Abstract

This memo contains annotated slides for the course “Introduction to Computer Science”. The material is inspired by the course material of Michael Kohlhase's course “General Computer Science”, Herbert Jaeger's short course on “Boolean Logic”, and the online textbook “Mathematics for Computer Science” by Eric Lehman, F. Thomson Leighton, and Albert R. Meyer.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary Boolean Operations and Functions</td>
<td>121</td>
</tr>
<tr>
<td>Boolean Functions and Formulas</td>
<td>132</td>
</tr>
<tr>
<td>Boolean Algebra Equivalence Laws</td>
<td>138</td>
</tr>
<tr>
<td>Normal Forms (CNF and DNF)</td>
<td>143</td>
</tr>
<tr>
<td>Complexity of Boolean Formulas</td>
<td>150</td>
</tr>
<tr>
<td>Boolean Logic and the Satisfiability Problem</td>
<td>156</td>
</tr>
<tr>
<td><strong>V  Computer Architecture</strong></td>
<td>161</td>
</tr>
<tr>
<td>Logic Gates and Digital Circuits</td>
<td>162</td>
</tr>
<tr>
<td>Sequential Digital Circuits</td>
<td>173</td>
</tr>
<tr>
<td>Von Neumann Computer Architecture</td>
<td>183</td>
</tr>
<tr>
<td><strong>VI  System Software</strong></td>
<td>191</td>
</tr>
<tr>
<td>Interpreter and Compiler</td>
<td>192</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>204</td>
</tr>
<tr>
<td><strong>VII  Software Correctness</strong></td>
<td>213</td>
</tr>
<tr>
<td>Software Specification</td>
<td>214</td>
</tr>
<tr>
<td>Software Verification</td>
<td>226</td>
</tr>
<tr>
<td>Automation of Software Verification</td>
<td>238</td>
</tr>
</tbody>
</table>
Part I

Introduction

The aim of this part is to explain what computer science is all about. After the introduction of a few terms, 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 the problem from different perspectives in order to find good algorithms to solve the problem.
Section 1: Computer Science and Algorithms

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

- Computer science is the study of the principles and use of computers. [Oxford Dictionary, September 2018]
- Computer science is a branch of science that deals with the theory of computation or the design of computers. [Merriam Webster, September 2018]
- Computer science is the study of the theory, experimentation, and engineering that form the basis for the design and use of computers. [Wikipedia, September 2018]
- Computer science is the study of computers, including both hardware and software design. [Webopedia, September 2018]

Further information:
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 algorithms (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?

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 often focus in computer science on the algorithms and their properties and less on the concrete implementations of the 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 and (ii) train you in algorithmic thinking.

Some algorithms are very 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’.

For further information:
- [https://en.wikipedia.org/wiki/Algorithm](https://en.wikipedia.org/wiki/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 find the basic actions that are adequate to the 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. 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.

For further information:

- Algorithmic Thinking: The Key for Understanding Computer Science
- Misconceptions About Computer Science
Section 2: Maze Generation Algorithms

1 Computer Science and Algorithms

2 Maze Generation Algorithms

3 String Search Algorithms

4 Complexity, Correctness, Engineering
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 generation of mazes. For further information:

- [https://en.wikipedia.org/wiki/Maze_solving_algorithm](https://en.wikipedia.org/wiki/Maze_solving_algorithm)
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 computer scientist discovers that the problem is under-specified 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.
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.

For further information:

- https://en.wikipedia.org/wiki/Pair_programming
• 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.
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.)

For further information:

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.
Definition:
• A maze is a graph $G = (V, E)$ with two special nodes, the start node $S$ 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 is the start of the maze
• The second special node is the exit of the maze

Another way of formulating this is to say that a maze is described by a triple $(G, S, X)$ 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 is represented as follows:

- $V = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
- $E = \{(a, b), (a, e), (b, c), (d, h), (e, i), (f, g), (f, j), (g, h), (g, k), (i, j), (k, o), (l, p), (m, n), (n, o), (o, p)\}$
- $S = a$
- $X = p$

Note that this is just one out of many different possible representations of the problem. 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):

```plaintext
graph maze {
  a -- b -- c;
  a -- e -- i -- 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 produce drawings of the graph. Note that producing good drawings for a given graphs 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 run 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
```

For further information:
- [https://en.wikipedia.org/wiki/Graph_drawing](https://en.wikipedia.org/wiki/Graph_drawing)
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 $T$ of an undirected graph $G$ is a subgraph that is a tree which includes all of the vertices of $G$, with minimum possible number of edges. 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.

For further information:

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.

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 $y$?

For further information:
- https://en.wikipedia.org/wiki/Kruskal%27s_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.

The following Haskell program (Listing 1) is calculating a spanning tree, following the ideas of the algorithm outlined on the slide. The implementation is not making a random selection and hence it produces always the same spanning tree for a given graph. (You are not expected to understand the code yet; but you should be able to do so the end of the semester.)

For further information:

Calculate a spanning tree (a maze) over a given graph.

module Maze (Node, Edge, Graph, maze, showDot) where

import Data.List

type Node = Char

type Edge = (Node, Node)

type Graph = ([Node], [Edge])

type Class = [Node]

-- Take a graph and return a spanning tree graph (a maze) by calling
-- buildMaze with a set of equivalence classes (one for each node) and
-- a new graph that initially has only nodes but no edges.
maze :: Graph -> Graph

maze g = buildMaze g (map ([:[]]) ns) (ns, [])
    where (ns,es) = g

-- Take a graph, a list of node equivalence classes, a new (spanning
-- tree) graph (which we are building up) and return a spanning tree
-- graph if only one equivalence is left. Otherwise, find an edge and
-- continue building the spanning tree by merging the classes
-- connected by the edge and adding the edge to the spanning tree.
buildMaze :: Graph -> [Class] -> Graph -> Graph

buildMaze g cs (ns,es)
    | length cs == 1 = (ns,es)
    | otherwise = buildMaze g (mergeClasses cs e) (ns,e:es)
    where e = findEdge cs (snd g)

-- Given a list of classes and edges, find an edge that connects two
-- equivalence classes. We pick the first edge is there are multiple
-- but this could also be a random selection. The distinct helper
-- function tests where an edge connects two equivalence classes.
findEdge :: [Class] -> [Edge] -> Edge

findEdge cs es = head (filter (distinct cs) es)
    where distinct cs (a,b) = filter (elem a) cs /= filter (elem b) cs

-- Given a list of equivalence classes and an edge, merge equivalence
-- classes given a specific edge by partitioning the classes into
-- those touched by the edge and those not touched by the edge. Concat
-- (join) the touched edges and then return the new list of
-- equivalence classes. The match helper function tests whether an
-- edge matches an equivalence class (i.e., whether any of the
-- endpoints is in the equivalence class.
mergeClasses :: [Class] -> Edge -> [Class]

mergeClasses cs e = concat m : n
    where (m,n) = partition (match e) cs
        where match (a,b) c = elem a c || elem b c

-- Show a graph in the dot notation.
showDot :: Graph -> String

showDot (ns,es) = "graph xyz {
    ++ (concat (map showNode ns))
    ++ (concat (map showEdge es))
    ++ "}"
    where showNode n = [n] ++ ","";
        showEdge e = [a] ++ "->" ++ [b] ++ ",";
        where (a,b) = e

-- Read a graph from the dot notation. TBD.
readDot :: String -> Graph
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

For further information:

- [https://en.wikipedia.org/wiki/Maze_generation_algorithm](https://en.wikipedia.org/wiki/Maze_generation_algorithm)
Section 3: String Search Algorithms

1 Computer Science and Algorithms

2 Maze Generation Algorithms

3 String Search Algorithms

4 Complexity, Correctness, Engineering
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.

The search we are considering is more precisely called substring search. There are more expressive search techniques that we do not consider here.

For further information:
• https://en.wikipedia.org/wiki/String_searching_algorithm
Problem Formalization

- Let \( \Sigma \) be a finite set, called an alphabet.
- Let \( k \) denote the number of elements in \( \Sigma \).
- Let \( \Sigma^* \) be the set of all words that can be created out of \( \Sigma \) (Kleene closure of \( \Sigma \)).
- Let \( t \in \Sigma^* \) be a (possible long) text and \( p \in \Sigma^* \) be a (typically short) pattern.
- Let \( n \) denote the length of \( t \) and \( m \) denote the length of \( p \).
- We assume that \( n \gg m \).
- Find the first occurrence 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 Kleene closure \( \Sigma^* \) of the alphabet \( \Sigma \) is the (infinite) set of all words that can be formed with elements out of \( \Sigma \). This includes the empty word of length 0, typically denoted by \( \epsilon \). 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 \( t \).
Naive String Search

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

\[
\begin{align*}
\text{FINDANEEDLEINAHAYSTACK} \\
\text{NEEDLE} \\
\text{NEEDLE} \\
\text{NEEDLE} \\
\text{NEEDLE} \\
\text{NEEDLE}
\end{align*}
\]

An implementation of naive string search in an imperative language like C is straightforward (see Listing 2). 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 text.

```haskell
{- |
module Search (findFirstIn, isPrefixOf) where

isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = (x == y) && isPrefixOf xs ys

findFirstIn :: Eq a => [a] -> [a] -> Int
findFirstIn = cntFindFirstIn 0
  where
   cntFindFirstIn :: Eq a => Int -> [a] -> [a] -> Int
   cntFindFirstIn _ [] [] = -1
   cntFindFirstIn n p t
      | p `isPrefixOf` t = n
      | otherwise = cntFindFirstIn (n+1) p (tail t)

Listing 2: Naive string search implemented in Haskell
```
Naive String Search Performance

• How “fast” is naive string search?
• Idea: Lets try to 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., “aaaaaaaa” 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.
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.
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 mismatching character and the position in the pattern to the number of alignments that can be skipped.

Example: Lets assume \( p = \text{NEED} \) and an alphabet consisting of the characters A-Z. Then the table looks as follows (counting character positions in the pattern starting with 0):

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
A & 0 & 1 & 2 & 3 \\
B & 0 & 1 & 2 & 3 \\
C & 0 & 1 & 2 & 3 \\
D & 0 & 1 & 2 & - \\
E & 0 & - & - & 0 \\
\vdots & \vdots & \vdots & \vdots \\
N & - & 0 & 1 & 2 \\
\vdots & \vdots & \vdots & \vdots \\
Z & 0 & 1 & 2 & 3 \\
\end{array}
\]

This 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. (It is possible to find more space efficient representations of the lookup table. For example, all rows for characters not present in the pattern look the same.)
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, 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: $t = \text{FINDANEEDELEINAHAYSTACK}$, $p = \text{NEEDUNEEED}$

```
  FINDANEEDELEINAHAYSTACK  skip
    needUNEEED
      needUNEEED
```

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

For further information:

- [https://doi.org/10.1145/359842.359859](https://doi.org/10.1145/359842.359859)
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.

For further information:
- [https://doi.org/10.1145/359146.359148](https://doi.org/10.1145/359146.359148)
Section 4: Complexity, Correctness, Engineering

1 Computer Science and Algorithms

2 Maze Generation Algorithms

3 String Search Algorithms

4 Complexity, Correctness, Engineering
Complexity of Algorithms

- Questions:
  - Which maze generation algorithm is faster?
  - Is there a fastest maze generation algorithm?
  - What happens if we consider mazes of different sizes or dimensions?
  - Instead of measuring execution time (which depends on the speed of the computer hardware), can we have a more neutral notion of “fast”?

- Computer science is about analyzing the complexity of algorithms.
- Complexity is an abstract measure of computational effort (time complexity) and memory usage (space complexity).

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

- Suppose we have three algorithms to choose from. For example, consider algorithms to detect cycles in a graph of \( n \) nodes.
- With \( n = 50 \), the exponential algorithm has an execution time of more than 35 years.
- For \( n \geq 1000 \), the exponential algorithm gives us execution times that are longer than the age of the universe!

<table>
<thead>
<tr>
<th>size ( n )</th>
<th>linear ( 100 \mu s )</th>
<th>quadratic ( 7n^2 \mu s )</th>
<th>exponential ( 2^n \mu s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 ( \mu s )</td>
<td>7 ( \mu s )</td>
<td>2 ( \mu s )</td>
</tr>
<tr>
<td>5</td>
<td>500 ( \mu s )</td>
<td>175 ( \mu s )</td>
<td>32 ( \mu s )</td>
</tr>
<tr>
<td>10</td>
<td>1 ms</td>
<td>700 ( \mu s )</td>
<td>1024 ( \mu s )</td>
</tr>
<tr>
<td>50</td>
<td>5 ms</td>
<td>17.5 ms</td>
<td>13 031.25 d</td>
</tr>
<tr>
<td>100</td>
<td>10 ms</td>
<td>70 ms</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>100 ms</td>
<td>7 s</td>
<td></td>
</tr>
<tr>
<td>10 000</td>
<td>1 s</td>
<td>700 s</td>
<td></td>
</tr>
<tr>
<td>100 000</td>
<td>10 s</td>
<td>70 000 s</td>
<td></td>
</tr>
</tbody>
</table>

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 proof 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 \( n_0 \) 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 \( n_0 \) 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**

<table>
<thead>
<tr>
<th>Landau Set</th>
<th>class name</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>$O(\log_2(n))$</td>
<td>logarithmic</td>
<td>2</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>3</td>
</tr>
<tr>
<td>$O(n \log_2(n))$</td>
<td>linear logarithmic</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Landau Set</th>
<th>class name</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td>5</td>
</tr>
<tr>
<td>$O(n^k)$</td>
<td>polynomial</td>
<td>6</td>
</tr>
<tr>
<td>$O(k^n)$</td>
<td>exponential</td>
<td>7</td>
</tr>
</tbody>
</table>

**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 $l > 1$.***

---

**Commonly Used Landau Sets**

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(2^n)$

---

**Figure:** Graph comparing the growth of commonly used Landau Sets.
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 $(f_1 + f_2) \in O(|g_1| + |g_2|)$.
- If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $(f_1f_2) \in O(g_1g_2)$.

Examples:

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

- 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 $\Omega$ 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 sciences 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 is always correct regarding a specification, i.e., 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 underlying machine always passing through the same sequence of states.

- Some factors that make an algorithm 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.
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 important 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".
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.

For further information:
Part II

Discrete Mathematics

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.
Section 5: Terminology, Notations, Proofs

- Terminology, Notations, Proofs
- Sets
- Relations
- 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)

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.
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 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 really difficult to work with. (And this makes natural language processing really difficult for computers.) In mathematics and computer science, we try hard to avoid ambiguities.

Note that predicates are more expressive than simple propositions. As a consequence, the mathematical logic to deal with propositions (called propositional logic or Boolean logic) is simpler than the logic that deals with predicates (called predicate logic or first-order logic).
When developing a theory and using formal proofs, it is important to be clear about the underlying axioms and the propositions that are used. Ideally, a small number of well defined axioms are sufficient to develop and prove 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 particularly interesting since it allows us to prove a statement for all natural numbers even though there are infinite many natural numbers. We will make use of this technique, called induction, frequently.
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.
Mathematical Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \land Q )</td>
<td>logical and of propositions ( P ) and ( Q )</td>
</tr>
<tr>
<td>( P \lor Q )</td>
<td>logical or of propositions ( P ) and ( Q )</td>
</tr>
<tr>
<td>( \neg P )</td>
<td>negation of proposition ( P )</td>
</tr>
<tr>
<td>( \forall x \in S. P )</td>
<td>the predicate ( P ) holds for all ( x ) in the set ( S )</td>
</tr>
<tr>
<td>( \exists x \in S. P )</td>
<td>there exists an ( x ) in the set ( S ) such that the predicate ( P ) holds</td>
</tr>
<tr>
<td>( P \implies Q )</td>
<td>the statement ( P ) implies statement ( Q )</td>
</tr>
<tr>
<td>( P \iff Q )</td>
<td>the statement ( P ) holds if and only if (iff) ( Q ) holds</td>
</tr>
</tbody>
</table>

Some examples:

- The following statement in mathematical notation
  \[
  \forall x. \exists y. x = y \iff \neg (x \neq y)
  \]
  reads as follows:
  For all \( x \), there is a \( y \), such that \( x = y \), if and only if it is not the case that \( x \) is unequal to \( y \).

- We can write the five Peano axioms in mathematical notation. Let's assume that the function \( s : \mathbb{N} \to \mathbb{N} \) returns the successor of its argument.
  
  \[
  \begin{align*}
  \text{P1} & \quad 0 \in \mathbb{N} \quad \quad \text{(zero is a natural number)} \\
  \text{P2} & \quad \forall n \in \mathbb{N}. s(n) \in \mathbb{N} \land n \neq s(n) \quad \text{(closed under successor, distinct)} \\
  \text{P3} & \quad \neg (\exists n \in \mathbb{N}. 0 = s(n)) \quad \text{(zero is not a successor)} \\
  \text{P4} & \quad \forall n \in \mathbb{N}. \forall m \in \mathbb{N}. s(n) = s(m) \Rightarrow n = m \quad \text{(different successors)} \\
  \text{P5} & \quad \forall P. (P(0) \land (\forall n \in \mathbb{N}. P(n) \implies P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}. P(m)) \quad \text{(induction)}
  \end{align*}
  \]

- 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.
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 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 really get much simpler if all people within a project follow common conventions, the so-called coding styles.

The same applies to mathematicians to some extent. 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.
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 scratchwork 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 \( \square \).
- 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.
Prove an Implication by Derivation

• An implication is a proposition of the form “If $P$, then $Q$”, or $P \Rightarrow Q$.
• 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 $Q$. \hfill $\square$

Theorem 2. \textit{Let} $x$ \textit{and} $y$ \textit{be two integers. If} $x$ \textit{and} $y$ \textit{are both odd, then the product} $xy$ \textit{is odd.} \hfill $\square$

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

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

Since $2(2ab + a + b)$ \textit{is even and we add 1 to it, it follows that the product of} $x$ \textit{and} $y$ \textit{is odd.} \hfill $\square$
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 $\neg P$. \hfill $\Box$}

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. \hfill $\Box$

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” 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 \( P \). \( \square \)

**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.

Now, 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., 2 divides \( x^2 \). This means that 2 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 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 [. . . ], 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

**add an example**
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:

  \[
  \text{Proof. We prove } P \text{ by considering the cases } c_1, \ldots, c_N. \\
  \text{Case 1: Suppose } c_1. \text{ Prove of } P \text{ for } c_1. \\
  \ldots \\
  \text{Case } N: \text{ Suppose } c_N. \text{ Prove of } P \text{ for } c_N. \\
  \text{Since } P \text{ holds for all cases } c_1, \ldots c_N, \text{ the statement } P \text{ holds.} \]

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

Proof. We proof $n^2 + n$ 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 $n \in \mathbb{Z}$. \qed
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.

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

Proof. We use proof by contradiction. Suppose the claim is false. Assume $\sqrt{2}$ 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 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
  
  $0 = \frac{0(0+1)}{2} = 0 \cdot \frac{1}{2}$

  and this is true since the product of 0 with any number is 0.

- Induction step:
  Assume that the equation is true for some $n$. Let’s consider the case $n + 1$:

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

  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 $n$. $\square$
Section 6: Sets

5 Terminology, Notations, 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 math, 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., \( \{x | x \geq 0 \land x < 2^8\} \),
  - 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 \( \{b, a, c\} \) are different representations of the same set.

Some popular sets in mathematics:

<table>
<thead>
<tr>
<th>symbol</th>
<th>set</th>
<th>elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>empty set</td>
<td>{0, 1, 2, 3, \ldots}</td>
</tr>
<tr>
<td>( \mathbb{N} )</td>
<td>nonnegative integers</td>
<td>{0, 1, 2, 3, \ldots}</td>
</tr>
<tr>
<td>( \mathbb{Z} )</td>
<td>integers</td>
<td>{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots}</td>
</tr>
<tr>
<td>( \mathbb{Q} )</td>
<td>rational numbers</td>
<td>( \frac{1}{2}, 42, \text{ etc.} )</td>
</tr>
<tr>
<td>( \mathbb{R} )</td>
<td>real numbers</td>
<td>( \pi, \sqrt{2}, 0 \text{ etc.} )</td>
</tr>
<tr>
<td>( \mathbb{C} )</td>
<td>complex numbers</td>
<td>( i, 5 + 9i, 0, \pi \text{ etc.} )</td>
</tr>
</tbody>
</table>

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:

\[
R = \{x | x \notin x\}, \text{ then } R \in R \iff R \notin R.
\]

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)
Let $A$ and $B$ be two sets. We define the following relations between sets:

1. $(A \equiv B) : \iff (\forall x. x \in A \iff x \in B)$ (set equality)
2. $(A \subseteq B) : \iff (\forall x. x \in A \Rightarrow x \in B)$ (subset)
3. $(A \subset B) : \iff (A \subseteq B) \land (A \neq B)$ (proper subset)
4. $(A \supseteq B) : \iff (\forall x. x \in B \Rightarrow x \in A)$ (superset)
5. $(A \supset B) : \iff (A \supseteq B) \land (A \neq B)$ (proper superset)

- Obviously:
  - $(A \subseteq B) \land (B \subseteq A) \Rightarrow (A \equiv B)$
  - $(A \subseteq B) \iff (B \supseteq A)$

Obviously, $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ 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 Jacobs University. How do you think this set is defined? And is it a proper subset of the set of all employees of Jacobs University?
Operations on Sets 1/2

**Definition (set union)**
The *union* of two sets $A$ and $B$ is defined as $A \cup B = \{x | x \in A \lor x \in B\}$.

**Definition (set intersection)**
The *intersection* of two sets $A$ and $B$ is defined as $A \cap B = \{x | x \in A \land x \in B\}$.

**Definition (set difference)**
The *difference* of two sets $A$ and $B$ is defined as $A \setminus B = \{x | x \in A \land x \notin B\}$.

Some basic properties of set unions:
- $A \cup B = B \cup A$
- $A \cup (B \cup C) = (A \cup B) \cup C$
- $A \subseteq (A \cup B)$
- $A \cup A = A$
- $A \cup \emptyset = A$
- $A \subseteq B \iff A \cup B = B$

Some basic properties of set intersections:
- $A \cap B = B \cap A$
- $A \cap (B \cap C) = (A \cap B) \cap C$
- $A \cap B \subseteq A$
- $A \cap A = A$
- $A \cap \emptyset = \emptyset$
- $A \subseteq B \iff A \cap B = A$

Some basic properties of set differences:
- $A \setminus B \neq B \setminus A$ for $A \neq B$
- $A \setminus A = \emptyset$
- $A \setminus \emptyset = A$
Definition (power set)

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

$$X_1 \times \ldots \times X_n = \{ (x_1, \ldots, x_n) \mid \forall i. 1 \leq i \leq n \Rightarrow x_i \in X_i \}.$$ 

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

Definition (countable)
A set $A$ is *countable* if and only if it is finite or countably infinite.

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. $|(A \times B)| = |A| \cdot |B|$

The proof of this theorem is straightforward and you may do this as a homework exercise.

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

- Terminology, Notations, Proofs
- Sets
- Relations
- Functions
Relations

Definition (relation)

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

- 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 $X_1 \times \ldots \times X_k$

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 often focus on binary relations.

Relations are a fairly general concept. Relations play a very 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
  - ...

- Relations:
  - enrolled_in $\subseteq (\text{Student} \times \text{Course})$
  - is_a $\subseteq (\text{Student} \times \text{Person})$
  - is_a $\subseteq (\text{Instructor} \times \text{Person})$
  - belongs_to $\subseteq (\text{Course} \times \text{Module})$
  - teaches $\subseteq (\text{Instructor} \times \text{Course})$
  - tutor_of $\subseteq (\text{Student} \times \text{Course})$
  - ...

Entity relationship models are usually written using a graphical notation. A standard graphical notation is part of the Unified Modeling Language (UML).
Binary Relations

Definition (binary relation)
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 (inverse of a binary relation)
The inverse of a binary relation $R \subseteq A \times B$ is the relation $R^{-1} \subseteq B \times A$ defined by the rule

$$b R^{-1} a \iff a R b.$$ 

- For $a \in A$ and $b \in B$, we often write $a R b$ to indicate that $(a, b) \in R$.
- The notation $a R b$ is called infix notation while the notation $R(a, b)$ is called the prefix notation. For binary relations, we commonly use the infix notation.

Another way to define the inverse relation is to use the set builder notation: Given a binary relation $R \subseteq A \times B$, we define the inverse relation $R^{-1}$ as $R^{-1} = \{(b, a) \in (B \times A) | (a, b) \in R\}$.

It is also possible to define the complement relation of a binary relation. Given a binary relation $R \subseteq A \times B$, we define the complement relation $\overline{R}$ as $\overline{R} = \{(a, b) \in (A \times B) | (a, b) \notin R\}$.

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 $\text{dom}(R)$ and $\text{codom}(R)$ to refer to the domain and the codomain of a relation $R$. 
Image and Range of Binary Relations

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 \( A \).

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 \( B \).

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

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

```dot
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 dot graph notation:

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

The complement \( \bar{R} \) of \( R \) defined in the dot graph notation:

```dot
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 \( \forall a, b, c \in A . ((a, b) \in R \land (b, c) \in R) \Rightarrow (a, c) \in R \)
- total iff \( \forall a, b \in A . (a, b) \in R \lor (b, a) \in R \)

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

Examples:

- The relation \( \text{is.as.old.as} \) on the set of persons is reflexive.
- The relation \( \text{is.older_than} \) on the set of persons is irreflexive.
- The relation \( \text{is.sibling_of} \) on the set of persons is symmetric.
- The relation \( \text{is.mother_of} \) on the set of persons is asymmetric.
- The relation \( \text{is.notOLDER.as} \) on the set of persons is antisymmetric.
- The relation \( \text{is.ancestor_of} \) on the set of persons is transitive.

The relation \( \text{is.as.old.as} \) is reflexive, symmetric and transitive. Hence it is an equivalence relation.

Note: An equivalence relation induces equivalence classes. Given a set of persons, the \( \text{is.as.old.as} \) relation induces classes of persons with the same age.
Equivalence, Partial Order, and Strict Partial Order

**Definition**
An equivalence 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 on $A$. The relation $R$ is called a strict partial order on $A$ if and only if it is irreflexive, asymmetric and transitive on $A$.

**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 $\leq$ induces a strict partial order $a \prec b \iff a \preceq b \land a \neq b$.

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

Examples:
- 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.

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

$$\vec{x} \preceq \vec{y} \text{ if and only if } \forall i \in \{1, \ldots, n\}.x_i \leq y_i$$

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.

<table>
<thead>
<tr>
<th>property</th>
<th>eq</th>
<th>po</th>
<th>spo</th>
<th>definition</th>
<th>$= \leq &lt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflexive</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \sim a$</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>irreflexive</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \not\sim a$</td>
<td>✓</td>
</tr>
<tr>
<td>symmetric</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \sim b \Rightarrow b \sim a$</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>asymmetric</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \sim b \Rightarrow b \not\sim a$</td>
<td>✓</td>
</tr>
<tr>
<td>antisymmetric</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \sim b \land b \sim a \Rightarrow a = b$</td>
<td>✓</td>
</tr>
<tr>
<td>transitive</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$a \sim b \land b \sim c \Rightarrow a \sim c$</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

**eq** = equivalence relation, **po** = partial order, **spo** = 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 Terminology, Notations, 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 $f(x) = \bot$. 
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 graph 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 graph 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 graph 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 \( C \).

Haskell provides the \( . \) operator for function composition.

```haskell
1 last' :: [a] -> a
2 last' = (head . reverse)

1 import Data.List
2 rsort :: Ord a => [a] -> [a]
3 rsort = (reverse . sort)

1 odd' :: Int -> Bool
2 odd' = not . even

1 sumOfOdds :: Integral a => [a] -> a
2 sumOfOdds = sum . filter (not . even)
```
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 in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

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:
  \[
  \lambda n \in \mathbb{N}. n = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid y = x\} = \{(0, 0), (1, 1), (2, 2), \ldots\}
  \]

- \( f(x) = x^2 \):
  \[
  \lambda x \in \mathbb{N}. x^2 = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid y = x^2\} = \{(0, 0), (1, 1), (2, 4), \ldots\}
  \]

- Addition of natural numbers:
  \[
  \lambda (x, y) \in \mathbb{N} \times \mathbb{N}. x + y = \{((x, y), z) \in (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \mid z = x + y\} = \{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 2), \ldots\}
  \]

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

1. \( f :: \text{Num a} \Rightarrow a \rightarrow a \)
2. \( f = (\lambda x \rightarrow x^2) \cdot (\lambda x \rightarrow x+1) \)

For further information:
- [https://en.wikipedia.org/wiki/Lambda_calculus](https://en.wikipedia.org/wiki/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 function with multiple arguments into a sequence of functions with a single argument is called currying.
- The term currying is a reference to the logician 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

Number Systems, Units, Characters, Date and Time

In this part, we look at vary basic data types and how they are typically 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 consider the different character sets used in the word. Operations like character comparisons can become quite challenging in practice.

We finally 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 and it turns out that time is actually often not a good concept in distributed computing systems since establishing an accurate common notion of time is quite challenging.
Numbers can be confusing...

- There are only 10 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?

For further information:

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Natural</td>
<td>0, 1, 2, 3, 4, ...</td>
</tr>
<tr>
<td>Z</td>
<td>Integer</td>
<td>..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...</td>
</tr>
<tr>
<td>Q</td>
<td>Rational</td>
<td>( \frac{a}{b} ), where ( a \in \mathbb{Z} ) and ( b \in \mathbb{Z} ) and ( b \neq 0 )</td>
</tr>
<tr>
<td>R</td>
<td>Real</td>
<td>The limit of a convergent sequence of rational numbers</td>
</tr>
<tr>
<td>C</td>
<td>Complex</td>
<td>( a + bi ), where ( a \in \mathbb{R} ) and ( b \in \mathbb{R} ) and ( i = \sqrt{-1} )</td>
</tr>
</tbody>
</table>

- Numbers should be distinguished from numerals, the symbols used to represent numbers. A single number can have many different representations.
Section 9: Natural Numbers

Natural Numbers

Integer Numbers

Rational and Real Numbers

Floating Point Numbers

International System of Units

Characters and Strings

Date and Time
Numeral Systems for Natural Numbers

- Natural numbers can be represented according to different bases. We commonly use decimal number (base 10) 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:

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

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

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

```plaintext
1: function DEC2BIN(n)
2:     s ← "n"                                        ▷ $s$ holds a bit string
3:     repeat
4:         b ← n mod 2                             ▷ the remainder gives us the next bit
5:         prepend bit $b$ to the bit string $s$
6:         n ← n div 2                            ▷ the number still left to convert
7:     until $n = 0$
8:     return $s$
9: end function
```

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

```plaintext
1: function BIN2DEC(s)
2:     n ← 0                                       ▷ $n$ holds a natural number
3:     while bit string $s$ is not empty do
4:         b ← leftmost bit of the bit string $s$   ▷ removes the bit from the bit string
5:         n ← 2 · $n + b$
6:     od
7:     return $n$
8: 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):

$$ 110001101011100_2 = 01102 \, 00112 \, 01012 \, 11002 = 635_{16} $$

$$ \underbrace{110011010111000}_6 \underbrace{110}_{3} \underbrace{1110}_{5} \underbrace{110}_{4} = 61534_8 $$

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

$$ \text{deadbeaf}_{16} = d_{16} \, c_{16} \, a_{16} \, d_{16} \, b_{16} \, e_{16} \, a_{16} \, f_{16} = 110111101010111101011111_{2} $$
Natural Numbers Literals

- Computer scientists often use special prefix conventions to write natural number literals in a way that indicates the base:

<table>
<thead>
<tr>
<th>prefix</th>
<th>example</th>
<th>meaning</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>42</td>
<td>$42_{10}$</td>
<td>decimal number</td>
</tr>
<tr>
<td>0x</td>
<td>0x42</td>
<td>$42_{16} = 66_{10}$</td>
<td>hexadecimal number</td>
</tr>
<tr>
<td>0</td>
<td>042</td>
<td>$42_{8} = 34_{10}$</td>
<td>octal number</td>
</tr>
<tr>
<td>0b</td>
<td>0b1000010</td>
<td>$1000010_2 = 42_{10}$</td>
<td>binary number</td>
</tr>
</tbody>
</table>

- Beware that 42 and 042 may not represent the same number!
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.

<table>
<thead>
<tr>
<th>bits</th>
<th>name</th>
<th>range (decimal)</th>
<th>range (hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>nibble</td>
<td>0-15</td>
<td>0x0-0xf</td>
</tr>
<tr>
<td>8</td>
<td>byte, octet, uint8</td>
<td>0-255</td>
<td>0x0-0xff</td>
</tr>
<tr>
<td>16</td>
<td>uint16</td>
<td>0-65 535</td>
<td>0x0-0xffff</td>
</tr>
<tr>
<td>32</td>
<td>uint32</td>
<td>0-4 294 967 295</td>
<td>0x0-0xffffffff</td>
</tr>
<tr>
<td>64</td>
<td>uint64</td>
<td>0-18 446 744 073 709 551 615</td>
<td>0x0-0xffffffffffffffff</td>
</tr>
</tbody>
</table>

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

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

```haskell
-- Surprises with fixed precision unsigned integers (not only in Haskell)
import Data.Word
import Text.Printf

w8max = maxBound :: Word8
w8min = minBound :: Word8

main = do
    printf "%3u + 1 = %3u\n" w8max (w8max + 1)
    printf "%3u - 1 = %3u\n" w8min (w8min - 1)
    printf " 42 + %3u + 1 = %3u\n" w8max (42 + w8max + 1)
    printf " 42 - %3u - 1 = %3u\n" w8max (42 - w8max - 1)
```

Careless choice of fixed precision number ranges can lead to serious disasters. Things are even more subtle in programming languages that allow automatic type conversions.
Section 10: Integer Numbers

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- Date and Time
Integer Numbers

• Integer numbers can be negative but surprisingly there are not “more” of them (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 $f : \mathbb{Z} \rightarrow \mathbb{N}$ (and the inverse function $f^{-1} : \mathbb{N} \rightarrow \mathbb{Z}$):

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

• So 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.
• 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.

• Let's represent the numbers \( 0 \ldots b^n - 1 \) 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: \( b = 2, n = 4 \): \( 5_{10} = 0101_2, -5_{10} = 1010_2 \)

• 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.
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.
- Let's again represent the numbers \( 0 \ldots b^{n-1} \) in the usual way.
- To represent negative numbers, we invert the absolute value \((a_{n-1}a_{n−2} \cdots a_1a_0)_b\) by calculating \((a'_{n-1}a'_{n−2} \cdots a'_1a'_0)_b\) with \(a'_i = (b − 1) − a_i\) and adding 1 to it.
- Example: \( b = 2, n = 4 \): \( 5_{10} = 0101_2, -5_{10} = 1011_2 \)

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 (b complement):
- 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:

\[
\begin{array}{c}
0010 \\
+ 1010 \\
\hline
1100
\end{array}
\]

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.

<table>
<thead>
<tr>
<th>bits</th>
<th>name</th>
<th>range (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>int8</td>
<td>−128 to 127</td>
</tr>
<tr>
<td>16</td>
<td>int16</td>
<td>−32 768 to 32 767</td>
</tr>
<tr>
<td>32</td>
<td>int32</td>
<td>−2 147 483 648 to 2 147 483 647</td>
</tr>
<tr>
<td>64</td>
<td>int64</td>
<td>−9 223 720 368 547 758 088 to 9 223 720 368 547 758 087</td>
</tr>
</tbody>
</table>

• 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.

```haskell
-- Surprises with fixed precision integers (not only in Haskell)
import Data.Int
import Text.Printf

i8max = maxBound :: Int8
i8min = minBound :: Int8

main = do
    printf "%3d + 1 = %3d\n" i8max (i8max + 1)
    printf "%3d - 1 = %3d\n" i8min (i8min - 1)
```
Section 11: Rational and Real Numbers

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- 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):
  
  ```haskell
  import Data.Ratio
  print $ 1%2 + 1%3
  ```

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 << '
';
    return 0;
}
```

C programmers can use the GNU multiple precision arithmetic library:

```c
#include <gmp.h>

int main()
{
    mpz_t a, b, c;
    mpz_inits(a, b, c, NULL);
    mpz_set_str(a, "1/2", 10);
    mpz_set_str(b, "1/3", 10);
    mpz_add(c, a, b);
    gmp_printf("%Qd\n", c);
    mpz_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 $\pi$ 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 if we use these approximations. If we are not very careful, these rounding errors can accumulate badly.
- The notion of numeric stability can be used to classify algorithms according how they propagate rounding errors.
Section 12: Floating Point Numbers

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- 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
  \[ s \times d_0d_1d_2 \ldots d_{p-1} \times b^e = s \times \sum_{k=0}^{p-1} d_k b^{-k} \times b^e \]
  where $b$ is the basis, $e$ 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.

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 $2.99792458 \times 10^8$ m s$^{-1}$ and the elementary positive charge as $1.602176634 \times 10^{-19}$ 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 bit string representing a binary fraction

1: function DECF2BIN(F)
2:  $s \leftarrow \emptyset$
3:  repeat
4:  $f \leftarrow f \cdot 2$
5:  $b \leftarrow \text{int}(f)$
6:  append $b$ to the bit string $s$
7:  $f \leftarrow \text{frac}(f)$
8:  until $f = 0$
9:  return $s$
10: end function

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

1: function BINF2DECF(S)
2:  $f \leftarrow 0.0$
3:  while bit string $s$ is not empty do
4:    $b \leftarrow \text{rightmost bit of the bit string } s$
5:    $f \leftarrow (f + b)/2$
6:  od
7:  return $f$
8: 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. 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.

For further information:
IEEE 754 Floating Point Formats

<table>
<thead>
<tr>
<th>precision</th>
<th>single (float)</th>
<th>double</th>
<th>quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>1 bit</td>
<td>1 bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>exponent</td>
<td>8 bit</td>
<td>11 bit</td>
<td>15 bit</td>
</tr>
<tr>
<td>exponent range</td>
<td>[-126...127]</td>
<td>[-1022...1023]</td>
<td>[-16382...16383]</td>
</tr>
<tr>
<td>exponent bias</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
</tr>
<tr>
<td>mantissa</td>
<td>23 bit</td>
<td>52 bit</td>
<td>112 bit</td>
</tr>
<tr>
<td>total size</td>
<td>32 bit</td>
<td>64 bit</td>
<td>128 bit</td>
</tr>
<tr>
<td>decimal digits</td>
<td>( \approx 7.2 )</td>
<td>( \approx 15.9 )</td>
<td>( \approx 34.0 )</td>
</tr>
</tbody>
</table>

- 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 \( 1 \times 10^{-1} \) cannot be represented precisely.

---

### 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|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.

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 a 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 clearing 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.

```haskell
-- Surprises with double floating point numbers (not only in Haskell)
import Text.Printf

x :: Double
x = 2.0 / 0.0

y :: Double
y = x + 1.0

z :: Double
z = x - x

a :: Double
a = (1.0 + 1e20) - 1e20

b :: Double
b = 1.0 + (1e20 - 1e20)

c :: Double
c = 0.1 + 0.2

d :: [Double]
d = iterate (+0.1) 0.0

main = do
    printf "2.0 / 0.0 = %g\n" x
    printf "2.0 / 0.0 + 1.0 = %g\n" y
    printf "2.0 / 0.0 - 2.0 / 0.0 = %g\n" z
    printf "(1.0 + 1e20) - 1e20 = %g\n" a
    printf "1.0 + (1e20 - 1e20) = %g\n" b
    printf "0.1 + 0.2 = %g\n" c
    printf "counting from 0.0 up in steps of 0.1 = %s\n" $ show $ take 11 d

For further information:
- https://www.cl.cam.ac.uk/teaching/1011/FPComp/fpcomp10slides.pdf
Section 13: International System of Units

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- Date and Time
Importance of Units and Unit Prefixes

- Most numbers we encounter in practice have associated units. It is important to be very clear 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 clear about units.
  - And always be clear about the unit prefixes.

For further information:
- https://en.wikipedia.org/wiki/Mars_Climate_Orbiter
### SI Base Units

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

For further information:

## SI Derived Units

- 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>herz</td>
<td>Hz</td>
<td>$s^{-1}$</td>
<td>frequency</td>
</tr>
<tr>
<td>newton</td>
<td>N</td>
<td>$kg \cdot m \cdot s^{-1}$</td>
<td>force</td>
</tr>
<tr>
<td>watt</td>
<td>W</td>
<td>$kg \cdot m^2 \cdot s^{-3}$</td>
<td>power</td>
</tr>
<tr>
<td>volt</td>
<td>V</td>
<td>$kg \cdot m^2 \cdot s^{-3} \cdot A^{-1}$</td>
<td>voltage</td>
</tr>
<tr>
<td>ohm</td>
<td>$\Omega$</td>
<td>$kg \cdot m^2 \cdot s^{-2} \cdot A^{-1}$</td>
<td>resistance</td>
</tr>
<tr>
<td>velocity</td>
<td></td>
<td>$m \cdot s^{-1}$</td>
<td>speed</td>
</tr>
</tbody>
</table>

For further information:
- [https://en.wikipedia.org/wiki/SI_derived_unit](https://en.wikipedia.org/wiki/SI_derived_unit)
### Metric Prefixes (International System of Units)

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

Jürgen Schönwälder (Jacobs University Bremen) 
Introduction to Computer Science 
December 5, 2019
<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Base 10</th>
<th>Base 1000</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>milli</td>
<td>m</td>
<td>$10^{-3}$</td>
<td>1000$^{-1}$</td>
<td>0.001</td>
</tr>
<tr>
<td>micro</td>
<td>µ</td>
<td>$10^{-6}$</td>
<td>1000$^{-2}$</td>
<td>0.000 001</td>
</tr>
<tr>
<td>nano</td>
<td>n</td>
<td>$10^{-9}$</td>
<td>1000$^{-3}$</td>
<td>0.000 000 001</td>
</tr>
<tr>
<td>pico</td>
<td>p</td>
<td>$10^{-12}$</td>
<td>1000$^{-4}$</td>
<td>0.000 000 000 001</td>
</tr>
<tr>
<td>femto</td>
<td>f</td>
<td>$10^{-15}$</td>
<td>1000$^{-5}$</td>
<td>0.000 000 000 000 001</td>
</tr>
<tr>
<td>atto</td>
<td>a</td>
<td>$10^{-18}$</td>
<td>1000$^{-6}$</td>
<td>0.000 000 000 000 000 001</td>
</tr>
<tr>
<td>zepto</td>
<td>z</td>
<td>$10^{-21}$</td>
<td>1000$^{-7}$</td>
<td>0.000 000 000 000 000 000 001</td>
</tr>
<tr>
<td>yocto</td>
<td>y</td>
<td>$10^{-24}$</td>
<td>1000$^{-8}$</td>
<td>0.000 000 000 000 000 000 000 001</td>
</tr>
</tbody>
</table>
### Binary Prefixes

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

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 $2^{30} \text{ B} = 1 073 741 824 \text{ B}$, or 1 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 GB, which is $10^9 \text{ B} = 1 000 \ 000 \ 000 \ \text{B}$. The difference is 73 741 824 B (almost 7 % of 1 GiB).

The binary prefixes, proposed in 2000, can help avoid the 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 right prefixes.
Section 14: Characters and Strings

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- 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 many different ways.

Please note again the distinction between a character and the encoding of a character. There can be many different encodings for the same character.
The American Standard Code for Information Interchange (ASCII) is a still widely used character encoding standard.

Originally, 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 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 many different character sets.
<table>
<thead>
<tr>
<th>ASCII Characters and Code Points (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 nul 1 soh 2 stx 3 etx 4 eot 5 enq 6 ack 7 bel</td>
</tr>
<tr>
<td>8 bs 9 ht 10 nl 11 vt 12 sp 13 cr 14 so 15 si</td>
</tr>
<tr>
<td>16 dle 17 dc1 18 dc2 19 dc3 20 dc4 21 nak 22 syn 23 etb</td>
</tr>
<tr>
<td>24 can 25 em 26 sub 27 esc 28 fs 29 gs 30 rs 31 us</td>
</tr>
<tr>
<td>32 sp 33 ! 34 &quot; 35 # 36 $ 37 % 38 &amp; 39 '</td>
</tr>
<tr>
<td>40 ( 41 ) 42 * 43 + 44 , 45 - 46 . 47 /</td>
</tr>
<tr>
<td>48 0 49 1 50 2 51 3 52 4 53 5 54 6 55 7</td>
</tr>
<tr>
<td>56 8 57 9 58 : 59 ; 60 &lt; 61 = 62 &gt; 63 ?</td>
</tr>
<tr>
<td>64 @ 65 A 66 B 67 C 68 D 69 E 70 F 71 G</td>
</tr>
<tr>
<td>72 H 73 I 74 J 75 K 76 L 77 M 78 N 79 O</td>
</tr>
<tr>
<td>80 P 81 Q 82 R 83 S 84 T 85 U 86 V 87 W</td>
</tr>
<tr>
<td>88 X 89 Y 90 Z 91 [ 92 \ 93 ] 94 ^ 95 _</td>
</tr>
<tr>
<td>96 ` 97 a 98 b 99 c 100 d 101 e 102 f 103 g</td>
</tr>
<tr>
<td>104 h 105 i 106 j 107 k 108 l 109 m 110 n 111 o</td>
</tr>
<tr>
<td>112 p 113 q 114 r 115 s 116 t 117 u 118 v 119 w</td>
</tr>
<tr>
<td>120 x 121 y 122 z 123 { 124</td>
</tr>
</tbody>
</table>
• 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 12.1 (published May 2019) defines 137,929 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 $n$ would become 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 Haskell, characters are Unicode characters. The Unicode code point for a given character can be obtained from the `ord :: Char -> Int` function and the Unicode character of a given code point can be obtained from the `chr :: Int -> Char` function defined in `Data.Char`. The `Data.Char` defines many useful functions to classify characters or to change the case of characters.
## Unicode Transformation Format UTF-8

<table>
<thead>
<tr>
<th>bytes</th>
<th>cp</th>
<th>bits</th>
<th>first cp</th>
<th>last cp</th>
<th>byte 1</th>
<th>byte 2</th>
<th>bytes 3</th>
<th>byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
<td>U+0000</td>
<td>U+007F</td>
<td>0xxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>U+0080</td>
<td>U+07FF</td>
<td>110xxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>16</td>
<td>U+0800</td>
<td>U+FFFF</td>
<td>1110xxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>21</td>
<td>U+10000</td>
<td>U+10FFFF</td>
<td>11110xx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
</tr>
</tbody>
</table>

- Variable-length encoding of Unicode code points (cp) in such a way that seven-bit ASCII becomes valid UTF-8.
- 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 2279, which was published in January 1998.

For further information:
- [https://doi.org/10.17487/RFC2279](https://doi.org/10.17487/RFC2279)
Strings

• Let $\Sigma$ be a non-empty finite set of symbols (or characters), called the alphabet.
• A string (or word) over $\Sigma$ is any finite sequence of symbols from $\Sigma$, including (of course) the empty sequence.
• Typical operations on strings are length(), concatenation(), reverse(), ...
• There are different ways to store strings internally. Two common approaches are:
  • The sequence is null-terminated, i.e., the characters of the string are followed by a special NUL character.
  • The sequence is length-prefixed, i.e., a natural number indicating the length of the string 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. In Haskell, a string is a list of characters.
Section 15: Date and Time

- Natural Numbers
- Integer Numbers
- Rational and Real Numbers
- Floating Point Numbers
- International System of Units
- Characters and Strings
- 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 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. 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, many embedded systems continue to use 32-bit signed integers as time_t and they will be vulnerable when we hit the year 2038 (only 20 years left).

For further information:

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.

For further information:
- [https://www.iana.org/time-zones](https://www.iana.org/time-zones)
- [https://youtu.be/-5wpm-gesOY](https://youtu.be/-5wpm-gesOY)
ISO 8601 Date and Time Formats

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

<table>
<thead>
<tr>
<th>name</th>
<th>format</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>yyyy-mm-dd</td>
<td>2017-06-13</td>
</tr>
<tr>
<td>time</td>
<td>hh:mm:ss</td>
<td>15:22:36</td>
</tr>
<tr>
<td>date and time</td>
<td>yyyy-mm-ddThh:mm:ss±hh:mm</td>
<td>2017-06-13T15:22:36+02:00</td>
</tr>
<tr>
<td>date and time</td>
<td>yyyy-mm-ddThh:mm:ss±hh:mm</td>
<td>2017-06-13T13:22:36+00:00</td>
</tr>
<tr>
<td>date and week</td>
<td>yyyy-mm-ddThh:mm:ssZ</td>
<td>2017-06-13T13:22:36Z</td>
</tr>
<tr>
<td>date and week</td>
<td>yyyy-Www</td>
<td>2017-W24</td>
</tr>
</tbody>
</table>

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, a profile of ISO 8601, however, defines that -00:00 indicates that the offset is unknown.

For further information:
- [https://doi.org/10.17487/RFC3339](https://doi.org/10.17487/RFC3339)
- [https://xkcd.com/1179/](https://xkcd.com/1179/)
Part IV

Boolean Algebra and Logic

Boolean logic is the mathematical framework for describing anything that is *binary*, that is, anything which can have only two values.

- Boolean logic is the foundation for understanding digital circuits, the foundation of today’s computers. The central processing unit (CPU) of a computer is just a very large collection of digital circuits consisting of logic gates. The mathematical foundation of a CPU is Boolean logic since it helps to understand how digital circuits and be composed and where necessary verified.

- Boolean logic is also the foundation of systems that behave as if they were intelligent, i.e., systems that exhibit artificial intelligence. Logic (not just Boolean logic) is a sub-field of artificial intelligence that enabled programs to reason about their environment, to solve planning problems, or to provide powerful search facilities on large amounts of formalized knowledge.

This part introduces the basics of Boolean logic. More advanced logics and the usage of advanced logics will be covered in second and third year courses.
Logic can be confusing.

- If all men are mortal and Socrates is a man, then Socrates is mortal.
- I like Pat or I like Joe.
  If I like Pat, I like Joe.
  Do I like Joe?
- If cats are dogs, then the sun shines.
- “Logic is the beginning of wisdom, not the end of it.”
Section 16: Elementary Boolean Functions

- Elementary Boolean Functions
- Boolean Functions and Formulas
- Boolean Algebra Equivalence Laws
- Normal Forms (CNF and DNF)
- Complexity of Boolean Formulas
- Boolean Logic and the Satisfiability Problem
Boolean Variables

- Boolean logic 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 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.
- The main purpose of Boolean logic is to describe (or design) interdependencies between Boolean variables.
**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 $X = \{X_1, X_2, X_3, \ldots\}$.

**Definition (Interpretation)**

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

- The set $X$ is very large. It is often sufficient to work with a suitable subset $D$ of $X$.
- An interpretation assigns to every Boolean variable a value.
- An interpretation is also called a truth value assignment.
Boolean ∧ Function (and)

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \land Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- The logical *and* ($\land$) can be viewed as a function that maps two Boolean values to a Boolean value:

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

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

- In programming languages like C or C++ (and even Haskell), the operator `&&` is often used to represent the $\land$ operation.
The logical or (∨) can be viewed as a function that maps 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.

In programming languages like C or C++ (and even Haskell), the operator `||` is often used to represent the `∨` operation.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ∨ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The logical *not* (¬) can be viewed as a unary function that maps a Boolean value to a Boolean value:

\[ \neg : \mathbb{B} \rightarrow \mathbb{B} \]

The ¬ operation simply flips a Boolean value.

In programming languages like C or C++, the operator ! is often used to represent the ¬ operation (in Haskell you can use the function not).
The logical implication (→) can be viewed as a function that maps 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.
The logical equivalence $\leftrightarrow$ can be viewed as a function that maps two Boolean values to a Boolean value:

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

In programming languages like C or C++ (and even Haskell), the operator $==$ is often used to represent the equivalence function as an operation.
### Boolean $\vee$ Function (exclusive or)

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \vee Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- The logical exclusive or $\vee$ can be viewed as a function that maps two Boolean values to a Boolean value:

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

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

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

$$1234_{16} \vee caf_{16} = 0001\ 0010\ 0011\ 0100_{2} \vee 1100\ 1010\ 1111\ 1110_{2} = 1101\ 1000\ 1100\ 1010_{2} = d8ca_{16}$$

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

$$d8ca_{16} \vee caf_{16} = 1101\ 1000\ 1100\ 1010_{2} \vee 1100\ 1010\ 1111\ 1110_{2} = 0001\ 0010\ 0011\ 0100_{2} = 1234_{16}$$

Apparently, it seems that $(X \vee Y) \vee Y = X$. To prove this property, we can simply write down a truth table:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \vee Y$</th>
<th>$(X \vee Y) \vee Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean \( \uparrow \) Function (not-and)

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( X \uparrow Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- The logical \( \textit{not-and} \) (nand) or \( \uparrow \) can be viewed as a function that maps two Boolean values to a Boolean value:
  \[ \uparrow : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \]
- The \( \uparrow \) function is also called Sheffer stroke.

- While we use the functions \( \land, \lor \), and \( \neg \) to introduce more complex Boolean functions, the Sheffer stroke is sufficient to derive all elementary Boolean functions from it.
- This is important for digital circuits since all you need are not-and gates.

The \( \textit{not-and} \) is a universal function since it can be used to derive all elementary Boolean functions.

- \( X \land Y = (X \uparrow Y) \uparrow (X \uparrow Y) \)

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( X \land Y )</th>
<th>( X \uparrow Y )</th>
<th>( (X \uparrow Y) \uparrow (X \uparrow Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( X \lor Y = (X \uparrow X) \uparrow (Y \uparrow Y) \)

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( X \lor Y )</th>
<th>( X \uparrow X )</th>
<th>( Y \uparrow Y )</th>
<th>( (X \uparrow X) \uparrow (Y \uparrow Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( \neg X = X \uparrow X \)

<table>
<thead>
<tr>
<th>( X )</th>
<th>( \neg X )</th>
<th>( X \land X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Function (not-or)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ↓ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- The logical *not-or* (nor) ↓ can be viewed as a function that maps two Boolean values to a Boolean value:

  \[ \downarrow : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \]

- The ↓ function is also called Quine arrow.

- The ↓ (nor) is like ↑ (nand) sufficient to derive all elementary Boolean functions.

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

- \( X \land Y = (X \downarrow X) \downarrow (Y \downarrow Y) \)
- \( X \lor Y = (X \downarrow Y) \downarrow (X \downarrow Y) \)
- \( \neg X = X \downarrow X \)
There are different symbols used to denote basic boolean functions. Here is an attempt to provide an overview. Note that authors sometimes mix symbols; there is no common standard notation.

<table>
<thead>
<tr>
<th>function</th>
<th>mnemonic</th>
<th>mathematics</th>
<th>engineering</th>
<th>C / C++</th>
<th>C / C++ (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$X$ and $Y$</td>
<td>$X \land Y$</td>
<td>$X \cdot Y$</td>
<td>$X &amp; Y$</td>
<td>$&amp;$</td>
</tr>
<tr>
<td>or</td>
<td>$X$ or $Y$</td>
<td>$X \lor Y$</td>
<td>$X + Y$</td>
<td>$X \mid Y$</td>
<td>$</td>
</tr>
<tr>
<td>not</td>
<td>not $X$</td>
<td>$\neg X$</td>
<td>$！X$</td>
<td>$~X$</td>
<td></td>
</tr>
<tr>
<td>implication</td>
<td>$X$ impl $Y$</td>
<td>$X \rightarrow Y$</td>
<td>$X \Rightarrow Y$</td>
<td>$X == Y$</td>
<td>$-$</td>
</tr>
<tr>
<td>equivalence</td>
<td>$X$ equiv $Y$</td>
<td>$X \leftrightarrow Y$</td>
<td>$X \equiv Y$</td>
<td>$X == Y$</td>
<td>$-$</td>
</tr>
<tr>
<td>exclusive or</td>
<td>$X$ xor $Y$</td>
<td>$X \oplus Y$</td>
<td>$X \oplus Y$</td>
<td>$X \oplus Y$</td>
<td>$-$</td>
</tr>
<tr>
<td>not and</td>
<td>$X$ nand $Y$</td>
<td>$X \uparrow Y$</td>
<td>$X \uparrow Y$</td>
<td>$X \uparrow Y$</td>
<td>$-$</td>
</tr>
<tr>
<td>not or</td>
<td>$X$ nor $Y$</td>
<td>$X \downarrow Y$</td>
<td>$X \downarrow Y$</td>
<td>$X \downarrow Y$</td>
<td>$-$</td>
</tr>
</tbody>
</table>
Boolean Functions

- Elementary Boolean functions ($\neg$, $\land$, $\lor$) can be composed to define more complex functions.
- An example of a composed function is
  \[ \varphi(X, Y) := \neg(X \land Y) \]
  which is a function $\varphi: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ and means “first compute the $\land$ of $X$ and $Y$, then apply the $\neg$ on the result you got from the $\land$”.
- Boolean functions can take a large number of arguments. Here is a function taking three arguments:
  \[ \varphi(X, Y, Z) := (\neg(X \land Y) \lor (Z \land Y)) \]
- 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 operations $\land$, $\lor$, and $\neg$:

- Implication
  \[ \rightarrow (X, Y) := \neg X \lor Y \]

- Equivalence
  \[ \leftrightarrow (X, Y) := (X \land Y) \lor (\neg X \land \neg Y) \]

- Exclusive or:
  \[ \checkmark(X, Y) := (X \land \neg Y) \lor (\neg X \land Y) \]

- Not and:
  \[ \uparrow (X, Y) := \neg (X \land Y) \]

- Not or:
  \[ \downarrow (X, Y) := \neg (X \lor Y) \]
Boolean Functions

Definition (Boolean function)

A **Boolean function** $\varphi$ is any function of the type $\varphi : \mathbb{B}^k \rightarrow \mathbb{B}$, where $k \geq 0$.

- The number $k$ of arguments is called the arity of the function.
- 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 arity-0 functions with the truth value constants 0 and 1.
- The truth table of a Boolean function with arity $k$ has $2^k$ rows. For a function with a large arity, truth tables become quickly unmanageable.

Boolean functions are interesting, since they can be used as computational devices. In particular, we can consider a computer CPU as collection of Boolean functions (e.g., a modern CPU with 64 inputs and outputs can be viewed as a sequence of 64 Boolean functions of arity 64: one function per output pin).

Another option to define a Boolean function is of course by writing down the truth table. We can define $\phi(X, Y, Z) := (\neg(X \land Y) \lor (Z \land Y))$ by filling out the following table:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
<th>$\neg(X \land Y) \lor (Z \land Y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

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

1. $(X \land Y) = (0 \land 0) = 0$
2. $(Z \land Y) = (0 \land 0) = 0$
3. $\neg(X \land Y) = \neg 0 = 1$
4. $(\neg(X \land Y) \lor (Z \land Y)) = 1 \lor 0 = 1$

Note that boolean formulas can be considered boolean polynomials. This becomes perhaps more obvious if one uses the alternate writing style where $X \cdot Y$ is used instead of $X \land Y$ and $X + Y$ is used instead of $X \lor Y$ and $\overline{X}$ is used instead of $\neg X$.

$\phi(X, Y, Z) = (X \cdot Y) + (Z \cdot Y)$
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 \( \varphi \) and \( \psi \) are Boolean formulas, then \( (\varphi \land \psi) \) is a Boolean formula.
2b If \( \varphi \) and \( \psi \) are Boolean formulas, then \( (\varphi \lor \psi) \) is a Boolean formula.
2c If \( \varphi \) is a Boolean formula, then \( \neg \varphi \) 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, \( \neg(X \land Y) \lor (Z \land Y) \) is a valid Boolean formula and \( \neg(X \land Y) \) is not.

In practice, we often use conventions that allow us to save parenthesis. For example, we may simply write \( (X \land Y \land Z) \) instead of \( (((X \land Y) \land Z) \lor (X \land (Y \land Z))) \).

Furthermore, we may write:
- \( (X \rightarrow Y) \) as a shorthand notation for \( \neg X \lor Y \)
- \( (X \leftrightarrow Y) \) as a shorthand notation for \( ((\neg X \lor Y) \land (\neg Y \lor X)) \)

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.
Definition (Semantics of Boolean formulas)

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

Basis of the inductive definition:

1a For every Boolean variable $X \in D$, $I^*(X) = I(X)$.

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

Because this generalized interpretation $I^*$ is the same as $I$ for Boolean variables $X \in D$, we say that $I^*$ extends $I$ from the domain $D$ to the domain $\Phi(D)$. Following common practice, we will use $I$ for the generalized interpretation too. Furthermore, since the set $D$ 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 \( \varphi \) and \( \psi \) in \( \Phi(D) \):

\[
\begin{align*}
2a & \quad \mathcal{I}^*(\varphi \land \psi) = \\
& \begin{cases} 
1 & \text{if } \mathcal{I}^*(\varphi) = 1 \text{ and } \mathcal{I}^*(\psi) = 1 \\
0 & \text{otherwise}
\end{cases} \\
2b & \quad \mathcal{I}^*(\varphi \lor \psi) = \\
& \begin{cases} 
1 & \text{if } \mathcal{I}^*(\varphi) = 1 \text{ or } \mathcal{I}^*(\psi) = 1 \\
0 & \text{otherwise}
\end{cases} \\
2c & \quad \mathcal{I}^!(\neg \varphi) = \\
& \begin{cases} 
1 & \text{if } \mathcal{I}^!(\varphi) = 0 \\
0 & \text{if } \mathcal{I}^!(\varphi) = 1
\end{cases}
\end{align*}
\]

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?
- Can we find a "simpler" boolean expression defining the same boolean function as a given boolean expression?
Section 18: Boolean Algebra Equivalence Laws

- Elementary Boolean Functions
- Boolean Functions and Formulas
- Boolean Algebra Equivalence Laws
- Normal Forms (CNF and DNF)
- Complexity of Boolean Formulas
- Boolean Logic and the Satisfiability Problem
**Tautology and contradiction**

**Definition (adapted interpretation)**
An interpretation $I : D \rightarrow \mathbb{B}$ is adapted to a Boolean formula $\varphi$ if all Boolean variables that occur in $\varphi$ are contained in $D$.

**Definition (tautologies and contradictions)**
A Boolean formula $\varphi$ is a tautology if for all interpretations $I$, which are adapted to $\varphi$, it holds that $I(\varphi) = 1$. A Boolean formula is a contradiction if for all interpretations $I$, which are adapted to $\varphi$, it holds that $I(\varphi) = 0$.

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 expression $\varphi$, $(\varphi \lor \neg \varphi)$ is a tautology and $(\varphi \land \neg \varphi)$ is a contradiction.
- If $\varphi$ is a tautology, then $\neg \varphi$ is a contradiction and vice versa.

For a complex Boolean formula $\varphi$ with many Boolean variables, it is not easy to find out whether a given Boolean formula 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 $\varphi$. Recall furthermore that if $\varphi$ contains $k$ Boolean variables, the truth table has $2^k$ rows. This becomes quickly impractical if $k$ grows. Unfortunately, there is no known general procedure to find out whether a given $\varphi$ is a tautology or 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 $I$ which is adapted to a Boolean formula $\varphi$ is said to satisfy the formula $\varphi$ if $I(\varphi) = 1$. A formula $\varphi$ is called satisfiable if there exists an interpretation which satisfies $\varphi$.

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 formulas as synonyms 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 $\varphi$, $\psi$ be two Boolean formulas. The formula $\varphi$ is equivalent to the formula $\psi$, written $\varphi \equiv \psi$, if for all interpretations $I$, which are adapted to both $\varphi$ and $\psi$, it holds that $I(\varphi) = I(\psi)$.

- 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.

A simple example is $(X \lor Y) \equiv (Y \lor X)$.

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

$$(X \lor Y) \equiv (X \lor Y) \land (Z \lor \neg Z)$$
Equivalence laws

Proposition (equivalence laws)

For any Boolean formulas \( \varphi, \psi, \chi \), the following equivalences hold:

1. \( \varphi \land 1 \equiv \varphi \) and \( \varphi \lor 0 \equiv \varphi \) (identity)
2. \( \varphi \lor 1 \equiv 1 \) and \( \varphi \land 0 \equiv 0 \) (domination)
3. \( (\varphi \land \varphi) \equiv \varphi \) and \( (\varphi \lor \varphi) \equiv \varphi \) (idempotency)
4. \( (\varphi \land \psi) \equiv (\psi \land \varphi) \) and \( (\varphi \lor \psi) \equiv (\psi \lor \varphi) \) (commutativity)
5. \( ((\varphi \land \psi) \land \chi) \equiv (\varphi \land (\psi \land \chi)) \) and \( ((\varphi \lor \psi) \lor \chi) \equiv (\varphi \lor (\psi \lor \chi)) \) (associativity)
6. \( \varphi \land (\psi \lor \chi) \equiv (\varphi \land \psi) \lor (\varphi \land \chi) \) and \( \varphi \lor (\psi \land \chi) \equiv (\varphi \lor \psi) \land (\varphi \lor \chi) \) (distributivity)
7. \( \neg \neg \varphi \equiv \varphi \) (double negation)
8. \( \neg (\varphi \land \psi) \equiv (\neg \varphi \lor \neg \psi) \) and \( \neg (\varphi \lor \psi) \equiv (\neg \varphi \land \neg \psi) \) (de Morgan’s laws)
9. \( \varphi \land (\varphi \lor \psi) \equiv \varphi \) and \( \varphi \lor (\varphi \land \psi) \equiv \varphi \) (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:

<table>
<thead>
<tr>
<th>( \varphi )</th>
<th>( \psi )</th>
<th>( \neg \varphi )</th>
<th>( \neg \psi )</th>
<th>( (\varphi \land \psi) )</th>
<th>( \neg (\varphi \land \psi) )</th>
<th>( (\neg \varphi \lor \neg \psi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

\( ((X \lor Y) \land (\neg Y \land Z)) \land Z = \ldots = X \land \neg Y \land Z \)

Alternatively, simplify the formula using a truth table.

---

142
Section 19: Normal Forms (CNF and DNF)

- Elementary Boolean Functions
- Boolean Functions and Formulas
- Boolean Algebra Equivalence Laws
- Normal Forms (CNF and DNF)
- Complexity of Boolean Formulas
- Boolean Logic and the Satisfiability Problem
Literals

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 logic and (product) of literals.

Definition (clause)
A clause (or sum term) is a literal or the logic or (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:
  - $X_1$ (this is a short form of $(1 \lor 1) \land (X_1 \lor 0)$)
  - $X_1 \land X_2$ (this is a short form of $(X_1 \lor X_1) \land (X_2 \lor X_2)$)
  - $X_1 \lor X_2$ (this is a short form of $(1 \lor 1) \land (X_1 \lor X_2)$)
  - $\neg X_1 \land (X_2 \lor X_3)$ (this is a 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:
  - $X_1$ (this is a short form of $(\neg 0 \land 0) \lor (X_1 \land 1)$)
  - $X_1 \land X_2$ (this is a short form of $(0 \land 0) \lor (X_1 \land X_2)$)
  - $X_1 \lor X_2$ (this is a 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.

If we order the boolean variables involved in a Boolean expression, we may write the monomials as $m_i$ where $i$ represents the number implied by the boolean variables or their negations.

For example,

$$\varphi(X, Y, Z) = (\neg X \land \neg Y \land Z) \lor (\neg X \land Y \land \neg Z) \lor (X \land \neg Y \land \neg Z) \lor (X \land Y \land \neg Z) \lor (X \land Y \land Z)$$

can be written as

$$\varphi(X, Y, Z) = m_1 \lor m_2 \lor m_4 \lor m_5 \lor m_6 \lor m_7$$

or with the alternative notation as

$$\varphi(X, Y, Z) = m_1 + m_2 + m_4 + m_5 + m_6 + m_7.$$
Equivalence of Normal Forms

**Proposition (CNF equivalence)**

*Every Boolean formula* \( \varphi \) *is equivalent to a Boolean formula* \( \chi \) *in conjunctive normal form.*

**Proposition (DNF equivalence)**

*Every Boolean formula* \( \varphi \) *is equivalent to a Boolean formula* \( \chi \) *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.

Here is a proof for the CNF equivalence. Since Boolean formulas are defined in an inductive way, we proof the equivalence by induction. The proof itself provides the basic idea how to convert a Boolean formula into CNF.

**Basis of induction:**

1a Let \( \varphi = X_i \) be a Boolean formula that just consists of a single Boolean variables \( X_i \). Obviously, \( \chi = (1 \lor 1) \land (\varphi \lor 0) \) is a formula in CNF that is equivalent to \( X_i \).

\[
\begin{align*}
\chi &= (1 \lor 1) \land (\varphi \lor 0) \\
&= 1 \land (\varphi \lor 0) \\
&= \varphi \lor 0 \\
&= \varphi \\
&= X_i 
\end{align*}
\]

(domination) (identity) (identity)

1b Let \( \varphi \) be a Boolean constant, i.e., \( \varphi = 0 \) or \( \varphi = 1 \). By the same construction as used in 1a, we obtain a formula in CNF.

**Induction step:**

2a Assume that \( \varphi = (\chi_1 \land \chi_2) \) is a conjunction of two formulas \( \chi_1 \) and \( \chi_2 \), which we assume to be in CNF. Due to the associativity of \( \land \) we can drop the parenthesis to obtain \( \chi = \chi_1 \land \chi_2 \) in CNF.

2b Assume that \( \varphi = (\chi_1 \lor \chi_2) \) is a disjunction of two formulas \( \chi_1 \) and \( \chi_2 \), which we assume to be in CNF.

2c Assume that \( \varphi = \neg \chi_1 \) is a negation of the formula \( \chi_1 \), which we assume to be in CNF.

(complete this)

(complete this)
### 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.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ∨ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 2nd row: \( \neg X \land Y \)
- 3rd row: \( X \land \neg Y \)
- \( \chi = (\neg X \land Y) \lor (X \land \neg Y) \)

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.
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.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ∨ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 1st row: $X \lor Y$
- 4th row: $\neg X \lor \neg Y$
- $\chi = (X \lor Y) \land (\neg X \lor \neg Y)$

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 will show that $(X \lor Y) \land (\neg X \lor \neg Y)$ is indeed the same as $(\neg X \land Y) \lor (X \land \neg Y)$:

\[
(X \lor Y) \land (\neg X \lor \neg Y) = ((X \lor Y) \land \neg X) \lor ((X \lor Y) \land \neg Y) \\
= (X \land \neg X) \lor (Y \land \neg Y) \lor (X \land \neg Y) \lor (Y \land \neg Y) \\
= 0 \lor (Y \land \neg Y) \lor 0 \\
= (\neg X \land Y) \lor (X \land \neg Y)
\]
Section 20: Complexity of Boolean Formulas

- Elementary Boolean Functions
- Boolean Functions and Formulas
- Boolean Algebra Equivalence Laws
- Normal Forms (CNF and DNF)
- Complexity of Boolean Formulas
- Boolean Logic and the Satisfiability Problem
Cost of Boolean Expressions and Functions

**Definition (cost of boolean expression)**
The cost $C(e)$ of a boolean expression $e$ is the number of operators in $e$.

**Definition (cost of boolean function)**
The cost $C(f)$ of a boolean function $f$ is the minimum cost of boolean expressions defining $f$, $C(f) = \min\{C(e)|e \text{ defines } f\}$.

- We can find expressions of arbitrary high cost for a given boolean function.
- How do we find an expression 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 $\{\land, \lor, \neg\}$. In some contexts, $\neg$ is not counted and only the number of $\land$ and $\lor$ 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 $\{\land, \lor, \neg\}$ operations and only count the number of $\land$ and $\lor$ operations unless stated otherwise.
Implicants and Prime Implicants

Definition (implicant)
A product term \( P \) of a Boolean function \( \varphi \) of \( n \) variables is called an implicant of \( \varphi \) if and only if for every combination of values of the \( n \) variables for which \( P \) is true, \( \varphi \) is also true.

Definition (prime implicant)
An implicant of a function \( \varphi \) is called a prime implicant of \( \varphi \) if it is no longer an implicant if any literal is deleted from it.

Definition (essential prime implicant)
A prime implicant of a function \( \varphi \) is called an essential prime implicant of \( \varphi \) if it covers a true output of \( \varphi \) that no combination of other prime implicants covers.

Observations:
- If an expression defining \( \varphi \) is in DNF, then every minterm of the DNF is an implicant of \( \varphi \).
- 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:

\[
\varphi(X, Y, Z) = (\neg X \land \neg Y \land \neg Z) \lor (X \land \neg Y \land \neg Z) \lor (X \land Y \land Z)
\]

\[
= (\neg Y \land \neg Z) \lor (X \land Z)
\]

- 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:

\[
(\neg X \land \neg Y \land \neg Z) \lor (X \land \neg Y \land \neg Z) = (\neg X \land X) \lor (\neg Y \land \neg Z)
\]

\[
= 0 \lor (\neg Y \land \neg Z)
\]

\[
= (\neg Y \land \neg Z)
\]

The resulting product term \(\neg Y \land \neg Z\) is still an implicant of \( \varphi \). 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 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 the 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** $\varphi(W, X, Y, Z) = m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{14} + m_{15}$

- Classify and sort minterms

<table>
<thead>
<tr>
<th>minterm</th>
<th>pattern</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_4$</td>
<td>0100</td>
<td>✓</td>
</tr>
<tr>
<td>$m_8$</td>
<td>1000</td>
<td>✓</td>
</tr>
<tr>
<td>$m_9$</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>$m_{10}$</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>$m_{12}$</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>$m_{11}$</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>$m_{14}$</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>$m_{15}$</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

- Combination steps

<table>
<thead>
<tr>
<th>minterm</th>
<th>pattern</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_4$</td>
<td>0100</td>
<td>✓</td>
</tr>
<tr>
<td>$m_8$</td>
<td>1000</td>
<td>✓</td>
</tr>
<tr>
<td>$m_9$</td>
<td>1001</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{10}$</td>
<td>1010</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{12}$</td>
<td>1100</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{11}$</td>
<td>1011</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{14}$</td>
<td>1110</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{15}$</td>
<td>1111</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minterms</th>
<th>pattern</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{4,12}$</td>
<td>-100</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{8,9}$</td>
<td>10-0</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{8,10}$</td>
<td>1-00</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{8,12}$</td>
<td>1-00</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{8,9,10,11}$</td>
<td>10--</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{8,10,12,14}$</td>
<td>1--0</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{9,11}$</td>
<td>10-1</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{10,11}$</td>
<td>101-</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{10,14}$</td>
<td>1-10</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{12,14}$</td>
<td>11-0</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{10,11,14,15}$</td>
<td>1-1-</td>
<td>✓</td>
</tr>
</tbody>
</table>

- This gives us four prime implicants:
  
  - $m_{4,12} = (X \land \neg Y \land \neg Z)$
  - $m_{8,9,10,11} = (W \land \neg X)$
  - $m_{8,10,12,14} = (W \land \neg Z)$
  - $m_{10,11,14,15} = (W \land Y)$
### 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 additional 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 columns 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.

<table>
<thead>
<tr>
<th></th>
<th>( m_4 )</th>
<th>( m_8 )</th>
<th>( m_9 )</th>
<th>( m_{10} )</th>
<th>( m_{11} )</th>
<th>( m_{12} )</th>
<th>( m_{14} )</th>
<th>( m_{15} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_{4,12} )</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>( m_{8,9,10,11} )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>( m_{8,10,12,14} )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>( m_{10,11,14,15} )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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 \( \varphi' (W, X, Y, Z) = (X \land \neg Y \land \neg Z) \lor (W \land \neg X) \lor (W \land Y) \). The minimal expression \( \varphi' \) uses 6 operations (out of \{\land, \lor\}). The original expression \( \varphi \) used \( 8 \cdot 3 + 7 = 31 \) operations (out of \{\land, \lor\}).
Section 21: Boolean Logic and the Satisfiability Problem

- Elementary Boolean Functions
- Boolean Functions and Formulas
- Boolean Algebra Equivalence Laws
- Normal Forms (CNF and DNF)
- Complexity of Boolean Formulas

- Boolean Logic and the Satisfiability Problem
Logic Statements

- A common task is to decide whether statements of the form
  \[ \text{if premises } P_1 \text{ and } \ldots \text{ and } P_m \text{ hold, then conclusion } C \text{ holds} \]
  are true.

- 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.

First order logic (also called predicate logic) is an extension of propositional logic (Boolean logic) that adds quantified variables and predicates that contain variables. First order logic is more powerful than propositional logic but unfortunately also more difficult to handle. Logic programming languages hence often use a subset of first order logic in order to be efficient.
Tautology and Satisfiability

• Recall that a Boolean formula $\tau$ 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

$$\tau = (\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi$$  \hspace{1cm} (1)

is a tautology, we may check whether

$$\tau' = \neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi)$$ \hspace{1cm} (2)

is unsatisfiable.

• If we show that $\tau'$ is satisfiable, we have disproven $\tau$.
Tautology and Satisfiability

- Since $\varphi \rightarrow \psi \equiv \neg(\varphi \land \neg\psi)$, we can rewrite the formulas as follows:

$$
\tau = (\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi = \neg(\varphi_1 \land \ldots \land \varphi_m \land \neg\psi) \quad (3)
$$

$$
\tau' = \neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi) = (\varphi_1 \land \ldots \land \varphi_m \land \neg\psi) \quad (4)
$$

- To disprove $\tau$, it is often easier to prove that $\tau'$ is satisfiable.

- Note that $\tau'$ 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 $\tau'$ 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, $\tau$ is disproven, otherwise, if there is no such interpretation, then $\tau$ 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, \( P \subseteq NP \). The big question is whether \( P = NP \) holds. (Please recall that \( NP \) stands for non-deterministic 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 prize.

For further information:

- [https://en.wikipedia.org/wiki/Millennium_Prize_Problems](https://en.wikipedia.org/wiki/Millennium_Prize_Problems)
Part V

Computer Architecture

This part takes a look at how you can put boolean functions to work by studying how you can turn them into digital circuits or how you can describe digital circuits using boolean functions.

We first study how to design combinatorial digital circuits for adding numbers or comparing numbers. We then move to sequential digital circuits and we study how to design latches and flip flops that can store bits.

With these ingredients prepared, we can look at the design of a von Neumann computer, which has a central processing unit, a memory system, and IO devices ann interconnected via a data, address, and control bus. We use a very simplistic CPU architecture in order to get an idea how assembly programs translate into program code in memory and how a program stored in memory is executed in a fetch, decode, execute cycle.

Given the time constraints, we will only touch on many interesting topics here. If you want to dive deeper, consider taking a digital design module in the future.
Section 22: Combinational Digital Circuits

- Combinational Digital Circuits
- Sequential Digital Circuits
- Von Neumann Computer Architecture
Recall elementary boolean operations and functions

- Recall the elementary boolean operations AND (\( \land \)), OR (\( \lor \)), and NOT (\( \neg \)) as well as the boolean functions XOR (\( \oplus \)), NAND (\( \uparrow \)), and NOR (\( \downarrow \)).

\[
X \lor Y := (X \lor Y) \land \neg(X \land Y) \\
X \uparrow Y := \neg(X \land Y) \\
X \downarrow Y := \neg(X \lor Y)
\]

- For each of these elementary boolean operations or functions, we can construct digital gates, for example, using transistors in Transistor-Transistor Logic (TTL).
- Note: NAND and NOR gates are called universal gates since all other gates can be constructed by using multiple NAND or NOR gates.

It is essential to recall the basic boolean operations and functions. The following tables summarize the truth tables.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X &amp; Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \lor Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>\neg X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1: Truth tables for the elementary operations AND, OR, and NOT

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \oplus Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \uparrow Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \downarrow Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2: Truth tables for the elementary functions XOR, NAND, and NOR

We will now introduce symbols for logic gates that implement these basic boolean operations and functions. Afterwards, we will design a logic circuit that can add N-bit digital numbers.

Further information:
Logic gates implementing logic functions

- **NOT** (¬)
  ![NOT gate](image)

- **AND** (∧)
  ![AND gate](image)

- **OR** (∨)
  ![OR gate](image)

- **XOR** (˙∨)
  ![XOR gate](image)

- **NAND** (↑)
  ![NAND gate](image)

- **NOR** (↓)
  ![NOR gate](image)

- There are different sets of symbols for logic gates (do not get confused if you look into other sources of information).

- The symbols used here are the ANSI (American National Standards Institute) symbols.

While logic gates implement elementary boolean operations, they introduce another important property: Logic gates take some time to propagate input signals to output signals. The *gate delay* is the length of the time interval starting when the input to a logic gates becomes stable and valid and ending when the output of that logic gate becomes stable and valid.
## Combinational Digital Circuits

### Definition (combinational digital circuit)

A *combinational digital circuit* implements pure boolean functions where the results only depend on the inputs.

- Examples of elementary combinational digital circuits are circuits to add $n$-bit numbers, to multiply $n$-bit numbers, or to compare $n$-bit numbers.
- Combinational digital circuits are pure since their behavior solely depends on the well-defined inputs of the circuit.
## Addition of decimal and binary numbers

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0010</td>
<td>3</td>
<td>0011</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
<td>+</td>
<td>0101</td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>6</td>
<td>0110</td>
<td>11</td>
<td>1011</td>
</tr>
</tbody>
</table>

- We are used to add numbers in the decimal number system.
- Adding binary numbers is essentially the same, except that we only have the digits 0 and 1 at our disposal and “carry overs” are much more frequent.
Adding two bits (half adder)

- The half adder adds two single binary digits $A$ and $B$.
- It has two outputs, sum ($S$) and carry ($C$).

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$S = A \lor B$

$C = A \land B$

The gate delay $t_g$ is the time it takes for the output signal to be a stable reflection of the two input signals. Assuming that every gate has the same gate delay, the half adder works with a constant gate delay of $t_{ha} = t_g$.

Further information:
Adding two bits (full adder)

- A full adder adds two single bit digits $A$ and $B$ and accounts for a carry bit $C_{in}$.
- It has two outputs, sum ($S$) and carry ($C_{out}$).

$$S = (A \lor B) \lor C_{in}$$

$$C_{out} = (A \land B) \lor (C_{in} \land (A \lor B))$$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_{in}$</th>
<th>$C_{out}$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Assuming that every gate has the same gate delay $t_g$, the full adder works with a constant gate delay of $t_{fa} = 3 \cdot t_g$. Note that the carry in signal only has a gate delay of $2 \cdot t_g$ to the carry out signal.

Further information:
Adding N Bits (ripple carry adder)

- And N-bit adder can be created using multiple full adders.
- Each full adder inputs a \( C_{in} \), which is the \( C_{out} \) of the previous adder.
- Each carry bit “ripples” to the next full adder.

The layout of a ripple-carry adder is simple. However, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated by the previous full adder. Assuming that every gate has the same gate delay \( t_g \), the n-bit ripple-carry adder has a gate delay of \( t_{ra} = n \cdot t_{fa} = 3n \cdot t_g \). If we built an n-bit ripple-carry adder from elementary logic gates, then the slow path is the carry propagation, which is just \( 1 + 2n \cdot t_g \). For a 64-bit adder, we get a gate delay of 129 gate times, which is not nice. The delay of an n-bit ripple-carry adder grows linearly with the number of bits.
Adding N Bits (carry-lookahead adder)

- A carry-lookahead adder uses a special circuit to calculate the carry bits.
- Half adder are used add the input bits and to feed the carry-lookahead circuit.
- Half adder are used to finally add the carry bits.

It is possible to design adders that reduce the gate delay. A classic example is the carry lookahead adder. Lets consider how the carry bit works: The carry bit is either the result of adding \( a_i \) and \( b_i \) or it is the result of \( c_i = 1 \) and either \( a_i = 1 \) or \( b_i = 1 \).

\[
\begin{align*}
c_{i+1} &= (a_i \land b_i) \lor ((a_i \lor b_i) \land c_i) = g_i \lor (p_i \land c_i) \\
g_i &= a_i \land b_i \\
p_i &= a_i \lor b_i
\end{align*}
\]

The function \( g_i \) “generates” a carry bit and the function \( p_i \) “propagates” a carry bit. Note that the equation above gives us a recursive definition how \( c_i \) can be calculated. From this, we can derive concrete expressions for the carry bits.

\[
\begin{align*}
c_0 &= c_0 \\
c_1 &= g_0 \lor (p_0 \land c_0) \\
c_2 &= g_1 \lor (p_1 \land c_1) \\
&= g_1 \lor (p_1 \lor (g_0 \lor (p_0 \land c_0))) \\
&= g_1 \lor (p_1 \lor g_0) \lor (p_1 \land p_0 \land c_0) \\
c_3 &= g_2 \lor (p_2 \land c_2) \\
&= g_2 \lor (p_2 \lor (g_1 \lor (p_1 \lor g_0) \lor (p_1 \land p_0 \land c_0))) \\
&= g_2 \lor (p_2 \lor g_1) \lor (p_2 \lor p_1 \land g_0) \lor (p_2 \land p_1 \land p_0 \land c_0) \\
c_4 &= g_3 \lor (p_3 \land c_3) = \ldots \\
&= g_4 \lor (p_3 \lor g_2) \lor (p_3 \lor p_2 \lor g_1) \lor (p_3 \lor p_2 \lor p_1 \lor g_0) \lor (p_3 \lor p_2 \lor p_1 \lor p_0 \land c_0)
\end{align*}
\]

Note that \( g_i \) and \( p_i \) are exactly the functions of our half adder \((C = g_i, S = p_i)\). Hence, we can use \( n \) half adders to produce \( g_0, \ldots, g_{n-1} \) and \( p_0, \ldots, p_{n-1} \). We then create a circuit to calculate \( c_1, \ldots, c_n \) following the expansion scheme above and we finally use \( n \) half adders to add the carry bits \( c_i \) to \( p_i \) in order to produce the sum bits \( s_i \). The overall delay of the carry lookahead adder becomes \( t_{cla} = 2 \cdot t_{ha} + t_{cc} \) where \( t_{cc} \) is the delay of the digital circuit calculating the carry bits. If our gates are restricted to two inputs, we can calculate the logical ands in a tree-like fashion, which gives us \( t_{cc} = \log(n) \cdot t_g \). Putting things together, the overall delay becomes \( t_{cla} = 2 \cdot t_g + \log(n) \cdot t_g \), i.e., the delay grows logarithmically with the number of bits. The price for this improvement is a more complex digital circuit.
Further information:

Multiplication of Decimal and Binary Numbers

\[
\begin{array}{c}
11 \times 13 \\
+ \quad 33 \\
\hline
143
\end{array}
\quad \begin{array}{c}
1011 \times 1101 \\
+ \quad 1011 \\
+ \quad 0000 \\
\hline
10001111
\end{array}
\]

- We can reduce multiplication to repeated additions.
- Multiplication by \(2^n\) is a left shift by \(n\) positions.

The multiplication of a two bits is simply a logical and operation.

The multiplication of two binary numbers can be reduced to a sequence of additions. The partial products are either 0 or the multiplicand.

Straight-forward digital multiplication circuits are conceptually simply but relatively complex in terms of the number of gates needed. Optimizations can be used for special cases. For example, multiplication by 2 is simply a left shift of bits. We will not study multiplication circuits further here.
Section 23: Sequential Digital Circuits

- Combinational Digital Circuits

- Sequential Digital Circuits

- Von Neumann Computer Architecture
## Sequential Digital Circuits

**Definition (sequential digital circuit)**
A *sequential digital circuit* implements a non-pure boolean functions where the results depend on both the inputs and the current state of the circuit.

**Definition (asynchronous sequential digital circuit)**
A sequential digital circuit is *asynchronous* if the state of the circuit and the results can change anytime in response to changing inputs.

**Definition (synchronous sequential digital circuit)**
A sequential digital circuit is *synchronous* if the state of the circuit and the results can change only at discrete times in response to a clock signal.
Basic Properties of Memory

Memory should have at least three properties:
1. It should be able to hold a value.
2. It should be possible to read the value that was saved.
3. It should be possible to change the value that was saved.

We start with the simplest case, a one-bit memory:
1. It should be able to hold a single bit.
2. It should be possible to read the bit that was saved.
3. It should be possible to change the bit that was saved.
SR Latch using NOR Gates

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

• Setting the inputs to \( S = 1 \land R = 0 \) sets the stored bit \((Q = 1)\).
• Setting the inputs to \( R = 1 \land S = 0 \) clears the stored bit \((Q = 0)\).
• The stored bit does not change while \( R = 0 \land S = 0 \).

The SR latch has two inputs \( S \) and \( R \) and two outputs \( Q \) and \( \bar{Q} \). The outputs \( Q \) and \( \bar{Q} \) feedback into the circuit and hence the value of the outputs \( Q \) and \( \bar{Q} \) depends on the two inputs and the previous values of \( Q \) and \( \bar{Q} \). We can write this as

\[
Q_{\text{next}} = R \downarrow \bar{Q}_{\text{current}} \\
\bar{Q}_{\text{next}} = S \downarrow Q_{\text{current}}
\]

Let’s try to understand the behavior of this circuit by looking at the different possible cases:

- \( S = 0 \land R = 0 \): The values of \( Q \) and \( \bar{Q} \) do not change.
  \[
  Q_{\text{next}} = R \downarrow \bar{Q}_{\text{current}} = 0 \downarrow \bar{Q}_{\text{current}} = \bar{Q}_{\text{current}} \\
  \bar{Q}_{\text{next}} = S \downarrow Q_{\text{current}} = 0 \downarrow Q_{\text{current}} = Q_{\text{current}}
  \]

- \( S = 1 \land R = 0 \): The value of \( \bar{Q} \) becomes 0 and subsequently the value of \( Q \) becomes 1.
  \[
  \bar{Q}_{\text{next}} = S \downarrow Q_{\text{current}} = 1 \downarrow Q_{\text{current}} = 0 \\
  Q_{\text{next}} = R \downarrow \bar{Q}_{\text{current}} = 0 \downarrow \bar{Q}_{\text{current}} = 1
  \]

- \( S = 0 \land R = 1 \): The value of \( Q \) becomes 1 and subsequently the value of \( \bar{Q} \) becomes 0.
  \[
  Q_{\text{next}} = R \downarrow \bar{Q}_{\text{current}} = 1 \downarrow \bar{Q}_{\text{current}} = 0 \\
  \bar{Q}_{\text{next}} = S \downarrow Q_{\text{current}} = 0 \downarrow Q_{\text{current}} = 1
  \]

- \( S = 1 \land R = 1 \): This may lead to oscillation and hence this input should not occur.

To properly understand sequential circuits, it is often useful to look at timing diagrams.
SR Latch using NAND Gates

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>̅Q</th>
<th>Q</th>
<th>̅Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Setting the inputs to \( S = 0 \land R = 1 \) sets the stored bit \((Q = 1)\).
- Setting the inputs to \( R = 0 \land S = 1 \) clears the stored bit \((Q = 0)\).
- The stored bit does not change while \( R = 1 \land S = 1 \).

The \( \overline{SR} \) latch behaves like an SR latch except that the input signals are inverted: The latch does not change as long as both \( S \) and \( R \) are high. If \( S \) goes low while \( R \) remains high, the bit in the latch gets set to high. Similarly, if \( R \) goes low while \( S \) remains high, the bit in the latch gets cleared. \( S \) and \( R \) should not go together low.
Gated SR Latch using NAND Gates

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>$\bar{Q}$</th>
<th>Q</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- The control input $C$ enables the SR latch.
There are different ways to create a D latch. The basic idea of a D latch is that the \( S \) and \( R \) signals are created from a data signal \( D \) and a control signal \( C \). Real-world circuits on the market often also expose the \( R \) and \( S \) (or \( \overline{R} \) and \( \overline{S} \)) lines in order to provide a direct way to control the latch.

State diagrams provide another way to describe the behaviour of latches (or more complex digital circuits). In a state diagram, we first draw a node for each state that the circuit can be in. The D latch has the two states \( Q = 0 \) and \( Q = 1 \) (we ignore \( \overline{Q} \) since it is just the negation of \( Q \)). We then draw labeled arrows indicating how the state changes when certain inputs are received. Below is a state diagram for the D latch with the inputs \( D \) and \( C \).

A common symbol for a D latch is show below:
D Flip-Flop (Master Slave)

- Edge-triggered D flip-flops propagate changes from the master to the slave latch on either the rising edge of a clock signal or the falling edge of the clock signal.

A major problem with latches is timing in situations where the bit stored in the latch is read (i.e., to provide input to an adder circuit) and also being written (i.e., to store the output produced by the adder circuit). This can only work with very tight constraints on the gate delays. Flip-flops solve this problem since one latch (the master latch) can receive a new bit while the other latch (the slave latch) still reports the old bit.

A positive edge-triggered D flip-flop has the master latch enabled when \( C = 0 \). The enabled master latch follows the input \( D \). When the clock changes to \( C = 1 \), the master latch is getting disabled and the slave latch becomes enabled. The bit stored in the master latch now propagates into the slave latch on the rising edge (when \( C \) changes from 0 to 1).

A negative edge-triggered D flip-flop has the master latch enabled when \( C = 1 \). The enabled master latch follows the input \( D \). When the clock changes to \( C = 0 \), the master latch is getting disabled and the slave latch becomes enabled. The bit stored in the master latch now propagates into the slave latch on the falling edge (when \( C \) changes from 1 to 0).

A D flip-flop can also be constructed out of three SR latches using a circuit only using NAND gates.

A common symbol for a positive edge triggered D flip-flop is show below:
Section 24: Von Neumann Computer Architecture

- Combinational Digital Circuits
- Sequential Digital Circuits
- Von Neumann Computer Architecture
Von Neumann computer architecture

- Control unit contains an instruction register and a program counter
- Arithmetic/logic unit (ALU) performs integer arithmetic and logical operations
- Program instructions and data is stored in a memory unit
- Processor registers provide small amount of storage as part of a central processing unit
- The central processing unit (CPU) carries out the actual computations

Further information:
The **data bus** transports data (primarily between registers and main memory).

The **address bus** selects which memory cell is being read or written.

The **control bus** activates components and steers the data flow over the data bus and the usage of the address bus.

**Parallel busses:**
- Carry data words in parallel on multiple wires.
- Are highly efficient if the distance to cover is small.

**Serial busses:**
- Carry data in bit-serial form over simple wires.
- Are highly efficient if the distance to cover is long.
- Inside the central processing unit, busses are usually parallel.

**Examples of parallel busses:**
- **PATA (IDE)** - Parallel Advanced Technology Attachment (primarily used to connect hard drives)
- **SCSI** - Small Computer System Interface (primarily used to connect external storage devices)
- **PCI** - Peripheral Component Interconnect (primarily used internally to interconnect computer components)

**Examples of serial busses:**
- **SATA** - Serial ATA (primarily used to connect hard drives)
- **PCI Express** - Peripheral Component Interconnect Express (serial bus designed to replace PCI)
- **RS232** (very old serial bus widely used for low-speed communication)
- **USB** - Universal Serial Bus (primarily used to connect external devices)
- **Thunderbold** - (combines PCI Express with a video interface)

Further information:
This is a model of a very simple central processing unit.

- The Accumulator (a single register) is used to carry out all operations.
- The Accumulator is connected to the Arithmetic/Logic Unit, which consists of digital circuits (such as an 8-bit adder).
- The Instruction Register holds the current instruction being executed.
- The Decoder determines from the Instruction which digital circuit needs to be activated.
- The Multiplexor controls whether the operand is read from a memory cell or out of the Instruction itself.
- The Program Counter holds the address of the current instruction in the Random Access Memory (RAM).
- The +1 circuit increments the Program Counter after each instruction.

The simple model can be improved in several ways:

- Larger memory sizes. What are the changes that necessary to support larger memory sizes?
- Additional instructions: What are good instructions to add to the CPU’s instruction set?
- Function calls: How can we support the calling of functions (and the return from functions) in a way that support recursion?
while True:
    advance_program_counter();
    instruction = fetch();
    decode(instruction);
    execute(instruction);

- The CPU runs in an endless loop fetching instructions, decoding them, and executing them.
- The set of instructions a CPU can execute is called the CPU's machine language.

- Typical instructions are to add two N-bit numbers, to test whether a certain register is zero, or to jump to a certain position in the ordered list of machine instructions.
- An assembly programming language is a mnemonic representation of machine code.

Further information:
- http://en.wikipedia.org/wiki/Assembly_language
Simple Machine Language

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>LOAD</td>
<td>Load the value of the operand into the accumulator</td>
</tr>
<tr>
<td>010</td>
<td>STORE</td>
<td>Store the value of the accumulator at the address specified by the operand</td>
</tr>
<tr>
<td>011</td>
<td>ADD</td>
<td>Add the value of the operand to the accumulator</td>
</tr>
<tr>
<td>100</td>
<td>SUB</td>
<td>Subtract the value of the operand from the accumulator</td>
</tr>
<tr>
<td>101</td>
<td>EQUAL</td>
<td>If the value of the operand equals the value of the Accumulator, skip the next instruction</td>
</tr>
<tr>
<td>110</td>
<td>JUMP</td>
<td>Jump to a specified instruction by setting the program counter to the value of the operand</td>
</tr>
<tr>
<td>111</td>
<td>HALT</td>
<td>Stop execution</td>
</tr>
</tbody>
</table>

- Each instruction of the machine language is encoded into 8 bits:
  - 3 bits are used for the op-code
  - 1 bit indicates whether the operand is a constant (1) or a memory address (0)
  - 4 bits are used to carry a constant or a memory address (the operand)

Further information:
### Program #1 in our simple machine language

<table>
<thead>
<tr>
<th>#</th>
<th>Machine Code</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001 0010</td>
<td></td>
<td>LOAD #2</td>
</tr>
<tr>
<td>1</td>
<td>010 0110</td>
<td>STORE 13</td>
<td>Store the value of the accumulator in memory location 13</td>
</tr>
<tr>
<td>2</td>
<td>001 0110</td>
<td>LOAD #5</td>
<td>Load the value 5 into the accumulator</td>
</tr>
<tr>
<td>3</td>
<td>010 0110</td>
<td>STORE 14</td>
<td>Store the value of the accumulator in memory location 14</td>
</tr>
<tr>
<td>4</td>
<td>001 0110</td>
<td>LOAD 13</td>
<td>Load the value of memory location 13 into the accumulator</td>
</tr>
<tr>
<td>5</td>
<td>011 0110</td>
<td>ADD 14</td>
<td>Add the value of memory location 14 to the accumulator</td>
</tr>
<tr>
<td>6</td>
<td>010 0111</td>
<td>STORE 15</td>
<td>Store the value of the accumulator in memory location 15</td>
</tr>
<tr>
<td>7</td>
<td>111 0000</td>
<td>HALT</td>
<td>Stop execution</td>
</tr>
</tbody>
</table>

- An animation of the execution of this program can be found here: [http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html](http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html)
- What is the equivalent C program?

Below is a trace of the execution of the program (showing the initial memory content, the machine instructions executed (PC = program counter, IR = instruction register, ACC = accumulator), and the final memory content. Numbers starting with 0x are in hexadecimal notation.

\[
\text{MEM } = 0x32 \ 4d \ 35 \ 4e \ 2d \ 6e \ 4f \ e0 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \\
\]

\[
1: \ PC= 0 \ IR= 0x32 \ ACC= 0x02 ; \text{LOAD } #2  \\
2: \ PC= 1 \ IR= 0x4d \ ACC= 0x02 ; \text{STORE } 13  \\
3: \ PC= 2 \ IR= 0x35 \ ACC= 0x05 ; \text{LOAD } #5  \\
4: \ PC= 3 \ IR= 0x4e \ ACC= 0x05 ; \text{STORE } 14  \\
5: \ PC= 4 \ IR= 0x2d \ ACC= 0x02 ; \text{LOAD } 13  \\
6: \ PC= 5 \ IR= 0x6e \ ACC= 0x07 ; \text{ADD } 14  \\
7: \ PC= 6 \ IR= 0x4f \ ACC= 0x07 ; \text{STORE } 15  \\
8: \ PC= 7 \ IR= 0xe0 \ ACC= 0x07 ; \text{HALT}  \\
\]

\[
\text{MEM } = 0x32 \ 4d \ 35 \ 4e \ 2d \ 6e \ 4f \ e0 \ 00 \ 00 \ 00 \ 00 \ 02 \ 05 \ 07 \\
\]
### Program #2 in our simple machine language

<table>
<thead>
<tr>
<th>#</th>
<th>Machine Code</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001 1 0101</td>
<td>LOAD #5</td>
<td>Load the value 5 into the accumulator</td>
</tr>
<tr>
<td>1</td>
<td>010 0 1111</td>
<td>STORE 15</td>
<td>Store the value of the accumulator in memory location 15</td>
</tr>
<tr>
<td>2</td>
<td>001 0 0000</td>
<td>LOAD #0</td>
<td>Load the value 0 into the accumulator</td>
</tr>
<tr>
<td>3</td>
<td>101 0 1111</td>
<td>EQUAL 15</td>
<td>Skip next instruction if accumulator equal to memory location 15</td>
</tr>
<tr>
<td>4</td>
<td>110 1 0110</td>
<td>JUMP #6</td>
<td>Jump to instruction 6 (set program counter to 6)</td>
</tr>
<tr>
<td>5</td>
<td>111 0 0000</td>
<td>HALT</td>
<td>Stop execution</td>
</tr>
<tr>
<td>6</td>
<td>011 1 0001</td>
<td>ADD #1</td>
<td>Add the value 1 to the accumulator</td>
</tr>
<tr>
<td>7</td>
<td>110 1 0011</td>
<td>JUMP #3</td>
<td>Jump to instruction 3 (set program counter to 3)</td>
</tr>
</tbody>
</table>

- An animation of the execution of this program can be found here: [http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html](http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html)
- What is the equivalent C program?

Below is a trace of the execution of the program using the same notation as used before.

MEM = 0x35 4f 30 af d6 e0 71 d3 00 00 00 00 00 00 00 05

```plaintext
1: PC= 0  IR= 0x35  ACC= 0x05  ; LOAD #5
2: PC= 1  IR= 0x4f  ACC= 0x05  ; STORE 15
3: PC= 2  IR= 0x30  ACC= 0x00  ; LOAD #0
4: PC= 3  IR= 0xaf  ACC= 0x00  ; EQUAL 15
5: PC= 4  IR= 0xd6  ACC= 0x00  ; JUMP #6
6: PC= 5  IR= 0xef  ACC= 0x01  ; ADD #1
7: PC= 6  IR= 0xd3  ACC= 0x01  ; JUMP #3
8: PC= 7  IR= 0xaf  ACC= 0x01  ; EQUAL 15
9: PC= 8  IR= 0xd6  ACC= 0x01  ; JUMP #6
10: PC= 9 IR= 0x71  ACC= 0x02  ; ADD #1
11: PC=10 IR= 0xd3  ACC= 0x02  ; JUMP #3
12: PC=11 IR= 0xaf  ACC= 0x02  ; EQUAL 15
13: PC=12 IR= 0xd6  ACC= 0x02  ; JUMP #6
14: PC=13 IR= 0xe7  ACC= 0x03  ; ADD #1
15: PC=14 IR= 0xd3  ACC= 0x03  ; JUMP #3
16: PC=15 IR= 0xaf  ACC= 0x03  ; EQUAL 15
17: PC=16 IR= 0xd6  ACC= 0x03  ; JUMP #6
18: PC=17 IR= 0x71  ACC= 0x04  ; ADD #1
19: PC=18 IR= 0xd3  ACC= 0x04  ; JUMP #3
20: PC=19 IR= 0xaf  ACC= 0x04  ; EQUAL 15
21: PC=20 IR= 0xd6  ACC= 0x04  ; JUMP #6
22: PC=21 IR= 0x71  ACC= 0x05  ; ADD #1
23: PC=22 IR= 0xd3  ACC= 0x05  ; JUMP #3
24: PC=23 IR= 0xaf  ACC= 0x05  ; EQUAL 15
25: PC=24 IR= 0xe0  ACC= 0x05  ; HALT

MEM = 0x35 4f 30 af d6 e0 71 d3 00 00 00 00 00 00 00 05
```
Part VI

System Software

System software is all the software needed to make a barebone computer do useful things. This includes the operating system kernel, compilers, linkers, and so on.

In this part, we first take a look at compilers and interpreters in order to get a rough idea how they function and differ from each other. Afterwards, we take a look at some of the services provided by operating systems in order to understand how the (concurrent) execution of multiple programs can be organized. We will also touch on some of the higher level abstractions operating system kernels usually provide to programs.
Section 25: Interpreter and Compiler

Interpreter and Compiler

Operating Systems
Are there better ways to write machine or assembler code?

- Observations:
  - Writing machine code or assembler code is difficult and time consuming.
  - Maintaining machine code or assembler code is even more difficult and time consuming (and most cost is spent on software maintenance).
  - A high-level programming language is a programming language with strong abstraction from the low-level details of the computer.
  - Rather than dealing with registers and memory addresses, high-level languages deal with variables, arrays, objects, collections, complex arithmetic or boolean expressions, subroutines and functions, loops, threads, locks, and other abstract computer science concepts, with a focus on usability over optimal program efficiency.

Higher-level programming languages are often designed (at least originally) to be implemented either with a compiler or an interpreter.

Examples of (typically) compiled programming languages:
- C, C++, Java, C#, Objective C, Rust, Go, Pascal, Fortran, Cobol, . . .

Examples of (typically) interpreted programming languages:
- Python, PHP, JavaScript, Perl, Basic, Bash, . . .

Note that an increasing number of (typically) interpreted languages are not executed by pure interpreters anymore. Many modern interpreters compile the source code into an intermediate byte-code that is executed by a byte-code interpreter. However, this internal compilation step is usually transparent for the user; the languages keep their highly interactive interface and they do not require an explicit compilation step.
Simple C program to add two numbers

/* C source code
(C is a compiled procedural programming language) */

int main()
{
    int a = 5;
    int b = 2;
    int c = a + b;
    return c;
}

Assuming the source code is in the file 'add.c', type the following commands into your shell:

gcc -o add add.c
./add
echo $?

The first command calls the gcc compiler and instructs it to compile and link the source code contained in add.c into an executable file add. The second command executes the program add stored in the current directory. The third command prints (echoes) the result of the last program execution. You should see the number 7.
Disassembled machine code (without optimizations)

# compile without optimization (gcc) and look at the machine code
# gcc (Debian 4.7.2-5) 4.7.2 on Linux

00000000004004ac <main>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ac:</td>
<td>push %rbp</td>
</tr>
<tr>
<td>4004ad:</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>4004b0:</td>
<td>movl $0x5,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004b7:</td>
<td>movl $0x2,-0x8(%rbp)</td>
</tr>
<tr>
<td>4004be:</td>
<td>mov -0x8(%rbp),%eax</td>
</tr>
<tr>
<td>4004c1:</td>
<td>mov -0x4(%rbp),%edx</td>
</tr>
<tr>
<td>4004c4:</td>
<td>add %edx,%eax</td>
</tr>
<tr>
<td>4004c6:</td>
<td>mov %eax,-0xc(%rbp)</td>
</tr>
<tr>
<td>4004c9:</td>
<td>mov -0xc(%rbp),%eax</td>
</tr>
<tr>
<td>4004cc:</td>
<td>pop %rbp</td>
</tr>
<tr>
<td>4004cd:</td>
<td>retq</td>
</tr>
<tr>
<td>4004ce:</td>
<td>nop</td>
</tr>
<tr>
<td>4004cf:</td>
<td>nop</td>
</tr>
</tbody>
</table>

The `objdump` utility can be used to disassemble the machine code generated by the `gcc` compiler for an Intel x86 processor. Disassembling means the translation of binary machine code into a mnemonic representation humans can easier read. Here is a description of the most important disassembled machine instructions:

- `4004ac` push old base pointer to the stack
- `4004ad` set base pointer to the current stack address
- `4004b0` move the constant 5 to the stack (offset -4)
- `4004b7` move the constant 2 to the stack (offset -8)
- `4004be` move the second constant into register eax
- `4004c1` move the first constant into register edx
- `4004c4` add register edx to register eax
- `4004c6` move the register eax to the stack (offset -12 = 0xc)
- `4004c9` move the value from the stack into register eax
- `4004cc` pop the old base pointer off the stack
- `4004cd` return from function call

The Intel x86 assembly language uses the following notation:

- `%eax`, `%edx`, `%rbp`, `%rsp`, ... refer to the processor registers eax (general accumulator), edx (general register), rbp (base register), rsp (stack register), ...
- `$0x5`, `$0x2`, ... denotes hexadecimal constants
- `-0x4(%rbp)` refers to an address with a negative 4 byte offset from the address stored in the register rbp
Disassembled machine code (with optimizations)

# compile with optimization (gcc -O2) and look at the machine code
# gcc (Debian 4.7.2-5) 4.7.2 on Linux

00000000004003a0 <main>:
  4003a0: b8 07 00 00 00 mov $0x7,%eax
  4003a5: c3 retq
  4003a6: 90 nop
  4003a7: 90 nop

With optimization turned on, the compiler has detected that the expression only depends on constants and thus the expression can already be evaluated at compile time to improve speed at runtime. Furthermore, the compiler has figured out that the variables a, b, and c are not used outside of the function main() and hence it is not necessary to store any values for them somewhere in memory.

As a result, the function now translates into two instructions: one to load the constant 7 into register eax and one to return from the function. The previous version used 11 instructions.

Note: A compiler can be smart and produce machine code that is optimized for a certain processor architecture. In particular, a smart compiler can rewrite parts of the program logic as long as the execution leads to the same runtime behavior.

Note: Most of the time, compilers optimize for speed but it is also possible to ask the compiler to optimize for space.
• The lexical analysis turns a sequence of characters into a sequence of tokens.
• The format of tokens is often specified using regular expressions.
• The syntax analysis turns the sequence of tokens into a parse tree.
• The possible structure of the parse tree is commonly defined using a formal grammar (a set of rules defining the structure of syntactically correct programs).
• The semantic analysis adds semantic information to the parse tree, resulting in an abstract syntax tree.
• Transformations on the abstract syntax tree can be used to optimize the program.
• The code generation for a specific target CPU is the final step.
• Multi-language compiler support multiple language frontends that produce a common abstract syntax tree.
• Multi-target compiler multiple target platforms by driving multiple target platform specific code generators.

Further information:
Euclidean algorithm to find the greatest common divisor of $a$ and $b$:

```python
while (b != 0):
    if (a > b):
        a = a - b
    else:
        b = b - a
return a
```
The syntax of programming languages is often defined using syntax rules. A common notation for syntax rules is the Backus-Naur-Form (BNF):

- Terminal symbols are enclosed in quotes
- Non-terminal symbols are enclosed in <>
- A BNF rule consists of a non-terminal symbol followed by the defined-as operator ::=
- A rule expression consists of terminal and non-terminal symbols and operators; the empty operator denotes concatenation and the | operator denotes an alternative
- Parenthesis may be used to group elements of a rule expression

A set of BNF rules has a non-terminal starting symbol.

Example: Let \( \Sigma = \{0, 1, \ldots, 9, x, y, z\} \).

1. \(<\text{expression}> ::= <\text{term}> | <\text{expression}> + <\text{term}>\)
2. \(<\text{term}> ::= <\text{factor}> | <\text{term}> * <\text{factor}>\)
3. \(<\text{factor}> ::= <\text{constant}> | <\text{variable}> | "(" <\text{expression}> ")"\)
4. \(<\text{variable}> ::= "x" | "y" | "z"\)
5. \(<\text{constant}> ::= <\text{digit}> | <\text{digit}> <\text{constant}>\)
6. \(<\text{digit}> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"\)

Using \(<\text{expression}>\) as a start symbol, the grammar defines a simplified format of expressions. Here is a sample step-by-step deviation:

\[
\begin{align*}
<\text{expression}> &\rightarrow <\text{expression}> + <\text{term}> \\
&\rightarrow <\text{term}> + <\text{term}> \\
&\rightarrow <\text{factor}> + <\text{term}> \\
&\rightarrow <\text{factor}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{factor}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{constant}> + <\text{term}> * <\text{factor}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow <\text{digit}> + <\text{digit}> * <\text{variable}> \\
&\rightarrow 42 + 8 * x
\end{align*}
\]

While a grammar can be used to derive a word of the language from a start symbol, it can also be used to reduce a given input to the start symbol if the input is a word of the language. This is used by compilers to test whether a given program text is a (syntactically) valid word of the language.

The BNF of the Haskell language can be found in Section 10.5 of the Haskell 2010 report.
Interpreters often implement read–eval–print loops (REPLs). REPLs facilitate exploratory programming because the programmer can inspect the result before deciding which expression to provide for the next read. The read–eval–print loop is more interactive than the edit-compile-run-debug cycle of compiled languages.

However, highly interactive REPLs can also trick programmers to spend a lot of time on trial-and-error coding efforts in situations where thinking about the problem and the algorithms to solve a problem would have been overall more time effective.
Compiler and Interpreter

[1] Source Code --> Interpreter

- An interpreter is a computer program that directly executes source code written in a higher-level programming language.
- A compiler is a program that transforms source code written in a higher-level programming language (the source language) into a lower-level computer language (the target language).

- Many modern high-level programming languages use both compilation and interpretation
  - Source code is first compiled (either using an explicit compilation step or on-the-fly) into an intermediate byte code.
  - The byte code is afterwards interpreted by an byte-code interpreter.
  - As an optimization, the byte code might be further compiled to machine code (on-the-fly compilation).
- Byte-code is often generated for Stack Machines that differ from Register Machines by not having registers and performing all operations on a stack.
- An interpreter is usually written in a higher-level programming language and compiled into machine code.

Further information:
Virtual Machines and Emulators

• A virtual machine (VM) is an emulation of a particular computer system. Virtual machines operate based on the computer architecture and functions of a real computer.

• An emulator is hardware or software or both that duplicates (or emulates) the functions of one computer system (the guest) in another computer system (the host), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest).

⇒ Virtual machines were invented in the 1970s and reinvented in the 1990s.

⇒ Virtual machines have been an enabler for cloud computing since they are easy to start / stop / clone / migrate and they separate the software implementing services form the underlying hardware.

• The software virtualizing the underlying hardware is called a hypervisor.

• Full virtualization: Virtual machines run a complete operating system inside of the virtual machine. The hypervisor virtualizes all aspects of the underlying hardware (e.g., VmWare).

• Operating-system level virtualization: Virtual machines (often called virtual servers) all run on a single underlying operating system instance; virtualization is achieved by partitioning operating system services (e.g., Linux Container).

• Paravirtualization: Virtual machines different operating systems that have been adapted to redirect certain services to a special designated operating system instance (e.g., Xen).

Further information:

• http://en.wikipedia.org/wiki/Emulator

• http://en.wikipedia.org/wiki/Virtual_machine
Hardware vs. System Software vs. Application Software

Remarks:

- The operating system provides services to application programs that can be used by making system calls.
- The operating system controls all hardware components and mediates between the hardware and application programs.
- Application programs are written against the system call interface (and associated libraries).

Relevant Linux tools to understand the difference between library and system calls:

- The `strace` tool can trace the system calls made by a program at runtime.
- The `ltrace` tool can trace the library calls made by a program at runtime.
Section 26: Operating Systems

Interpreter and Compiler

Operating Systems
Operating System Kernel Functions

- Execute many programs concurrently (instead of just one program at a time)
- Assign resources to running programs (memory, CPU time, ...)
- Ensure a proper separation of concurrent processes
- Enforce resource limits and provide means to control processes
- Provide logical filesystems on top of block-oriented raw storage devices
- Control and coordinate input/output devices (keyboard, display, ...)
- Provide basic network communication services to applications
- Provide input/output abstractions that hide device specifics
- Enforce access control rules and privilege separation
- Provide a well defined application programming interface (API)

Hello world in x64 assembly language using a system library function:

```assembly
; -----------------------------------------------
; Writes "Hello, world" to the console using the C library (try on Linux).
; gcc -static -o hello-asm-libc hello-asm-libc.s
; -----------------------------------------------

.global main
.text
main:
    mov $message, %rdi          # First integer (or pointer) parameter in %rdi
    call puts                  # puts(message)
    ret                        # Return to C library code

.data
message:
    .asciz "Hello, world"      # asciz puts a 0 byte at the end
```

Hello world in x64 assembly language using system calls:

```assembly
; -----------------------------------------------
; Writes "Hello, world" to the console using system calls (try on Linux)
; gcc -nostdlib -static -o hello-asm-syscall hello-asm-syscall.s
; -----------------------------------------------

.global _start
.text
_start:
    mov $1, %rax                # Write system call number is 1
    mov $1, %rdi                # File descriptor 1 is stdout
    mov $message, %rsi          # Address of the string to output
    mov $13, %rdx               # Number of bytes in message
    syscall                     # Invoke write(1, message, 13) system call
    mov $60, %rax               # Exit system call number is 60
    xor %rdi, %rdi               # We want return code 0
    syscall                     # Invoke exit(0) system call
.data
message:
    .ascii "Hello, world\n"
```
Definition (process)
An instance of a computer program that is being executed is called a process.

- The OS kernel maintains information about each running process and assigns resources and ensures protection of concurrently running processes.
- In Unix-like Operating Systems
  - a new process is created by “cloning” (forking) an already existing process
  - a process may load a new program image (machine code) to execute
  - a terminating process returns a number to its parent process
  - a parent process can wait for child processes to terminate

⇒ A very basic command interpreter can be written in a few lines of Python code.

Relevant Linux command line tools to inspect the processes running on a system:
- The tool `ps` shows a list of processes.
- The tool `pstree` shows the process tree.
- The tool `top` periodically shows the list of processes (and threads) sorted by some sorting criteria.
The figure illustrates what happens if you type `date` into a command interpreter like the `bash` shell. The shell uses a `read()` system call to read the input. It then invokes the `fork()` system call to create a clone of itself. It then waits for the clone (child process) to finish by invoking the `wait()` system call. The child process uses the `exec()` system call to replace the current process image with the process image of the `date` program. The child process finally exits by calling `exit()`. The process stays around until the exit code has been delivered to the parent process.

Description of the system calls:

- The `fork()` system call creates a new child process which is an exact copy of the parent process, except that the result of the system call differs: 0 is returned to the new process, the process number of the new process is returned to the parent process.
- The `exec()` system call replaces the current process image with a new process image.
- The `wait()` system call waits for a child process to exit.
- The `exit()` system call terminates the calling process. (Returning from `main()` eventually leads to a call of `exit()`.)
OS Abstraction #1: Processes and Process Lifecycle

while (1) {
    show_prompt(); /* display prompt */
    read_command(); /* read and parse command */
    pid = fork(); /* create new process */
    if (pid < 0) { /* continue if fork() failed */
        perror("fork");
        continue;
    }
    if (pid != 0) { /* parent process */
        waitpid(pid, &status, 0); /* wait for child to terminate */
    } else { /* child process */
        execvp(args[0], args, 0); /* execute command */
        perror("execvp"); /* only reach on exec failure */
        _exit(1); /* exit without any cleanups */
    }
}

A very basic shell in haskell.

{ - |
 module: hsh/hsh.hs

module Main where

import System.IO
import System.Exit (exitSuccess)
import Control.Monad (forever)

exec :: [String] -> IO ()
exec [] = return ()
exec (cmd:args) = executeFile cmd True args Nothing

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering -- make stdout unbuffered
    forever $ do
        putStrLn "hsh > "
        done <- isEOF
        if done then do
            putStrLn ""
            exitSuccess
        else do
            line <- getLine
            pid <- forkProcess $ exec $ words line
            getProcessStatus True False pid
OS Abstraction #2: File Systems

- Files are persistent containers for the storage of data
- Unstructured files contain a sequence of bytes
- Applications interpret the content of a file in a specific way
- Files also have meta data (owner, permissions, timestamps)
- Hierarchical file systems use directories to organize files into a hierarchy
- Names of files and directories at one level of the hierarchy usually have to be unique
- The operating system maps the logical structure of a hierarchical file system to a block-oriented storage device
- The operating system must ensure file system integrity
- The operating system may support compression and encryption of file systems

- It is difficult to design a file system that satisfies to some extend conflicting requirements:
  - It should be fast and at the same time it should maintain data integrity.
  - It should not show any aging effects.
  - It should work with small and large block storage devices.
  - It should work with slow and fast block storage devices.
  - It should work well for small and large files.
  - It should support file names with international character sets.
  - ...
- General purpose file systems try to find a balance but in general there is not a single 'best' file system.

Further information:
OS Abstraction #2: File Systems (Unix)

- The logical structure of a typical Unix file system
- The . in a directory always refers to the directory itself
- The .. in a directory always refers to the parent directory, except in the root directory
- A link is a reference of a file system object from a directory
- Any file system changes need to maintain the integrity of these links

- Some file system operations require updates of multiple data blocks.
- As a consequence, file systems can be temporarily inconsistent.
- If a file system became inconsistent, it needs to be repaired by special programs. Sometimes this leads to data loss.
- Warning: Removing a file often means only that the name referring to the data blocks is removed (the link to the file is removed, the file is unlinked).
- Classic storage devices store data on a (rotating) magnetic surface. The magnetic surface ‘remembers’ data even if it was overwritten. Solid state storage devices do not show this behavior.
## OS Abstraction #2: File and Directory Operations (Unix)

### File operations
- **open()**: open a file
- **read()**: read data from the current file position
- **write()**: write data at the current file position
- **seek()**: seek to a file position
- **stat()**: read meta data
- **close()**: close an open file
- **unlink()**: remove a link to a file

### Directory operations
- **mkdir()**: create a directory
- **rmdir()**: delete a directory
- **chdir()**: change to a directory
- **opendir()**: open a directory
- **readdir()**: read a directory entry
- **closedir()**: close a directory

```python
#!/usr/bin/env python3.4
import os
import pwd
import grp
import time
import sys

def ls(path):
    if os.path.isdir(path):
        for entry in os.listdir(path):
            ls(os.path.join(path, entry))
    else:
        s = os.stat(path)
        print("%-8s %-8s %6d %s %s"
              % (pwd.getpwuid(s.st_uid)[0],
                  grp.getgrgid(s.st_gid)[0],
                  s.st_size,
                  time.strftime("%Y-%m-%dT%H:%M:%S", time.localtime(s.st_mtime)),
                  path))
    for arg in sys.argv[1:]:
        print(arg)
    ls(arg)
```

---

Jürgen Schönwälder (Jacobs University Bremen)  Introduction to Computer Science  December 5, 2019  218 / 256
OS Abstraction #3: Inter-process Communication

- Communication between processes:
  - Signals (software interrupts)
  - Pipes (local unidirectional byte streams)
  - Sockets (local and global bidirectional byte or datagram streams)
  - Shared memory (memory regions shared between multiple processes)
  - Message queues (a queue of messages between multiple processes)
  - Sockets are the basic inter-process communication abstraction used for communication between processes over the Internet

```python
#!/usr/bin/env python3.4
import socket
import sys

def connect(server):
    ai_list = socket.getaddrinfo(server, "http")
    err = None
    s = None
    for (family, socktype, proto, canonname, sockaddr) in ai_list:
        try:
            s = socket.socket(family, socktype, proto)
            s.connect(sockaddr)
        except Exception as e:
            err = e
            if s:
                s.close()
            else:
                break
        else:
            raise err
    return s

def wget(server):
    s = connect(server)
    if s:
        with s.makefile(mode='w') as f:
            f.write("GET / HTTP/1.0\r\nHost: %s\r\n\r\n" % (server))
        with s.makefile(mode='r') as f:
            for l in f.readlines():
                print(l.strip())
            for arg in sys.argv[1:]:
                wget(arg)
```

Further information:

- [http://en.wikipedia.org/wiki/Inter-process_communication](http://en.wikipedia.org/wiki/Inter-process_communication)
Part VII

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 excellent “Background reading on Hoare Logic”.

213
Section 27: Software Specification

27 Software Specification

28 Software Verification

29 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.
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”:

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

- 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 [1]. 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 conditions on program variables. They will be 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:

- $\{X = 1\} \ Y := 2 \ {X = 1 \land Y = 2}$
- $\{X = 1\} \ Y := 2 \times X \ {X = 1 \land Y = 2}$
- $\{X = 1\} \ Y := X + X \ {X = 1 \land Y = 2}$
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 always a result is 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.

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 $\{P\} \text{C} \{Q\}$ usually provides a partial correctness specification. We may use the notation $\lbrack P \rbrack \text{C} \lbrack Q \rbrack$ for a total correctness specification.
Hoare Notation Conventions

1. The symbols $V, V_1, \ldots, V_n$ stand for arbitrary variables. Examples of particular variables are $X, Y, R$ etc.

2. The symbols $E, E_1, \ldots, E_n$ stand for arbitrary expressions (or terms). These are expressions like $X + 1, \sqrt{2}$ etc., which denote values (usually numbers).

3. The symbols $S, S_1, \ldots, S_n$ stand for arbitrary statements. These are conditions like $X < Y, X^2 = 1$ etc., which are either true or false.

4. The symbols $C, C_1, \ldots, C_n$ stand for arbitrary commands of our programming language; these commands are described in 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 term \( E \) to the variable \( V \). All variables are assumed to have global scope.
• Example: \( X := X + 1 \)
Hoare Skip Command

- Syntax: \textit{SKIP}
- Semantics: Do nothing. The state after execution the \textit{SKIP} command is the same as the state before executing the \textit{SKIP} command.
- Example: \textit{SKIP}

The \textit{SKIP} command does nothing. It is still useful since it allows us to construct a single conditional command.
Hoare Command Sequences

- Syntax: $C_1; \ldots; C_n$
- Semantics: The commands $C_1, \ldots, C_n$ 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 \textit{SKIP} to create conditional statements without a \textit{THEN} or \textit{ELSE} branch:

\[
\begin{align*}
\text{IF } S \text{ THEN } C \text{ ELSE SKIP FI} \\
\text{IF } S \text{ THEN SKIP ELSE } C \text{ FI}
\end{align*}
\]
Hoare While Loop

• Syntax: \texttt{WHILE }S\texttt{ DO }C\texttt{ 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: \texttt{WHILE }\neg(X = 0)\texttt{ DO } X := X - 2\texttt{ 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 (\texttt{if} / \texttt{fi}, \texttt{case} / \texttt{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

1: function collatz(X)
2:     while X > 1 do
3:         if (X%2) \neq 0 then
4:             X \leftarrow (3 \cdot X) + 1
5:         else
6:             X \leftarrow X/2
7:         end if
8:     end while
9:     return X
10: 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.

For further information:

Specification can be Tricky

- Specification for the maximum of two variables:
  \[ \{ T \} \ C \{ Y = \max(X, Y) \} \]

- \( C \) could be:
  \[
  \text{IF } X > Y \text{ THEN } Y := X \text{ ELSE SKIP FI}
  \]
- But \( C \) could also be:
  \[
  \text{IF } X > Y \text{ THEN } X := Y \text{ ELSE SKIP FI}
  \]
- 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:

- \( \{ X = 1 \} \ Y := 2 \{ X = 1 \land Y = 2 \} \)
- \( \{ X = 1 \} \ Y := X + 1 \{ X = 1 \land Y = 2 \} \)
- \( \{ X = 1 \} \ Y := 2 \times X \{ X = 1 \land Y = 2 \} \)

A slightly more complex example (factorial):

<table>
<thead>
<tr>
<th>Precondition: { ( X &gt; 0 \land X = x ) }</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( F := 1 )</td>
</tr>
<tr>
<td>2: \textbf{while} ( X &gt; 0 ) \textbf{do}</td>
</tr>
<tr>
<td>3: ( F := F \times X )</td>
</tr>
<tr>
<td>4: ( X := X - 1 )</td>
</tr>
<tr>
<td>5: \textbf{od}</td>
</tr>
<tr>
<td>Postcondition: { ( F = x! ) }</td>
</tr>
</tbody>
</table>
Section 28: Software Verification

Software Specification

Software Verification

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

$$
\vdash \{X + 1 = n + 1\} X := X + 1 \{X = n + 1\}
$$


The precondition $P$ is stronger than $P'$ ($P \rightarrow P'$) if the set of states $\{s \mid s \vdash P\} \subseteq \{s \mid s \vdash P'\}$.

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:

$$\vdash \{P\} C \{Q'\}, \quad \vdash Q' \rightarrow Q \quad \vdash \{P\} C \{Q\}$$

• 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 $\{s \mid s \vdash Q'\} \subseteq \{s \mid s \vdash Q\}$. 
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 precondition 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 (stronges 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).
Assignment Axiom

- Let $P[E/V]$ ($P$ with $E$ for $V$) denote the result of substituting the term $E$ for all occurrences 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\} \sqsubseteq \{Q_1\} and \{P_2\} \sqsubseteq \{Q_2\}, then \{P_1 \land P_2\} \sqsubseteq \{Q_1 \land Q_2\}

\[ \vdash \{P_1\} \sqsubseteq \{Q_1\}, \quad \vdash \{P_2\} \sqsubseteq \{Q_2\} \]
\[ \vdash \{P_1 \land P_2\} \sqsubseteq \{Q_1 \land Q_2\} \]

• We get a similar rule for disjunctions:

\[ \vdash \{P_1\} \sqsubseteq \{Q_1\}, \quad \vdash \{P_2\} \sqsubseteq \{Q_2\} \]
\[ \vdash \{P_1 \lor P_2\} \sqsubseteq \{Q_1 \lor Q_2\} \]

• These rules allow us to prove \( \vdash \{P\} \sqsubseteq \{Q_1 \land Q_2\} \) by proving both \( \vdash \{P\} \sqsubseteq \{Q_1\} \) and \( \vdash \{P\} \sqsubseteq \{Q_2\} \).
Skip Command Rule

- Syntax: $SKIP$
- Semantics: Do nothing. The state after execution the command $SKIP$ is the same as the state before executing the command $SKIP$.
- Skip Command Rule:

$$\vdash \{P\} \; SKIP \; \{P\}$$
Sequence Rule

• Syntax: $C_1; \ldots; C_n$
• Semantics: The commands $C_1, \ldots, C_n$ are executed in that order.
• Sequence Rule:

\[
\vdash \{ P \} C_1 \{ R \}, \quad \vdash \{ R \} C_2 \{ Q \}
\]

\[\vdash \{ P \} C_1; C_2 \{ Q \}\]

The sequence rule can be easily generalized to $n > 2$ commands:

\[
\vdash \{ P \} C_1 \{ R_1 \}, \quad \vdash \{ R_1 \} C_2 \{ R_2 \}, \ldots, \quad \vdash \{ R_{n-1} \} C_n \{ Q \}
\]

\[\vdash \{ P \} C_1; C_2; \ldots; C_n \{ Q \}\]

Example (swapping two numbers):

**Precondition:** $\{ X = x \land Y = y \}$

1. $R := X$
2. $X := Y$
3. $Y := R$

**Postcondition:** $\{ X = y \land Y = x \}$

The proof of the correctness of the sequence of assignments is broken down into the following steps:

(i) $\vdash \{ X = x \land Y = y \} R := X \{ R = x \land Y = y \}$ (assignment axiom)
(ii) $\vdash \{ R = x \land Y = x \} X := Y \{ R = x \land X = y \}$ (assignment axiom)
(iii) $\vdash \{ R = x \land X = y \} Y := R \{ Y = x \land X = y \}$ (assignment axiom)
(iv) $\vdash \{ X = x \land Y = y \} R := X \{ R = x \land X = y \}$ (sequence rule for (i) and (ii))
(v) $\vdash \{ X = x \land Y = y \} R := X \{ R = x \land X = y \}$ (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:

\[
\frac{\vdash \{ P \land S \} \ C_1 \ \{ Q \} \quad \vdash \{ P \land \neg S \} \ C_2 \ \{ Q \}}{\vdash \{ P \} \ IF \ S \ THEN \ C_1 \ ELSE \ C_2 \ FI \ \{ Q \}}
\]

Consider the following specification and program (max):

**Precondition:** \( \{ X = x \land Y = y \} \)

1: if \( X \geq Y \) then
2: \( M := X \)
3: else
4: \( M := Y \)
5: fi

**Postcondition:** \( \{ M = \text{max}(x, y) \} \)

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:

\[
\begin{align*}
\{ X = x \land Y = y \land X \geq Y \} & \quad M := X \ \\ \{ X = x \land Y = y \land X < Y \} & \quad M := Y
\end{align*}
\]

The definition of \( \text{max}(x, y) \) we are going to use is the following:

\[
\text{max}(x, y) = \begin{cases} 
  x & x \geq y \\
  y & x < y
\end{cases}
\]

This gives us the following implications:

\[
\begin{align*}
M &= x \land X = x \land Y = y \land X \geq Y \rightarrow M = \text{max}(x, y) \\
M &= y \land X = x \land Y = y \land X < Y \rightarrow M = \text{max}(x, y)
\end{align*}
\]

Postcondition weakening gives us:

\[
\begin{align*}
\{ X = x \land Y = y \land X \geq Y \} & \quad M := X \ \{ M = \text{max}(x, y) \} \\
\{ X = x \land Y = y \land X < Y \} & \quad M := Y \ \{ M = \text{max}(x, y) \}
\end{align*}
\]

Applying the conditional rule, we get:

\[
\{ X = x \land Y = y \} \ IF \ X \geq Y \ THEN \ M := X \ ELSE \ M := Y \ FI \ \{ M = \text{max}(x, y) \}
\]
While Command Rule

- Syntax: \texttt{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:

\[
\frac{\Gamma \vdash P \land S \quad \Gamma \vdash C}{\Gamma \vdash P \quad \text{WHILE S DO C OD}}\]

\( P \) is an invariant of \( C \) whenever \( S \) holds. Since executing \( C \) preserves the truth of \( P \), executing \( C \) any number 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):

\begin{align*}
\text{Precondition: } & \{ Y = 1 \land Z = 0 \land X = x \land X \geq 0 \} \\
1: & \textbf{while } Z \neq X \textbf{ do} \\
2: & \quad Z := Z + 1 \\
3: & \quad Y := Y \times Z \\
4: & \textbf{od} \\
\text{Postcondition: } & \{ Y = x! \}
\end{align*}

We need to find an invariant \( P \) such that:
- \( \{ P \land Z \neq X \} Z := Z + 1; \ Y := Y \times Z \ {P} \) \hspace{1cm} \text{(while rule)}
- \( Y = 1 \land Z = 0 \rightarrow P \) \hspace{1cm} \text{(precondition strengthening)}
- \( P \land \neg(Z \neq X) \rightarrow Y = X! \) \hspace{1cm} \text{(postcondition weakening)}

The invariant \( Y = Z! \) serves the purpose:
- \( Y = Z! \land Z \neq X \rightarrow Y \cdot (Z + 1) = (Z + 1)! \) \hspace{1cm} \text{(assignment axiom)}
- \( \{ Y \cdot (Z + 1) = (Z + 1)! \} Z := Z + 1 \{ Y \cdot Z = Z! \} \)
- \( \{ Y \cdot Z = Z! \} Y := Y \times Z \{ Y = Z! \} \)
- \( \{ Y = Z! \} Z := Z + 1; \ Y := Y \times Z \{ Y = Z! \} \)
- \( Y = 1 \land Z = 0 \rightarrow Y = Z! \) since \( 0! = 1 \)
- \( Y = Z! \land \neg(Z \neq X) \rightarrow Y = X! \) since \( \neg(Z \neq X) \) is equivalent to \( Z = X \)
Section 29: Automation of Software Verification

Software Specification

Software Verification

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:
  1. Let the human expert provide annotations of the specification (e.g., loop invariants) that help with the generation of proof obligations
  2. Generate proof obligations automatically (verification conditions)
  3. Use automated theorem provers to verify some of the proof obligations
  4. 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:

<table>
<thead>
<tr>
<th>Precondition: { \top }</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( R := X )</td>
</tr>
<tr>
<td>2: ( Q := 0 )</td>
</tr>
<tr>
<td>3: while ( Y \leq R ) do</td>
</tr>
<tr>
<td>4: ( R := R - Y )</td>
</tr>
<tr>
<td>5: ( Q := Q + 1 )</td>
</tr>
<tr>
<td>6: od</td>
</tr>
</tbody>
</table>

Postcondition: \( \{ X = Y \cdot Q + R \land R < Y \} \)
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:** $\{ \top \}$
1: $R := X$
2: $Q := 0$
3: $\{ R = X \land Q = 0 \}$
4: while $Y \leq R$ do
5: $\{ X = Y \cdot Q + R \}$
6: $R := R - Y$
7: $Q := Q + 1$
8: 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. $(X = Y \cdot Q + R \land Y \leq R) \rightarrow (X = Y \cdot (Q + 1) + (R - Y)$
Starting with the annotated example:

**Precondition:** \{⊤\}
1: \( R := X \)
2: \( Q := 0 \)
3: \( \{R = X \land Q = 0\} \)
4: while \( Y \leq R \) do
5: \( \{X = Y \cdot Q + R\} \)
6: \( R := R - Y \)
7: \( Q := Q + 1 \)
8: 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 \(⊤\rightarrow (X = X \land 0 = 0)\) as follows:

\[
\{⊤\} R := X; Q := 0 \{R = X \land Q = 0\}
\{⊤\} R := X \{R = X \land 0 = 0\}
⊤\rightarrow (X = X \land 0 = 0)
\]

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:

\[
\{X = Y \cdot Q + R \land Y \leq R\} R := R - Y; Q := Q + 1 \{X = Y \cdot Q + R\}
\{X = Y \cdot Q + R \land Y \leq R\} R := R - Y; \{X = Y \cdot (Q + 1) + R\}
\{X = Y \cdot Q + R \land Y \leq R\} \rightarrow (X = Y \cdot (Q + 1) + (R - Y))
\]
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:

\[
\vdash \{P\} \ C \ \{Q\} \\
\vdash [P] \ C \ [Q]
\]
Rules for Total Correctness [1/4]

- Assignment axiom
  \[ \vdash [P[E/V]] V := E \ [P] \]

- Precondition strengthening
  \[ \vdash P \rightarrow P', \vdash [P'] C [Q] \]

  \[ \vdash [P] C [Q] \]

- Postcondition weakening
  \[ \vdash [P] C [Q'], \vdash Q' \rightarrow Q \]

  \[ \vdash [P] C [Q] \]
Rules for Total Correctness [2/4]

- Specification conjunction

\[ \vdash [P_1] C [Q_1], \vdash [P_2] C [Q_2] \]
\[ \vdash [P_1 \land P_2] C [Q_1 \land Q_2] \]

- Specification disjunction

\[ \vdash [P_1] C [Q_1], \vdash [P_2] C [Q_2] \]
\[ \vdash [P_1 \lor P_2] C [Q_1 \lor Q_2] \]

- Skip command rule

\[ [P] \text{SKIP} [P] \]
Rules for Total Correctness [3/4]

- **Sequence rule**
  \[ \vdash [P] C_1 [R_1], \vdash [R_1] C_2 [R_2], \ldots, \vdash [R_{n-1}] C_n [Q] \]
  \[ \vdash [P] C_1; C_2; \ldots; C_n [Q] \]

- **Conditional rule**
  \[ \vdash [P \land S] C_1 [Q], \vdash [P \land \lnot S] C_2 [Q] \]
  \[ \vdash [P] IF S THEN C_1 ELSE C_2 FI [Q] \]
Rules for Total Correctness [4/4]

- While rule
\[
\frac{\vdash [P \land S \land E = n] \quad C \quad [P \land (E < n)], \quad \vdash P \land S \rightarrow E \geq 0}{\vdash [P] \text{ WHILE } S \text{ DO } C \text{ OD } [P \land \neg S]}
\]

- 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: $\{\top\}$

1: $R := X$
2: $Q := 0$
3: while $Y \leq R$ do
4: $R := R - Y$
5: $Q := Q + 1$
6: 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:

$P \land S \rightarrow E \geq 0$
$Y > 0 \land Y \leq R \rightarrow R > 0$
The rules for the generation of termination verification conditions follow directly from the rules for the generation of partial correctness verification 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

\[
P \rightarrow R \\
R \land \neg S \rightarrow Q \\
R \land S \rightarrow E \geq 0
\]

and add verification 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: \(\{\top\}\)

1: \(R := X\)
2: \(Q := 0\)
3: \(\{R = X \land Q = 0\}\)
4: while \(Y \leq R\) do
5: \(\{X = Y \cdot Q + R\}\)
6: \([R]\)
7: \(R := R - Y\)
8: \(Q := Q + 1\)
9: od

Postcondition: \(\{X = Y \cdot Q + R \land R < Y\}\)

The while loop rule gives use the following termination VCs

\[
(R = X \land Q = 0) \rightarrow (X = Y \cdot Q + R) \\
(X = Y \cdot Q + R \land \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \land R < Y) \\
(X = Y \cdot Q + R \land (Y \leq R)) \rightarrow R \geq 0
\]

and the VC generated as follows:

\[
\{X = Y \cdot Q + R \land Y \leq R \land R = n\} R := R - Y; \ Q := Q + 1 \{X = Y \cdot Q + R \land R < n\} \\
\{X = Y \cdot Q + R \land Y \leq R \land R = n\} R := R - Y; \ {X = Y \cdot (Q + 1) + R \land R < n\} \\
(X = Y \cdot Q + R \land Y \leq R \land R = n) \rightarrow (X = Y \cdot (Q + 1) + (R - Y) \land (R - Y) < n)
\]

The last VC is not true in general and hence the algorithm does not always terminate:

\(Y = 0\):

\((((X = R \land 0 \leq R \land R = n) \rightarrow (X = R \land R < n)) \rightarrow \bot\)

\(Y < 0\):

\(\((X = Y \cdot Q + R \land Y \leq R \land R = n) \rightarrow (X = Y \cdot (Q + 1) + (R - Y) \land (R - Y) < n)) \rightarrow \bot\)
Termination and Correctness

- Partial correctness and termination implies total correctness:

\[ \vdash \{ P \} C \{ Q \}, \quad \vdash [P] C [T] \]
\[ \vdash [P] C [Q] \]

- Total correctness implies partial correctness and termination:

\[ \vdash [P] C [Q] \]
\[ \vdash \{ P \} C \{ Q \}, \quad \vdash [P] C [T] \]