification.

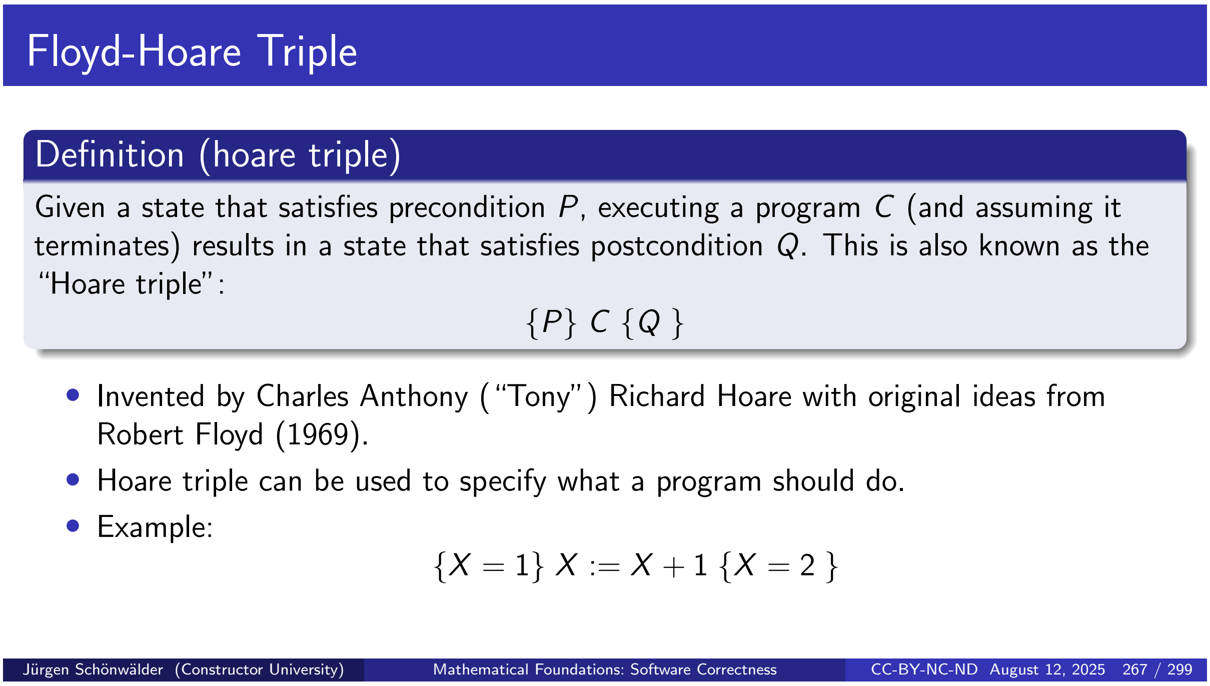
- For many non-trivial problems, creating a formal, correct, and complete specification is a problem by itself.
- A bug in a formal specification leads to programs with verified bugs.

Processors (CPUs) are designed using hardware description languages and we commonly assume that processors are correct. However, efficient designs introduce significant complexity and this increases the likelihood of introducing bugs. Fixing design flaws after silicon has been produced and deployed is very expensive. A classic example is the Intel Pentium FDIV bug discovered in 1994. Many more processor design flaws have been discovered since then.

Modern compilers are also very complex software designs, but still they tend to be super reliable for most users. One reason is that they are tested extensively and serious bugs can be fixed relatively quickly. This is a fundamental difference to systems that are difficult to fix once deployed. Creating certified correct compilers is still a very complex undertaking but required in some parts of the industry where safety-critical code must be written.

Further online information:

- **Wikipedia**: Pentium FDIV bug



Floyd-Hoare Triple

Definition (hoare triple)

Given a state that satisfies precondition P, executing a program C (and assuming it terminates) results in a state that satisfies postcondition Q. This is also known as the "Hoare triple":

$$\{P\}\ C\ \{Q\ \}$$

- Invented by Charles Anthony ("Tony") Richard Hoare with original ideas from Robert Floyd (1969).
- Hoare triple can be used to specify what a program should do.
- Example:

$$\{X = 1\}\ X := X + 1\ \{X = 2\ \}$$

The classic publication introducing Hoare logic is [9]. Tony Hoare has made several other notable contributions to computer science: He invented the basis of the Quicksort algorithm (published in 1962) and he has developed the formalism Communicating Sequential Processes (CSP) to describe patterns of interaction in concurrent systems (published in 1978).

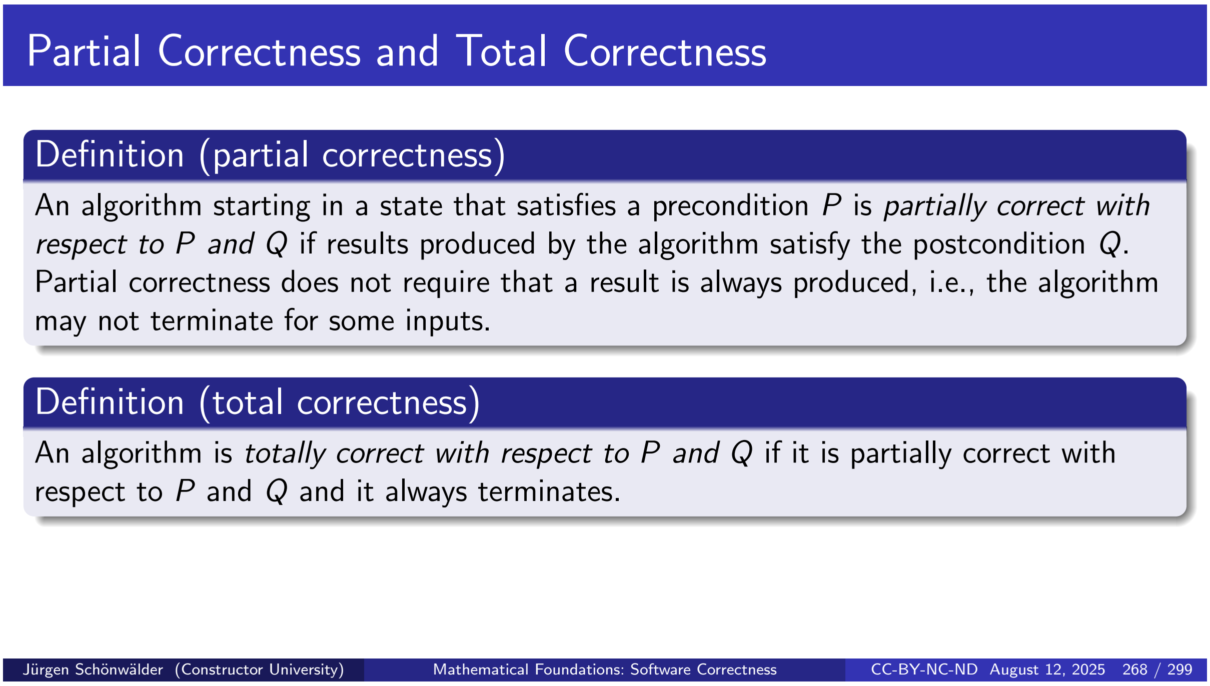
P and Q are logcial expressions on program variables. They are written using standard mathematical notation and logical operators. The predicate P defines the subset of all possible states for which a program C is defined. Similarly, the predicate Q defines the subset of all possible states for which the program's result is defined.

It is possible that different programs satisfy the same specification:

$$\{X = 1\ \}\ Y := 2\ \{X = 1 \land Y = 2\ \}$$

$$\{X = 1\ \}\ Y := 2 * X\ \{X = 1 \land Y = 2\ \}$$

$$\{X = 1\ \}\ Y := X + X\ \{X = 1 \land Y = 2\ \}$$



Partial Correctness and Total Correctness

Definition (partial correctness)

An algorithm starting in a state that satisfies a precondition P is *partially correct with respect to P and Q* if results produced by the algorithm satisfy the postcondition Q. Partial correctness does not require that a result is always produced, i.e., the algorithm may not terminate for some inputs.

Definition (total correctness)

An algorithm is *totally correct with respect to P and Q* if it is partially correct with respect to P and Q and it always terminates.

The distinction between partial correctness and total correctness is of fundamental importance. Total correctness requires termination, which is generally impossible to prove in an automated way as this would require to solve the famous halting problem. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

A definition of the form $\{P\}\ C\ \{Q\ \}$ usually provides a partial correctness specification. We use the notation $[\ P\ ]\ C\ [\ Q\ ]$ for a total correctness specification.

Further online information:

- **YouTube**: How This One Question Breaks Computers

Hoare Notation Conventions

1. The symbols V, V_1, ..., V_n stand for arbitrary variables. Examples of particular variables are X, Y, R etc.
2. The symbols E, E_1, ..., E_n stand for arbitrary expressions (or terms). These are expressions like $X + 1$, $\sqrt{2}$ etc., which denote values (usually numbers).
3. The symbols S, S_1, ..., S_n stand for arbitrary statements. These are conditions like $X < Y$, $X^2 = 1$ etc., which are either true or false.
4. The symbols C, C_1 , ..., C_n stand for arbitrary commands of our programming language; these commands are described on the following slides.

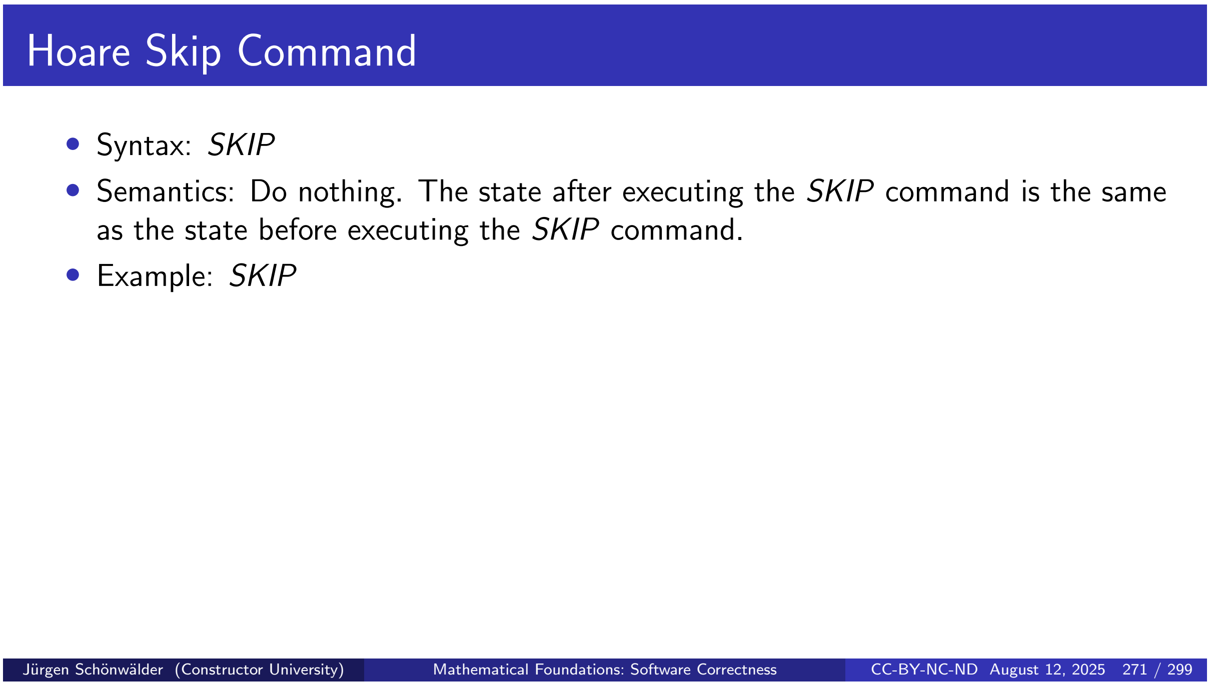
- We will use lowercase letters such as x and y to denote auxiliary variables (e.g., to denote values stored in variables).

We are focusing in the following on a purely imperative programming model where a global set of variables determines the current state of the computation. A subset of the variables are used to provide the input to an algorithm and another subset of the variables provides the output of an algorithm.

Note that we talk about a programming language consisting of commands and we use the term statements to refer to conditions. This may be a bit confusing since programming languages often call our commands statements and they may call our statements conditions.

Hoare Assignments

- Syntax: $V := E$
- Semantics: The state is changed by assigning the value of the expression E to the variable V. All variables are assumed to have global scope.
- Example: $X := X + 1$



Hoare Skip Command

- Syntax: *SKIP*
- Semantics: Do nothing. The state after executing the *SKIP* command is the same as the state before executing the *SKIP* command.
- Example: *SKIP*

The $SKIP$ command does nothing. It is still useful since it allows us to construct a single conditional command.

Hoare Command Sequences

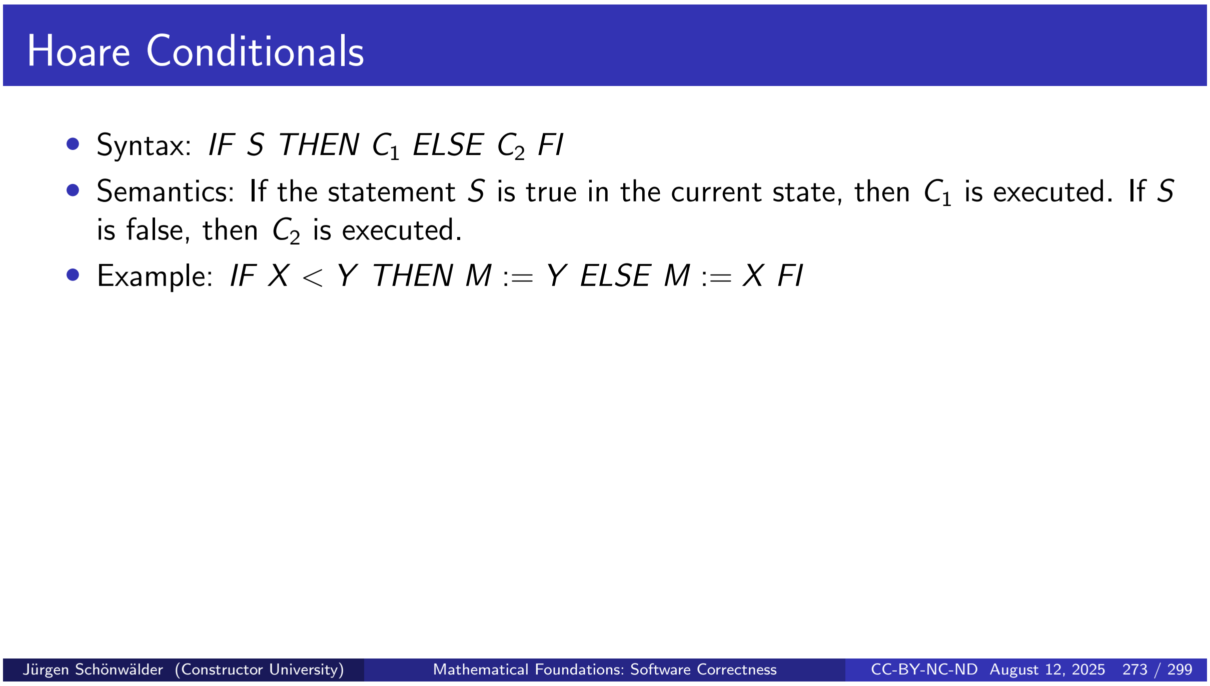
- Syntax: $C_1; \dots ; C_n$
- Semantics: The commands $C_1, \dots, C_n$ are executed in that order.
- Example: $R := X; X := Y; Y := R$

The example sequence shown above swaps the content of X and Y. Note that it has a side-effect since it also assigns the initial value of X to R. A specification of the swap program as a Floyd-Hoare triple would be the following:

$$\{\ X = x \land Y = y\ \}\ R := X; X := Y; Y := R\ \{\ X = y \land Y = x\ \}$$

Since the program does not involve any loops, we can easily specify total correctness as well:

$$[\ X = x \land Y = y\ ]\ R := X; X := Y; Y := R\ [\ X = y \land Y = x\ ]$$



Hoare Conditionals

- Syntax: *IF S THEN C_1 ELSE C_2 FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Example: *IF $X < Y$ THEN $M := Y$ ELSE $M := X$ FI*

Note that we can use $SKIP$ to create conditional statements without a $THEN$ or $ELSE$ branch:

IF S THEN C ELSE SKIP FI

IF S THEN SKIP ELSE C FI

Hoare While Loop

- Syntax: $WHILE\ S\ DO\ C\ OD$
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- Example: $WHILE\ \neg(X = 0)\ DO\ X := X - 2\ OD$

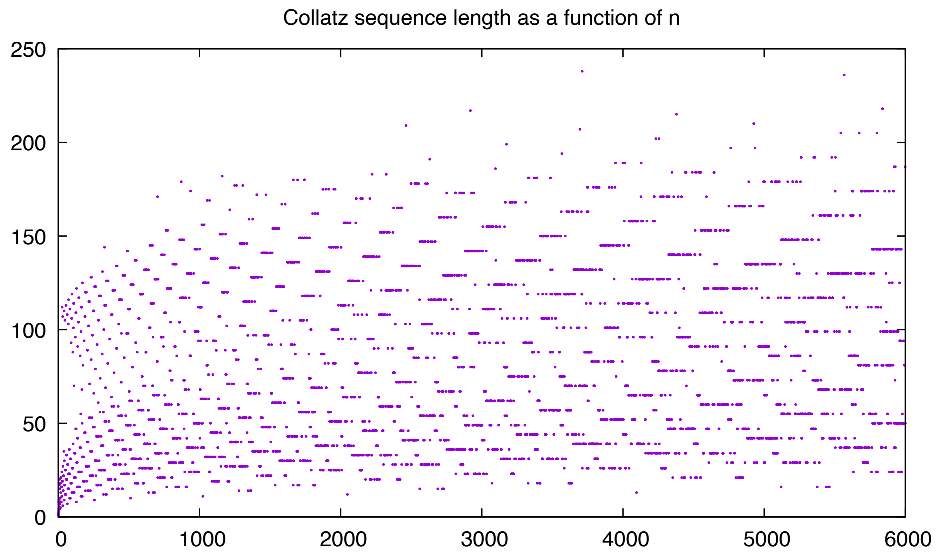
Our notation uses a convention that was popular in the 1970s to denote the end of a programming language construct by repeating a keyword with the letters reversed. An early programming language using this notation was Algol 68. You find similar syntactic ideas in Bourne shells (if / fi, case / esac). More modern languages tend to use curly braces instead or they require suitable indentation to find the end of a command sequence.

Termination can be Tricky

```
function COLLATZ($X$)
    while $X > 1$ do
        if $(X\%2) \neq 0$ then
            $X \leftarrow (3 \cdot X) + 1$
        else
            $X \leftarrow X/2$
        end if
    end while
    return $X$
end function
```

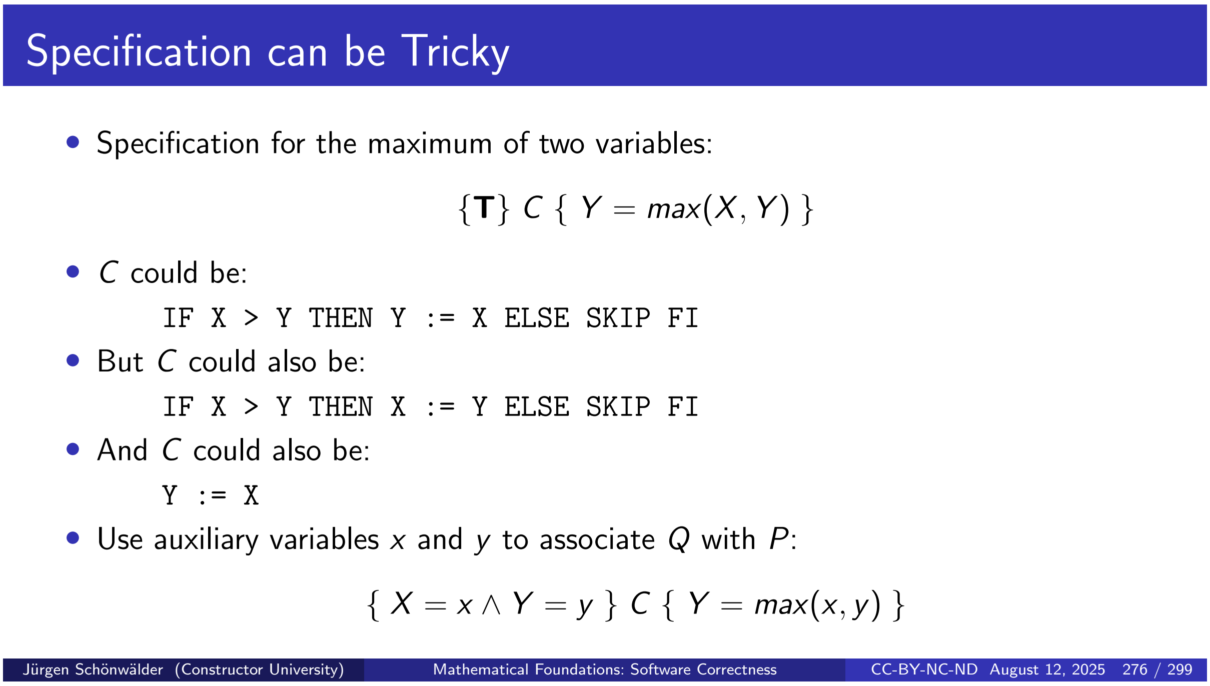
- Collatz conjecture: The program will eventually return the number 1, regardless of which positive integer is chosen initially.

This program calculates the so called Collatz sequence. The Collatz conjecture says that no matter what value of $n \in \mathbb{N}$ you start with, the sequence will always reach 1. For example, starting with $n = 12$, one gets the sequence $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$. The Collatz conjecture is one of the famous unsolved problems in mathematics. Plotting the length of the collatz sequence gives us interesting plots to look at:



Further online information:

- **Wikipedia**: Collatz conjecture
- **YouTube**: The Simplest Math Problem No One Can Solve - Collatz Conjecture



Specification can be Tricky

- Specification for the maximum of two variables:

$$\{\mathbf{T}\}\ C\ \{\ Y = max(X, Y)\ \}$$

- C could be:

```
IF X > Y THEN Y := X ELSE SKIP FI
```

- But C could also be:

```
IF X > Y THEN X := Y ELSE SKIP FI
```

- And C could also be:

```
Y := X
```

- Use auxiliary variables x and y to associate Q with P:

$$\{\ X = x \land Y = y\ \}\ C\ \{\ Y = max(x, y)\ \}$$

Obviously, multiple programs can satisfy a given specification:

$$\{\ X = 1\ \}\ Y := 2\ \{\ X = 1 \land Y = 2\ \}$$

$$\{\ X = 1\ \}\ Y := X + 1\ \{\ X = 1 \land Y = 2\ \}$$

$$\{\ X = 1\ \}\ Y := 2 * X\ \{\ X = 1 \land Y = 2\ \}$$

A slightly more complex example (factorial):

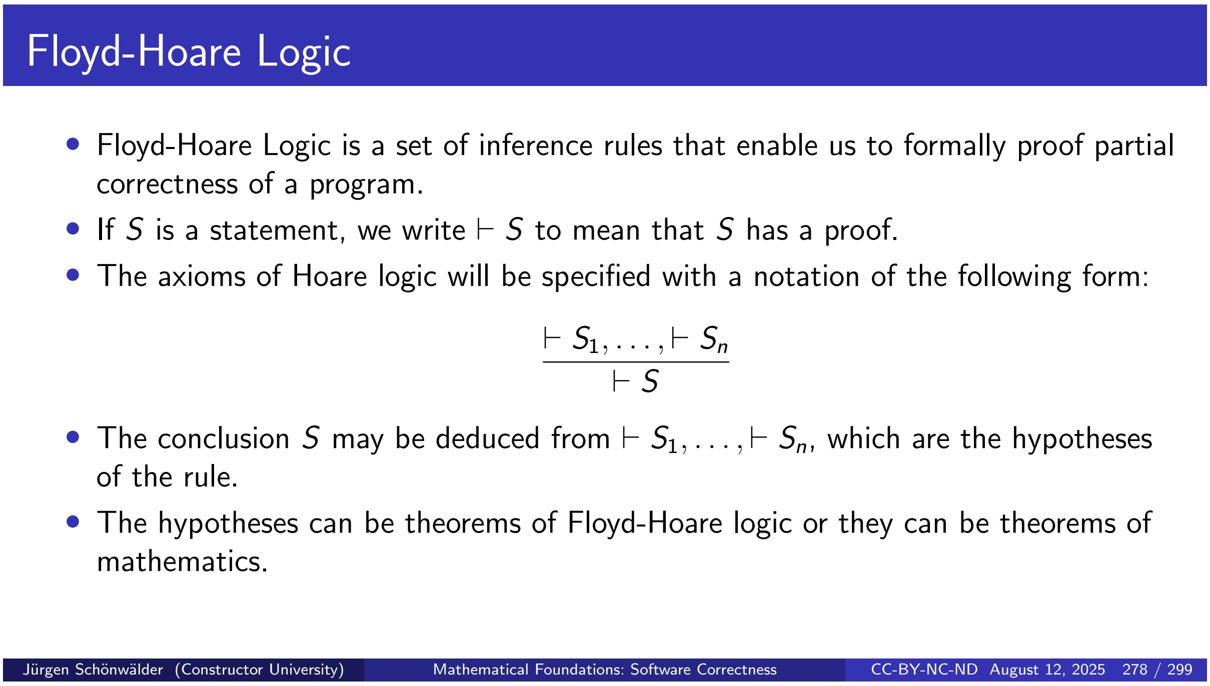
```
Precondition: $\{X > 0 \land X = x\}$
$F := 1$
while $X > 0$ do
    $F := F \cdot X$
    $X := X - 1$
od
Postcondition: $\{F = x!\}$
```

Section 34: Software Verification

33 Software Specification

34 Software Verification

35 Automation of Software Verification



Floyd-Hoare Logic

- Floyd-Hoare Logic is a set of inference rules that enable us to formally proof partial correctness of a program.
- If S is a statement, we write $\vdash S$ to mean that S has a proof.
- The axioms of Hoare logic will be specified with a notation of the following form:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

- The conclusion S may be deduced from $\vdash S_1, \ldots, \vdash S_n$, which are the hypotheses of the rule.
- The hypotheses can be theorems of Floyd-Hoare logic or they can be theorems of mathematics.

So far we have discussed the formal specification of software using preconditions and postconditions and we have introduced a simple imperative programming language consisting essentially of variables, expressions and variable assignments, a conditional command, a loop command, and command sequences. The next step is to define inference rules that allow us to make inferences over the commands of this simple programming language. This will give us a formal framework to prove that a program processing input satisfying the precondition will produce a result satisfying the postcondition.

Floyd-Hoare logic is a deductive proof system for Floyd-Hoare triples. It can be used to extract verification conditions (VCs), which are proof obligations or proof subgoals that must be proven so that $\{ P \} C \{ Q \}$ is true.

Precondition Strengthening

- If P implies P' and we have shown $\{P'\}\ C\ \{Q\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash P \rightarrow P', \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

- Example: Since $\vdash X = n \rightarrow X + 1 = n + 1$, we can strengthen

$$\vdash \{\ X + 1 = n + 1\ \}\ X := X + 1\ \{\ X = n + 1\ \}$$

to

$$\vdash \{\ X = n\ \}\ X := X + 1\ \{\ X = n + 1\ \}.$$

The precondition P is stronger than P' ($P \rightarrow P'$) if the set of states $\{s|s \vdash P\} \subseteq \{s|s \vdash P'\}$.

Precondition strengthening applied to the assignment axiom gives us a triple that feels more intuitive. But keep in mind that $\vdash \{\ X = n\ \}\ X := X + 1\ \{\ X = n + 1\ \}$ has been derived by combining the assignment axiom with precondition strengthening.

Postcondition Weakening

- If Q' implies Q and we have shown $\{P\}\ C\ \{Q'\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \quad \vdash Q' \rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

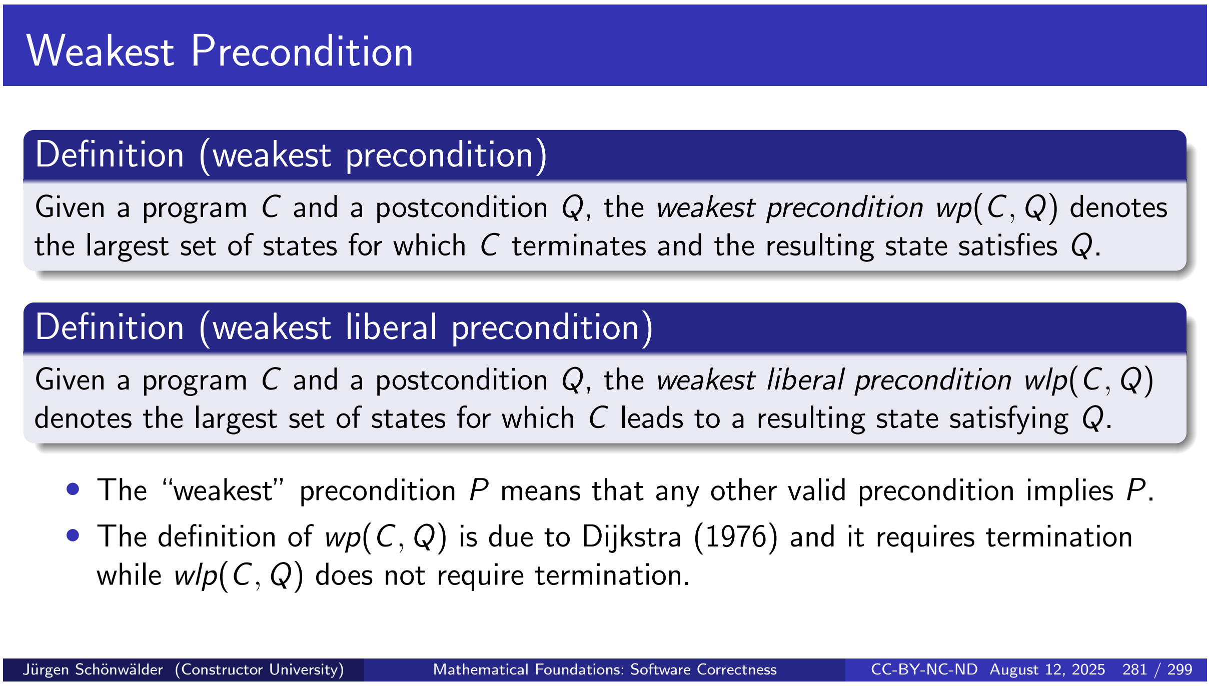
- Example: Since $X = n + 1 \rightarrow X > n$, we can weaken

$$\vdash \{\ X = n\ \}\ X := X + 1\ \{\ X = n + 1\ \}$$

to

$$\vdash \{\ X = n\ \}\ X := X + 1\ \{\ X > n\ \}$$

The postcondition Q is weaker than Q' ($Q' \rightarrow Q$) if the set of states $\{s|s \vdash Q'\} \subseteq \{s|s \vdash Q\}$.



Weakest Precondition

Definition (weakest precondition)

Given a program C and a postcondition Q, the *weakest precondition* $wp(C, Q)$ denotes the largest set of states for which C terminates and the resulting state satisfies Q.

Definition (weakest liberal precondition)

Given a program C and a postcondition Q, the *weakest liberal precondition* $wlp(C, Q)$ denotes the largest set of states for which C leads to a resulting state satisfying Q.

- The “weakest” precondition P means that any other valid precondition implies P.
- The definition of $wp(C, Q)$ is due to Dijkstra (1976) and it requires termination while $wlp(C, Q)$ does not require termination.

In Hoare Logic, we can usually define many valid preconditions. For example, all of the following are valid Hoare triples:

$$\vdash \{\, X = 1 \,\}\; X := X + 1 \;\{\, X > 0 \,\}$$

$$\vdash \{\, X > 0 \,\}\; X := X + 1 \;\{\, X > 0 \,\}$$

$$\vdash \{\, X > -1 \,\}\; X := X + 1 \;\{\, X > 0 \,\}$$

Obviously, the second preconditions is weaker than the first since $X = 1$ implies $X > 0$. With a similar argument, the third precondition is weaker than the second since $X > 0$ implies $X > -1$. How does the precondition $X = 0$ compare to the second and third alternative?

The weakest liberal precondition for $X := X + 1$ and the postcondition $X > 0$ is:

$$wlp(X := X + 1, X > 0) = (X > -1)$$

Since we can assume that the assignment always terminates in this specific case, we have:

$$wp(X := X + 1, X > 0) = wlp(X := X + 1, X > 0) = (X > -1)$$

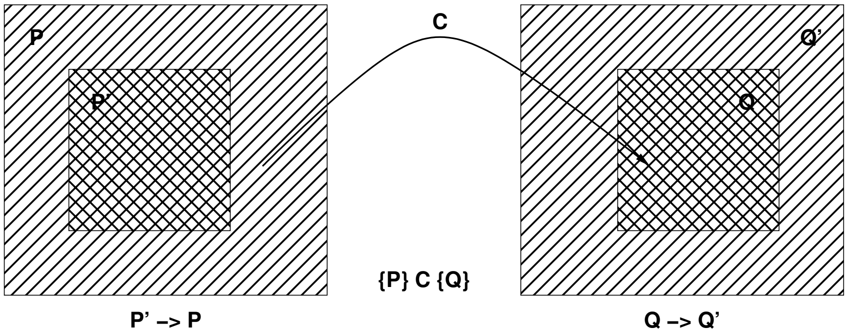
Strongest Postcondition

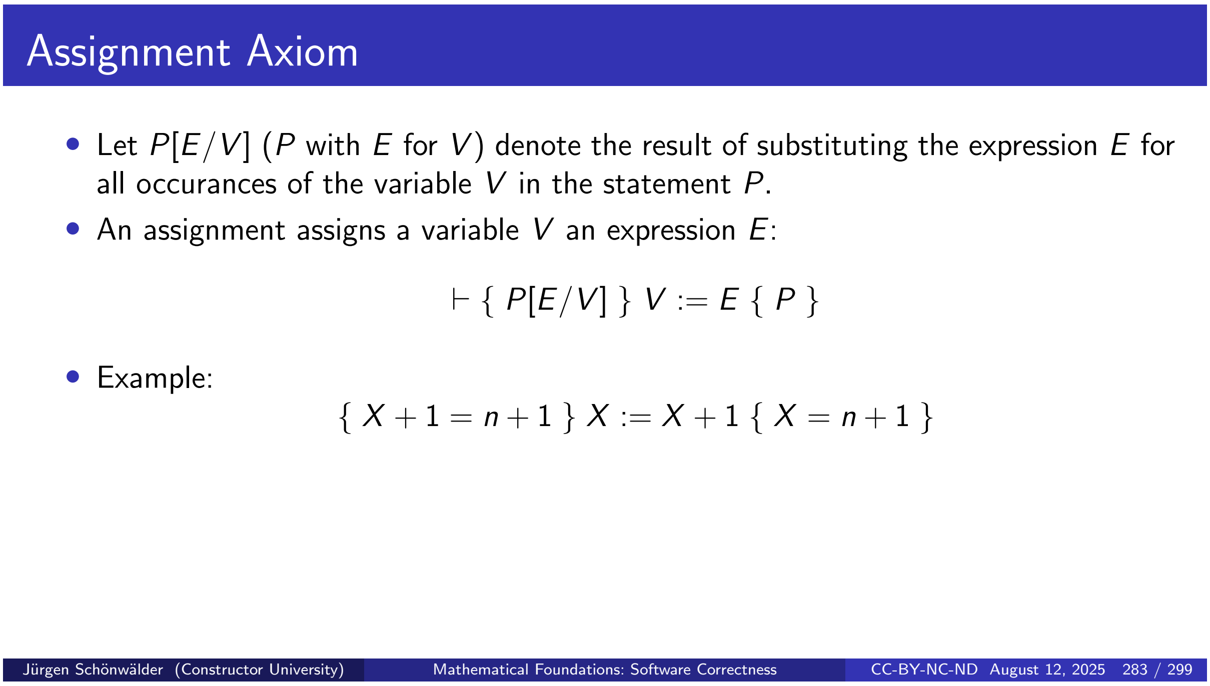
Definition (strongest postcondition)

Given a program C and a precondition P, the *strongest postcondition* $sp(C,P)$ has the property that $\vdash \{\ P\ \}\ C\ \{\ sp(C,P)\ \}$ and for any Q with $\vdash \{\ P\ \}\ C\ \{\ Q\ \}$, we have $\vdash sp(C,P) \rightarrow Q$.

- The "strongest" postcondition Q means that any other valid postcondition is implied by Q (via postcondition weakening).

Our goal is to find the weakest precondition and the strongest postcondition. This translates into finding the largest set of states that are valid inputs of the computation C and the smallest set of possible result states. Finding $P = wp(C,Q)$ makes it easy to cover any P' for which $P' \rightarrow P$ holds. Similarly, finding $Q = sp(C,P)$ makes it easy to cover any Q' for which $Q \rightarrow Q'$ holds.





Assignment Axiom

- Let $P[E/V]$ (P with E for V) denote the result of substituting the expression E for all occurances of the variable V in the statement P.
- An assignment assigns a variable V an expression E:

$$\vdash \{\ P[E/V]\ \}\ V := E\ \{\ P\ \}$$

- Example:

$$\{\ X + 1 = n + 1\ \}\ X := X + 1\ \{\ X = n + 1\ \}$$

The assignment axiom kind of works backwards. In the example, we start with P, which is $\{X = n+1\}$. In P, we substitute E, which is $X+1$, for V, which is X. This gives us $\{X+1 = n+1\}$.

Note that the term E is evaluated in a state where the assignment has not yet been carried out. Hence, if a statement P is true after the assignment, then the statement obtained by substituting E for V in P must be true before the assignment.

Two common erroneous intuitions:

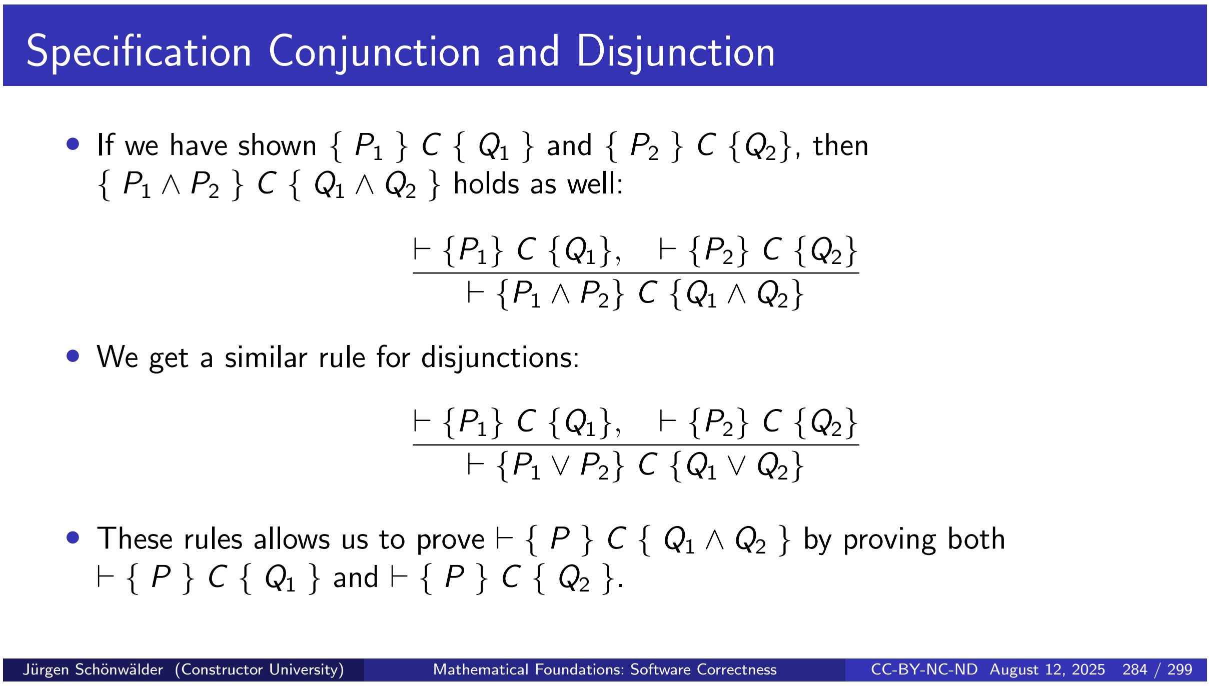
1. $\vdash \{P\}\ V := E\ \{P[V/E]\}$

   This has the consequence $\vdash \{\ X = 0\ \}\ X := 1\ \{\ X = 0\ \}$ since $X = 0[X/1]$ is equal to $X = 0$ (since 1 does not occur in $X = 0$).

2. $\vdash \{P\}\ V := E\ \{P[E/V]\}$

   This has the consequence $\vdash \{\ X = 0\ \}\ X := 1\ \{\ 1 = 0\ \}$ since one would substitute X with 1 in $X = 0$.

Warning: An important assumption here is that expressions have no side effects that modify the program state. The assignment axiom depends on this property. (Many real-world programming languages, however, do allow side effects.) To see why side effects cause problems, consider an expression $(C; E)$ that consists of a command C and an expression E, e.g. $(Y := 1; 2)$. With this, we would get $\vdash \{\ Y = 0\ \}\ X := (Y := 1; 2)\ \{\ Y = 0\ \}$ (the substitution would not affect Y).



Specification Conjunction and Disjunction

- If we have shown $\{\ P_1\ \}\ C\ \{\ Q_1\ \}$ and $\{\ P_2\ \}\ C\ \{Q_2\}$, then $\{\ P_1 \land P_2\ \}\ C\ \{\ Q_1 \land Q_2\ \}$ holds as well:

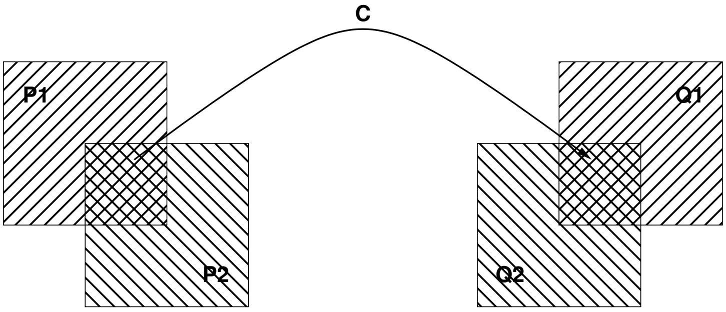
$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \land P_2\}\ C\ \{Q_1 \land Q_2\}}$$

- We get a similar rule for disjunctions:

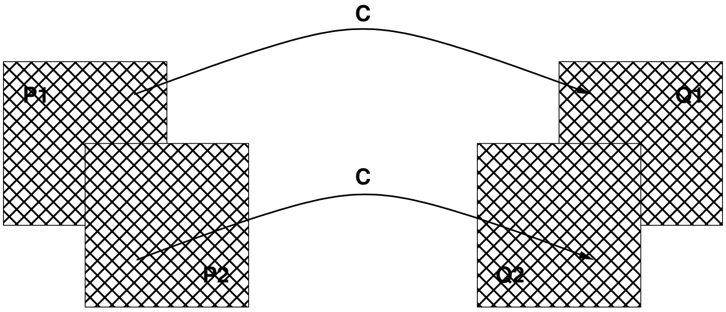
$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \lor P_2\}\ C\ \{Q_1 \lor Q_2\}}$$

- These rules allows us to prove $\vdash \{\ P\ \}\ C\ \{\ Q_1 \land Q_2\ \}$ by proving both $\vdash \{\ P\ \}\ C\ \{\ Q_1\ \}$ and $\vdash \{\ P\ \}\ C\ \{\ Q_2\ \}$.

The rules can be easily understood by looking at the sets of states satisfying the predicates. The conjunction $P_1 \land P_2$ translates to the intersection of the corresponding sets of states and in the same way the conjunction $Q_1 \land Q_2$ translates to the intersection of the corresponding sets of states. The rule then follows directly from the fact that $\{\ P_1 \land P_2\ \}\ C\ \{\ Q_1 \land Q_2\ \}$ maps from the intersection to the intersection.



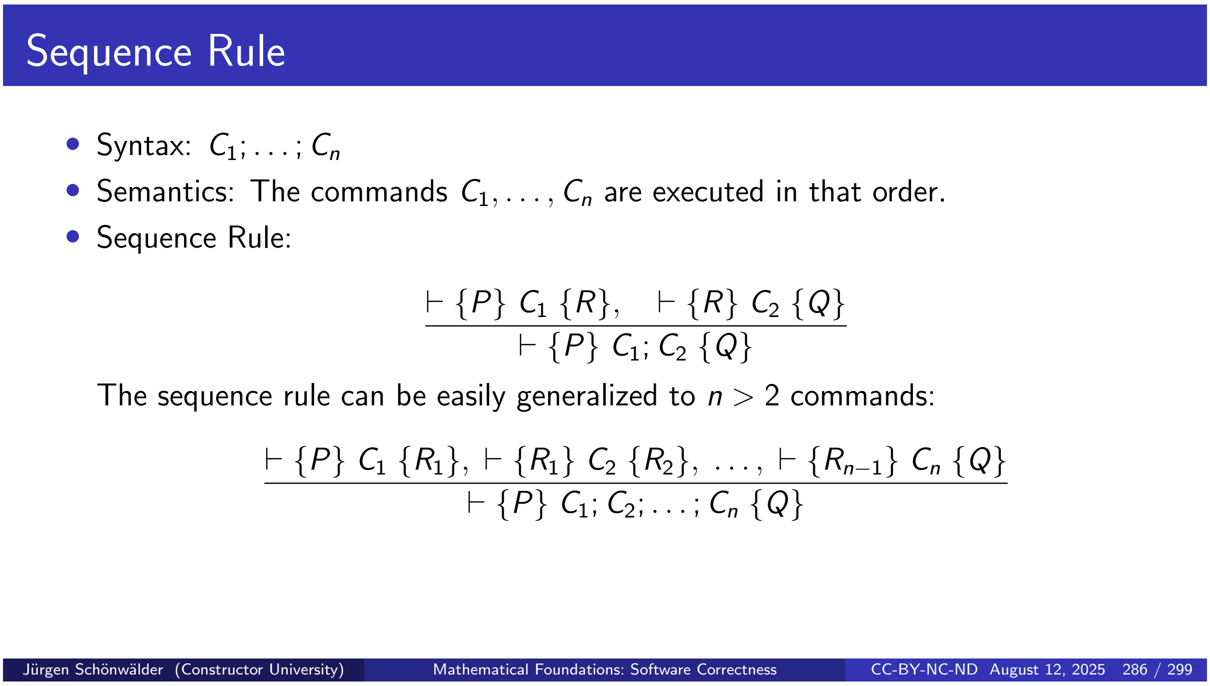
The disjunction $P_1 \lor P_2$ translates to the union of the sets of states associated with P_1 and P_2 and the disjunction $Q_1 \lor Q_2$ to the union of the sets of states associated with Q_1 and Q_2. In order to cover the entire space, it is necessary that both $\{\ P_1\ \}\ C\ \{\ Q_1\ \}$ and $\{\ P_2\ \}\ C\ \{\ Q_2\ \}$ are true.



Skip Command Rule

- Syntax: *SKIP*
- Semantics: Do nothing. The state after executing the *SKIP* command is the same as the state before executing the command *SKIP*.
- Skip Command Rule:

$$\frac{}{\vdash \{P\}\ \mathit{SKIP}\ \{P\}}$$



Sequence Rule

- Syntax: $C_1; \dots; C_n$
- Semantics: The commands $C_1, \dots, C_n$ are executed in that order.
- Sequence Rule:

$$\frac{\vdash \{P\}\ C_1\ \{R\}, \quad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

The sequence rule can be easily generalized to $n > 2$ commands:

$$\frac{\vdash \{P\}\ C_1\ \{R_1\}, \vdash \{R_1\}\ C_2\ \{R_2\},\ \dots, \vdash \{R_{n-1}\}\ C_n\ \{Q\}}{\vdash \{P\}\ C_1; C_2; \dots; C_n\ \{Q\}}$$

Example (swapping two numbers):

```
Precondition: $\{ X = x \wedge Y = y \}$
  $R := X$
  $X := Y$
  $Y := R$
Postcondition: $\{ X = y \wedge Y = x \}$
```

The proof of the correctness of the sequence of assignments is broken down into the following steps:

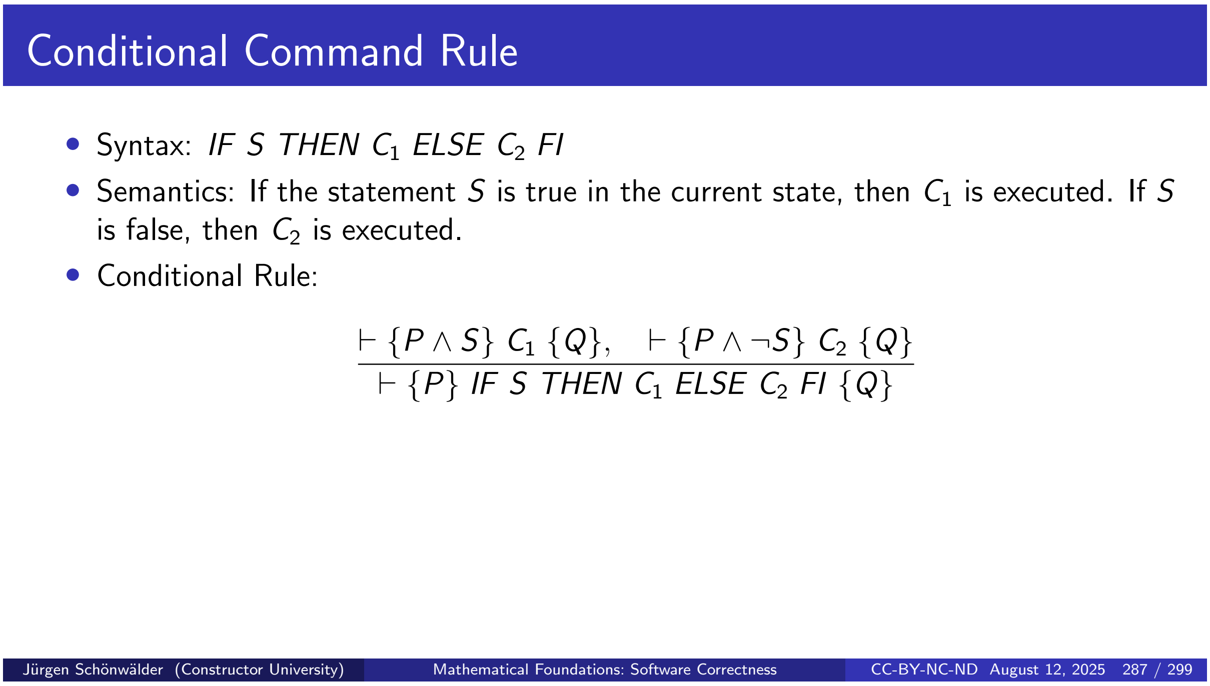
(i) $\vdash \{ X = x \wedge Y = y \}\ R := X\ \{ R = x \wedge Y = y \}$ (assignment axiom)

(ii) $\vdash \{ R = x \wedge Y = y \}\ X := Y\ \{ R = x \wedge X = y \}$ (assignment axiom)

(iii) $\vdash \{ R = x \wedge X = y \}\ Y := R\ \{ Y = x \wedge X = y \}$ (assignment axiom)

(iv) $\vdash \{ X = x \wedge Y = y \}\ R := X; X := Y\ \{ R = x \wedge X = y \}$ (sequence rule for (i) and (ii))

(v) $\vdash \{ X = x \wedge Y = y \}\ R := X; X := Y; Y := R\ \{ Y = x \wedge X = y \}$ (sequence rule for (iv) and (iii))



Conditional Command Rule

- Syntax: *IF S THEN C_1 ELSE C_2 FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Conditional Rule:

$$\frac{\vdash \{P \land S\}\ C_1\ \{Q\}, \quad \vdash \{P \land \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ \{Q\}}$$

Consider the following specification and program (max):

```
Precondition: { $X = x \land Y = y$ }
if $X \geq Y$ then
    $M := X$
else
    $M := Y$
fi
Postcondition: { $M = max(x, y)$ }
```

In order to prove the partial correctness of this program, we have to prove the correctness of the two assignments under the statement $X \geq Y$ being either true or false. The application of the assignment axiom gives us the following two statements:

$$\{\ X = x \land Y = y \land X \geq Y\ \}\ M := X\ \{\ M = x \land X = x \land Y = y \land X \geq Y\ \}$$

$$\{\ X = x \land Y = y \land X < Y\ \}\ M := Y\ \{\ M = y \land X = x \land Y = y \land X < Y\ \}$$

The definition of $max(x, y)$ we are going to use is the following:

$$max(x, y) = \begin{cases} x & x \geq y \\ y & x < y \end{cases}$$

This gives us the following implications:

$$M = x \land X = x \land Y = y \land X \geq Y \rightarrow M = max(x, y)$$

$$M = y \land X = x \land Y = y \land X < Y \rightarrow M = max(x, y)$$

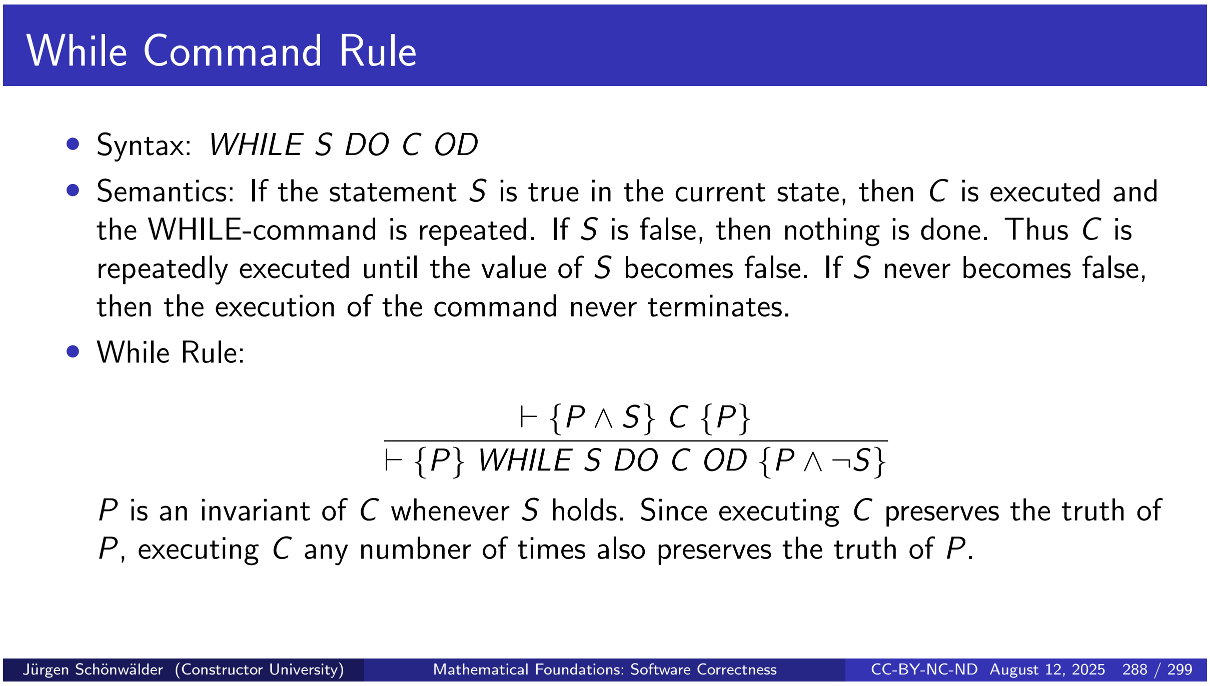
Postcondition weakening gives us:

$$\{\ X = x \land Y = y \land X \geq Y\}\ M := X\ \{M = max(x, y)\ \}$$

$$\{\ X = x \land Y = y \land X < Y\}\ M := Y\ \{M = max(x, y)\ \}$$

Applying the conditional rule, we get:

$$\{\ X = x \land Y = y\ \}\ \text{IF}\ X \geq Y\ \text{THEN}\ M := X\ \text{ELSE}\ M := Y\ \text{FI}\ \{\ M = max(x, y)\ \}$$



While Command Rule

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- While Rule:

$$\frac{\vdash \{P \land S\}\ C\ \{P\}}{\vdash \{P\}\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ \{P \land \neg S\}}$$

P is an invariant of C whenever S holds. Since executing C preserves the truth of P, executing C any numbner of times also preserves the truth of P.

Finding invariants is the key to prove the correctness of while loops. The invariant should

- say what has been done so far together with what remains to be done;
- hold at each iteration of the loop;
- give the desired result when the loop terminates.

Example (factorial):

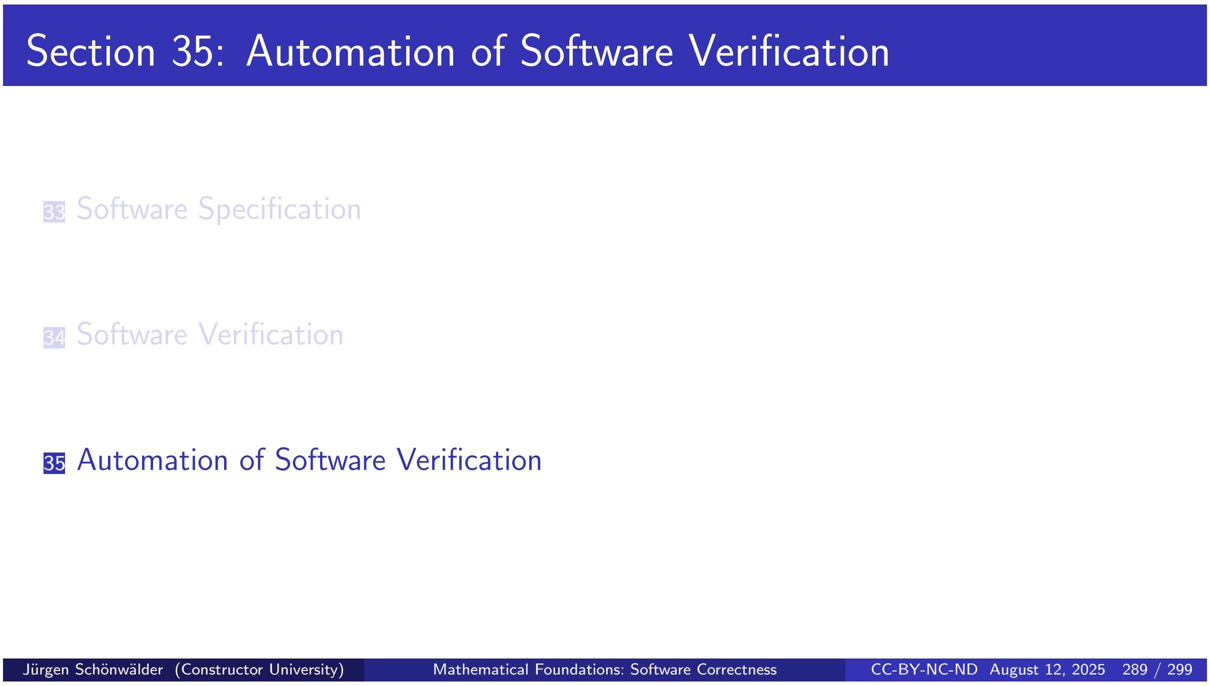
```
Precondition: $\{ Y = 1 \land Z = 0 \land X = x \land X \geq 0 \}$
while $Z \neq X$ do
    $Z := Z + 1$
    $Y := Y * Z$
od
Postcondition: $\{ Y = x! \}$
```

We need to find an invariant P such that:

- $\{ P \land Z \neq X \}\ Z := Z + 1;\ Y := Y * Z\ \{ P \}$ (while rule)
- $Y = 1 \land Z = 0 \rightarrow P$ (precondition strengthening)
- $P \land \neg(Z \neq X) \rightarrow Y = X!$ (postcondition weakening)

The invariant $Y = Z!$ serves the purpose:

- $Y = Z! \land Z \neq X \rightarrow Y \cdot (Z + 1) = (Z + 1)!$
  $\{ Y \cdot (Z + 1) = (Z + 1)! \}\ Z := Z + 1\ \{ Y \cdot Z = Z! \}$ (assignment axiom)
  $\{ Y \cdot Z = Z! \}\ Y := Y * Z\ \{ Y = Z! \}$ (assignment axiom)
  $\{ Y = Z! \}\ Z := Z + 1;\ Y := Y * Z\ \{ Y = Z! \}$ (sequence rule)
- $Y = 1 \land Z = 0 \rightarrow Y = Z!$ since $0! = 1$
- $Y = Z! \land \neg(Z \neq X) \rightarrow Y = X!$ since $\neg(Z \neq X)$ is equivalent to $Z = X$

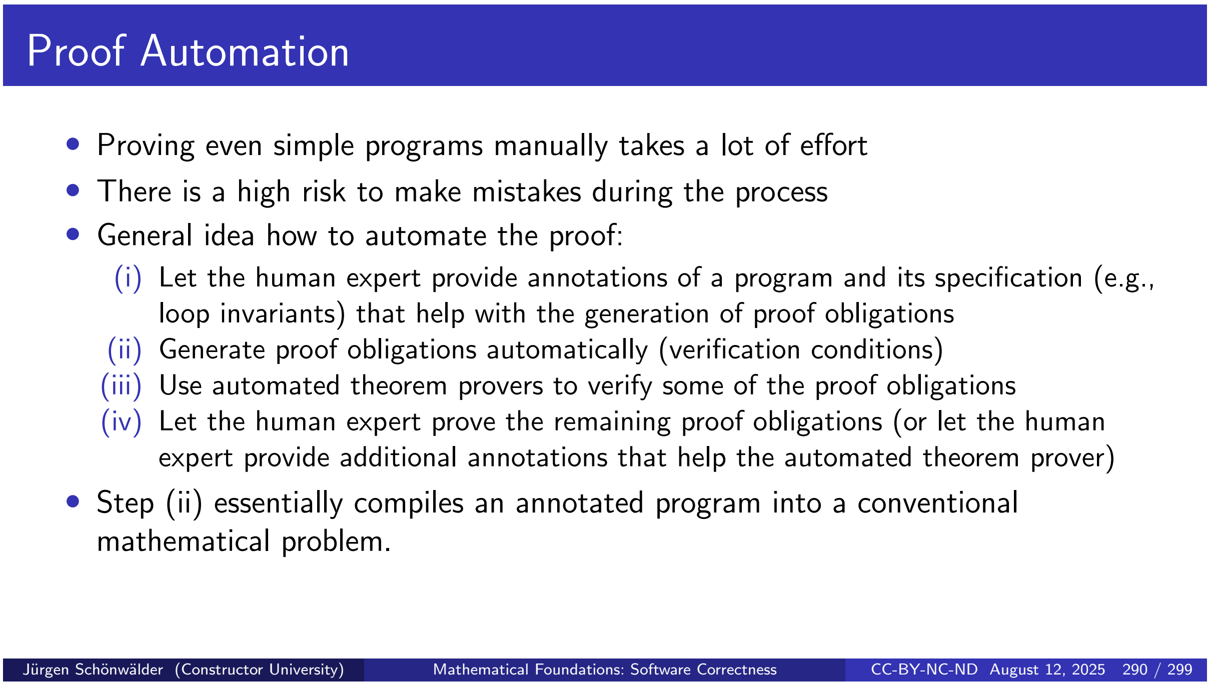


Section 35: Automation of Software Verification

33 Software Specification

34 Software Verification

35 Automation of Software Verification

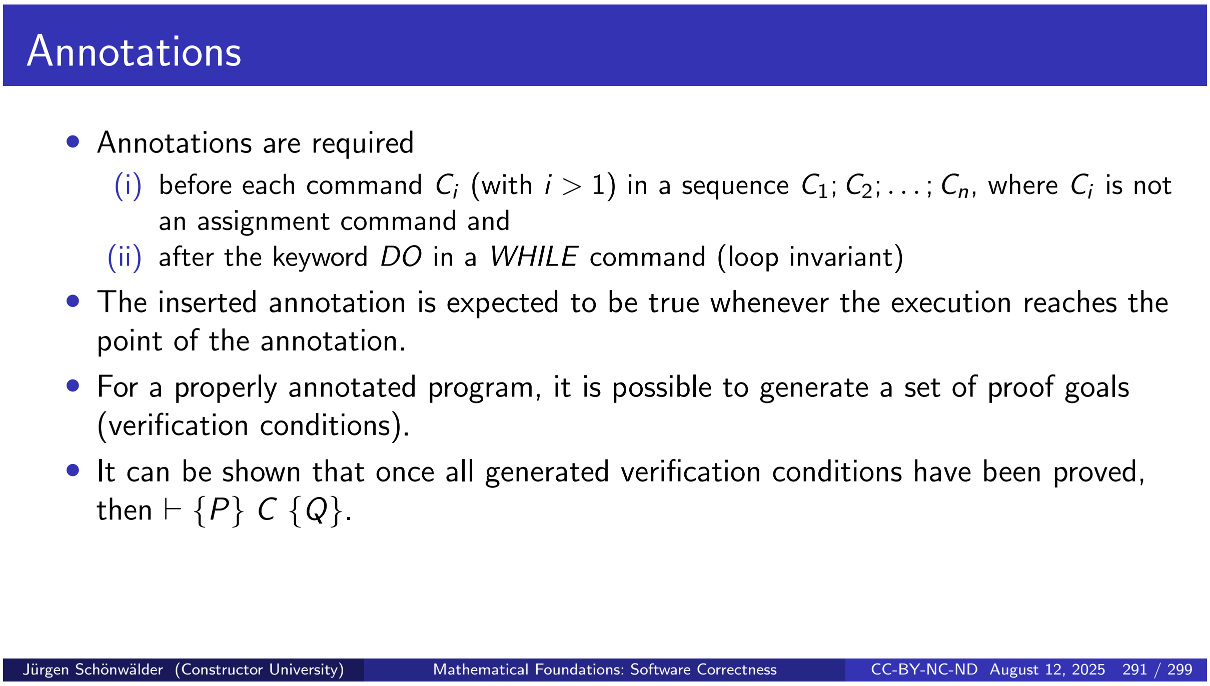


Proof Automation

- Proving even simple programs manually takes a lot of effort
- There is a high risk to make mistakes during the process
- General idea how to automate the proof:
  (i) Let the human expert provide annotations of a program and its specification (e.g., loop invariants) that help with the generation of proof obligations
  (ii) Generate proof obligations automatically (verification conditions)
  (iii) Use automated theorem provers to verify some of the proof obligations
  (iv) Let the human expert prove the remaining proof obligations (or let the human expert provide additional annotations that help the automated theorem prover)
- Step (ii) essentially compiles an annotated program into a conventional mathematical problem.

Consider the following program:

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
while $Y \leq R$ do
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$
```



Annotations

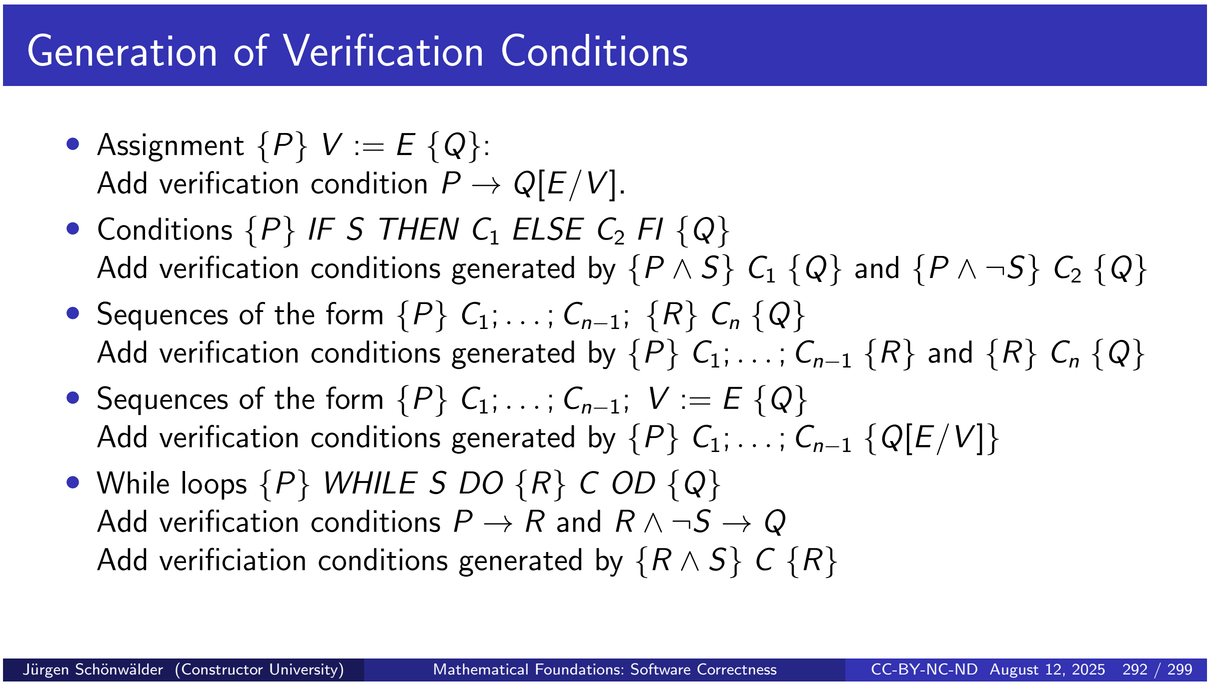
- Annotations are required
  - (i) before each command C_i (with $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$, where C_i is not an assignment command and
  - (ii) after the keyword *DO* in a *WHILE* command (loop invariant)
- The inserted annotation is expected to be true whenever the execution reaches the point of the annotation.
- For a properly annotated program, it is possible to generate a set of proof goals (verification conditions).
- It can be shown that once all generated verification conditions have been proved, then $\vdash \{P\}\ C\ \{Q\}$.

We add suitable annotations:

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \land Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$
```

This should (ideally automatically) lead to the following proof obligations (verification conditions):

1. $\top \rightarrow (X = X \land 0 = 0)$
2. $(R = X \land Q = 0) \rightarrow (X = Y \cdot Q + R)$
3. $(X = Y \cdot Q + R \land \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \land R < Y)$
4. $(X = Y \cdot Q + R \land Y \leq R) \rightarrow (X = Y \cdot (Q + 1) + (R - Y)$



Generation of Verification Conditions

- Assignment $\{P\}\ V := E\ \{Q\}$:
  Add verification condition $P \rightarrow Q[E/V]$.
- Conditions $\{P\}$ *IF S THEN* C_1 *ELSE* C_2 *FI* $\{Q\}$
  Add verification conditions generated by $\{P \land S\}\ C_1\ \{Q\}$ and $\{P \land \neg S\}\ C_2\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ \{R\}\ C_n\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{R\}$ and $\{R\}\ C_n\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ V := E\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{Q[E/V]\}$
- While loops $\{P\}$ *WHILE S DO* $\{R\}$ *C OD* $\{Q\}$
  Add verification conditions $P \rightarrow R$ and $R \land \neg S \rightarrow Q$
  Add verificiation conditions generated by $\{R \land S\}\ C\ \{R\}$

Starting with the annotated example:

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \land Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$
```

According to the second sequence rule, we have to generate VCs for the while loop and the sequence consisting of the initial assignments. The initial assignments reduce to $\top \rightarrow (X = X \land 0 = 0)$ as follows:

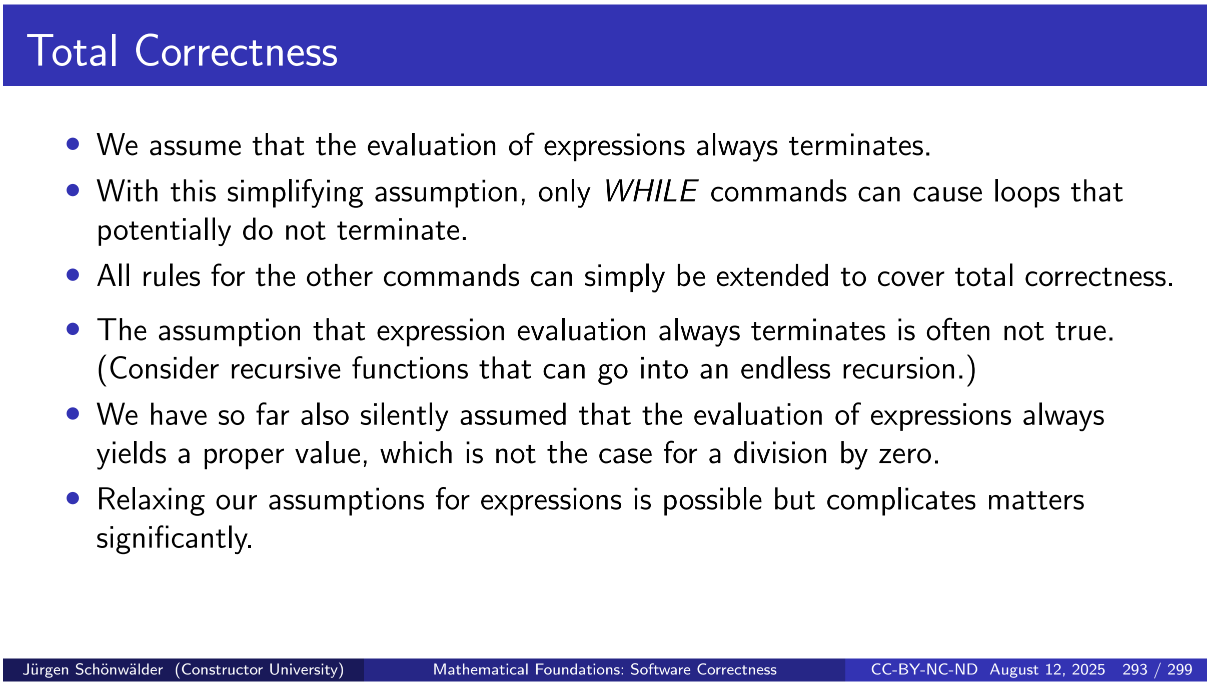
$$\{\top\}\ R := X;\ Q := 0\ \{R = X \land Q = 0\}$$
$$\{\top\}\ R := X\ \{R = X \land 0 = 0\}$$
$$\top \rightarrow (X = X \land 0 = 0)$$

The while loop rule gives us the following two VCs

$$(R = X \land Q = 0) \rightarrow (X = Y \cdot Q + R)$$
$$(X = Y \cdot Q + R \land \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \land R < Y)$$

and the VC generated as follows:

$$\{X = Y \cdot Q + R \land Y \leq R\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R\}$$
$$\{X = Y \cdot Q + R \land Y \leq R\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R\}$$
$$(X = Y \cdot Q + R \land Y \leq R) \rightarrow (X = Y \cdot (Q + 1) + (R - Y))$$



Total Correctness

- We assume that the evaluation of expressions always terminates.
- With this simplifying assumption, only *WHILE* commands can cause loops that potentially do not terminate.
- All rules for the other commands can simply be extended to cover total correctness.
- The assumption that expression evaluation always terminates is often not true. (Consider recursive functions that can go into an endless recursion.)
- We have so far also silently assumed that the evaluation of expressions always yields a proper value, which is not the case for a division by zero.
- Relaxing our assumptions for expressions is possible but complicates matters significantly.

If C does not contain any while commands, then we have the simple rule:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [1/4]

- Assignment axiom

$$\vdash [P[E/V]]\ V := E\ [P]$$

- Precondition strengthening

$$\frac{\vdash P \rightarrow P', \quad \vdash [P']\ C\ [Q]}{\vdash [P]\ C\ [Q]}$$

- Postcondition weakening

$$\frac{\vdash [P]\ C\ [Q'], \quad \vdash Q' \rightarrow Q}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [2/4]

- Specification conjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \land P_2]\ C\ [Q_1 \land Q_2]}$$

- Specification disjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \lor P_2]\ C\ [Q_1 \lor Q_2]}$$

- Skip command rule

$$\frac{}{[P]\ \mathit{SKIP}\ [P]}$$

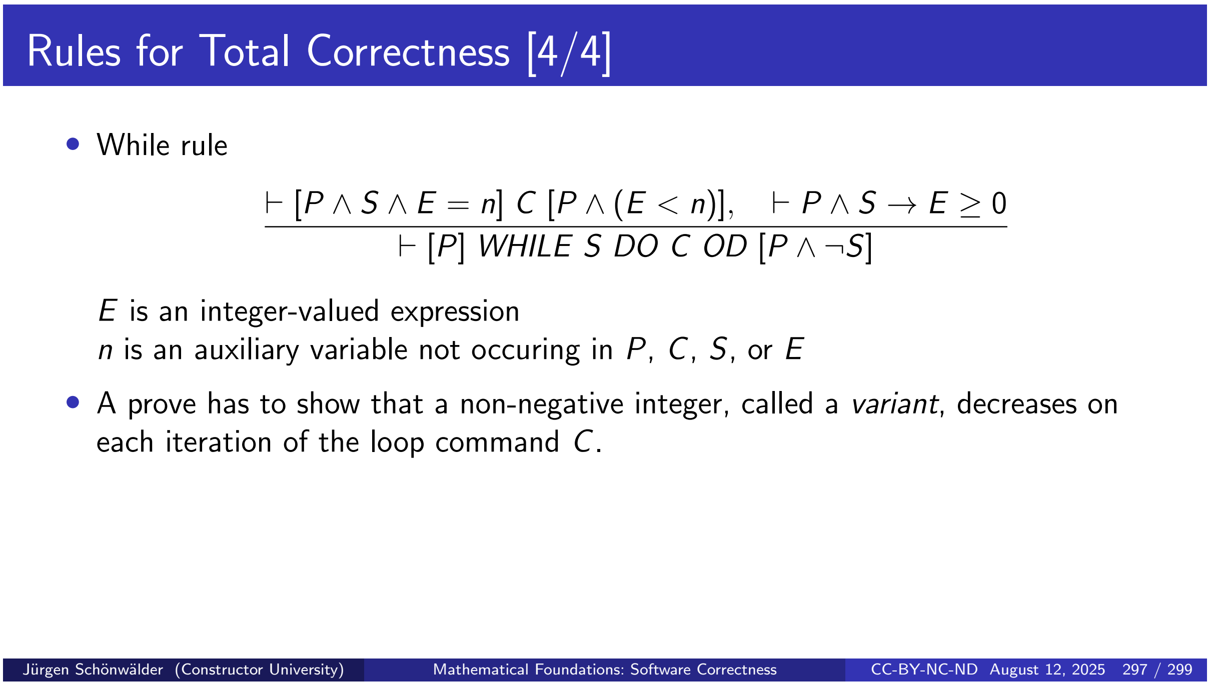
Rules for Total Correctness [3/4]

- Sequence rule

$$\frac{\vdash [P]\ C_1\ [R_1],\ \vdash [R_1]\ C_2\ [R_2],\ \ldots,\ \vdash [R_{n-1}]\ C_n\ [Q]}{\vdash [P]\ C_1; C_2; \ldots; C_n\ [Q]}$$

- Conditional rule

$$\frac{\vdash [P \land S]\ C_1\ [Q], \quad \vdash [P \land \neg S]\ C_2\ [Q]}{\vdash [P]\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ [Q]}$$



Rules for Total Correctness [4/4]

- While rule

$$\frac{\vdash [P \land S \land E = n]\ C\ [P \land (E < n)], \quad \vdash P \land S \rightarrow E \geq 0}{\vdash [P]\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ [P \land \neg S]}$$

  E is an integer-valued expression
  n is an auxiliary variable not occuring in P, C, S, or E

- A prove has to show that a non-negative integer, called a *variant*, decreases on each iteration of the loop command C.

We show that the while loop in the following program terminates.

```
Precondition: ${\top}$
$R := X$
$Q := 0$
while $Y \leq R$ do
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: ${X = Y \cdot Q + R \land R < Y}$
```

We apply the while rule with

$$P = Y > 0$$
$$S = Y \leq R$$
$$E = R$$

and we have to show the following to be true:

1. $[P \land S \land E = n]\ R := R - Y; Q := Q + 1\ [P \land (E < n)]$

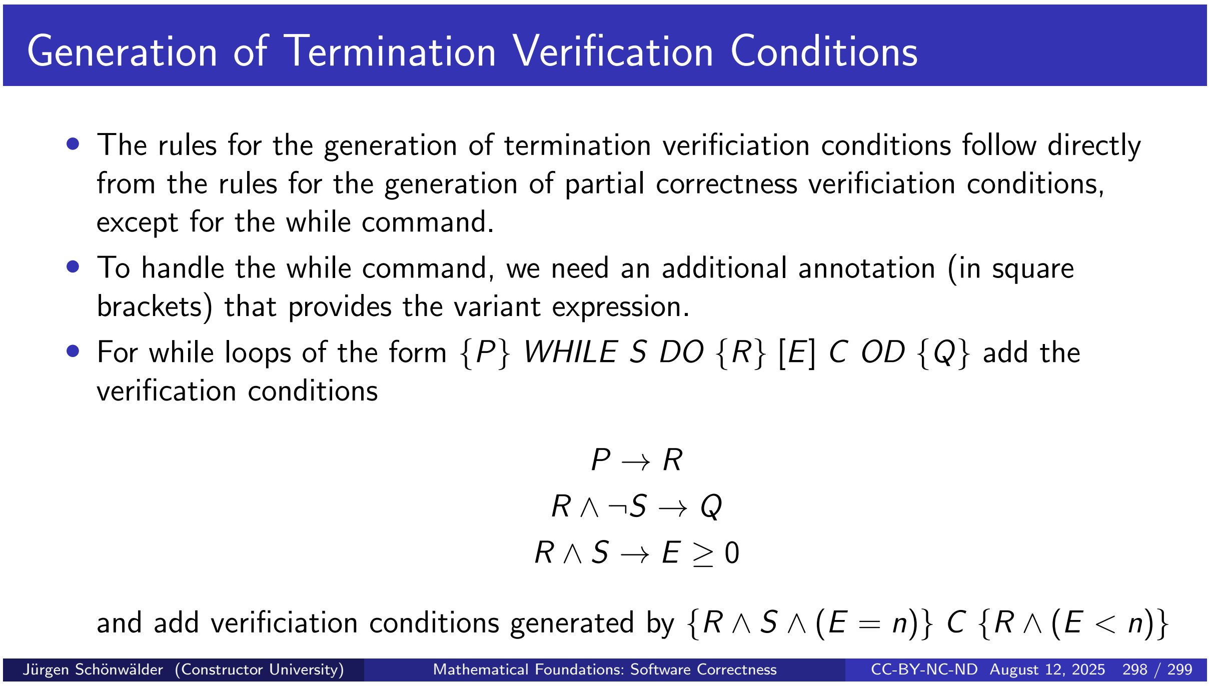
   This follows from the following derivation:

$$[P \land S \land E = n]\ R := R - Y; Q := Q + 1\ [P \land (E < n)]$$
$$[Y > 0 \land Y \leq R \land R = n]\ R := R - Y; Q := Q + 1\ [Y > 0 \land (R < n)]$$
$$Y > 0 \land Y \leq R \land R = n \rightarrow Y > 0 \land (R < n)[Q + 1/Q][R - Y/R]$$
$$Y > 0 \land Y \leq R \land R = n \rightarrow Y > 0 \land ((R - Y) < n)$$

2. $P \land S \rightarrow E \geq 0$

   This follows from:

$$P \land S \rightarrow E \geq 0$$
$$Y > 0 \land Y \leq R \rightarrow R > 0$$



Generation of Termination Verification Conditions

- The rules for the generation of termination verificiation conditions follow directly from the rules for the generation of partial correctness verificiation conditions, except for the while command.
- To handle the while command, we need an additional annotation (in square brackets) that provides the variant expression.
- For while loops of the form $\{P\}$ *WHILE S DO* $\{R\}$ $[E]$ C *OD* $\{Q\}$ add the verification conditions

$$\begin{gathered} P \to R \\ R \land \neg S \to Q \\ R \land S \to E \geq 0 \end{gathered}$$

and add verificiation conditions generated by $\{R \land S \land (E = n)\}$ C $\{R \land (E < n)\}$

Annotated example including the variant annotation for termination verification rule generation:

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \land Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $[R]$
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$
```

The while loop rule gives use the following termination VCs

$$\begin{aligned} &(R = X \land Q = 0) \to (X = Y \cdot Q + R) \\ &(X = Y \cdot Q + R \land \neg(Y \leq R)) \to (X = Y \cdot Q + R \land R < Y) \\ &(X = Y \cdot Q + R \land (Y \leq R)) \to R \geq 0 \end{aligned}$$

and the VC generated as follows:

$$\begin{aligned} &\{X = Y \cdot Q + R \land Y \leq R \land R = n\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R \land R < n\} \\ &\{X = Y \cdot Q + R \land Y \leq R \land R = n\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R \land R < n\} \\ &(X = Y \cdot Q + R \land Y \leq R \land R = n) \to (X = Y \cdot (Q + 1) + (R - Y) \land (R - Y) < n) \end{aligned}$$

The last VC is not true in general and hence the algorithm does not always terminate:

$Y = 0:$

$$((X = R \land 0 \leq R \land R = n) \to (X = R \land R < n)) \to \bot$$

$Y < 0:$

$$((X = Y \cdot Q + R \land Y \leq R \land R = n) \to (X = Y \cdot (Q + 1) + (R - Y) \land (R - Y) < n)) \to \bot$$

Termination and Correctness

- Partial correctness and termination implies total correctness:

$$\frac{\vdash \{P\}\ C\ \{Q\}, \quad \vdash [P]\ C\ [\mathbf{T}]}{\vdash [P]\ C\ [Q]}$$

- Total correctness implies partial correctness and termination:

$$\frac{\vdash [P]\ C\ [Q]}{\vdash \{P\}\ C\ \{Q\}, \quad \vdash [P]\ C\ [\mathbf{T}]}$$

References

[1] BIPM. The international system of units. Technical Report 8th edition, Bureau International des Poids et Mesures, 2014.

[2] BIPM. The international system of units. Technical Report 9th edition, Bureau International des Poids et Mesures, 2019.

[3] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[4] Peter J. Denning, Matti Tedre, and Pat Yongpradit. Misconceptions about computer science. *Communications of the ACM*, 60(3), February 2017.

[5] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, Alis Technologies, November 2003.

[6] G. Futschek. Algorithmic Thinking: The Key for Understanding Computer Science. In *Informatics Education – The Bridge between Using and Understanding Computers, Lecture Notes in Computer Science 4226*. Springer, 2006.

[7] Zvi Galil. On improving the worst case running time of the boyer-moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, September 1979.

[8] R. Hammack. *Book of Proof*. 3 edition, 2019.

[9] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[10] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339, Clearswift Corporation, Sun Microsystems, July 2002.

[11] D. E. Knuth. *The Art of Computer Programming*, volume 1, Fundamental Algorithms. Addison Wesley, 3 edition, 1997.

[12] D. E. Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching. Addison Wesley, 2 edition, 1998.

[13] D. E. Knuth. *The Art of Computer Programming*, volume 4a, Combinatorial Algorithms. Addison Wesley, 1 edition, 2011.

[14] D. E. Knuth. *The Art of Computer Programming*, volume 2, Semi Numerical Algorithms. Addison Wesley, 3 edition, 2014.

[15] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1), February 1956.

[16] E. Lehmann, F.T. Leighton, and A.R. Meyer. *Mathematics for Computer Science*. MIT Open Courseware, 2019.

[17] James W. McGuffee. Defining computer science. *ACM SIGSE Bulletin*, 32(2), June 2020.

[18] U. Sharma and C. Bormann. Date and time on the internet: Timestamps with additional information. RFC 9557, Igalia, Universität Bremen, April 2024.