Introduction to Computer Science

Jürgen Schönwälder

November 13, 2018

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.

Contents

Part I

Introduction and Examples

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

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:

- https://en.wikipedia.org/wiki/Computer_science
- Defining Computer Science (ACM SIGCSE Bulletin, Vol. 32, Issue 2, June 2000)

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

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?

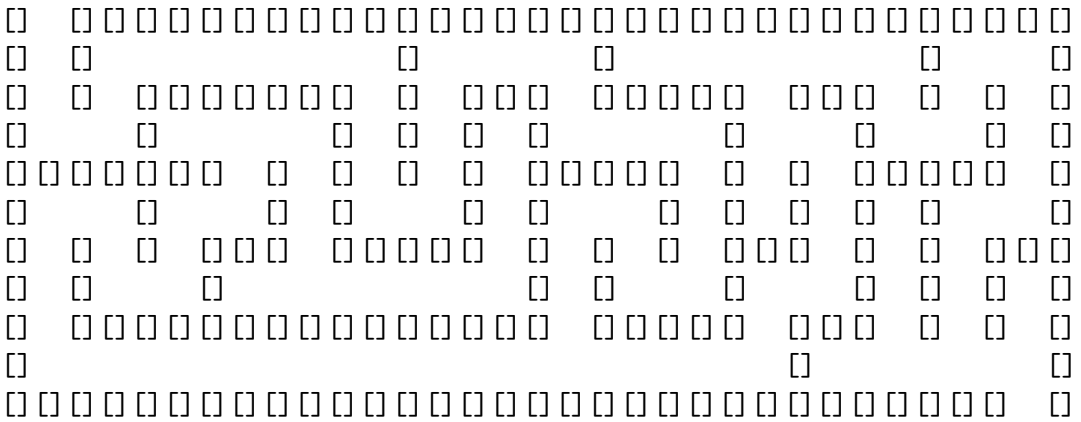
While thinking about a problem, it is generally useful to go through a number of examples. The examples should cover regular cases and corner cases. It is useful to write the examples down since they may serve later as test cases for an implementation of an algorithm that has been selected to solve the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

For further information:

- https://doi.org/10.1007/11915355_15

Section 2: Maze Generation Algorithms

Maze (33 x 11)



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

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

For further information:

- https://en.wikipedia.org/wiki/Maze_solving_algorithm

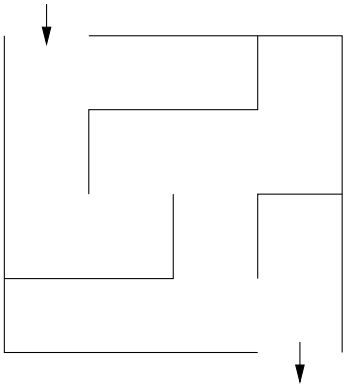
Problem Statement

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.

Hacking. . .

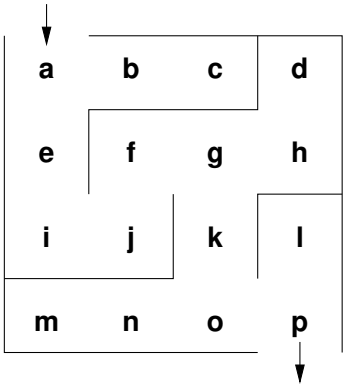
While hacking can be fun, it is often far more effective to think first before opening the editor and starting to write code. It often also helps to discuss the problem with others. Even coding together in pairs of two (pair programming) has been found to lead to better programs. Despite common believes,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

For further information:

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

Problem Formalization (1/3)

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



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

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

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

Problem Formalization (2/3)

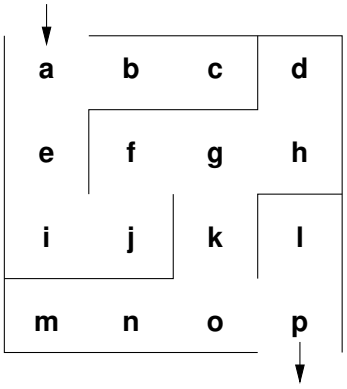
Lets try to formalize the problem in mathematical terms:

- 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:

- https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

Why use a mathematical formalization?

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

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

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

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

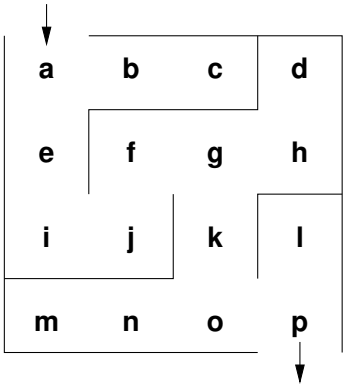
Problem Formalization (3/3)

Definition:

- A maze 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.

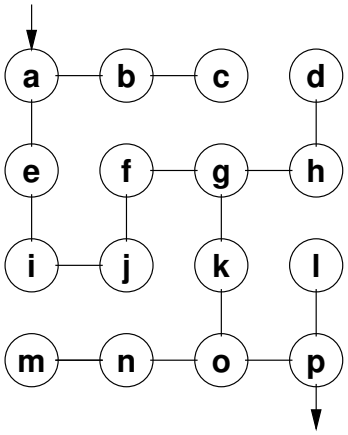
Given this formalization, the example maze is represented as follows:

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

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):

```
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
- http://www.graphviz.org/

Mazes as Graphs (Good Mazes)

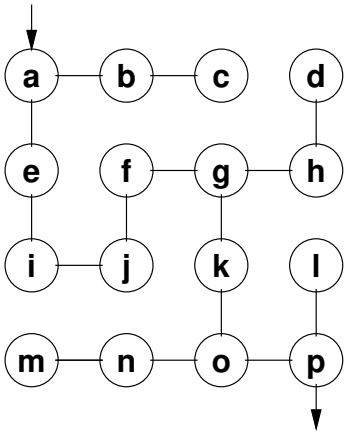
Recall, what is a good maze?

- 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:

- https://en.wikipedia.org/wiki/Spanning_tree

Kruskal's Algorithm (1/2)

General approach:

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

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:

- https://en.wikipedia.org/wiki/Equivalence_class
- https://en.wikipedia.org/wiki/Disjoint-set_data_structure

```haskell
{- |
   Module: maze/maze.hs

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

module Maze ( Node, Edge, Graph, maze ) 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)
maze :: Graph -> Graph
maze g = maze' g ( map (:[]) (fst g) ) (fst g, [])

-- take a graph, a list of node equivalence classes, a new (tree)
-- graph (which we are building up) and return a spanning tree graph
maze' :: Graph -> [Class] -> Graph -> Graph
maze' g cs n | length cs == 1 = n
             | otherwise      = maze' g (merge cs e) (fst n, e : snd n)
  where e = select cs (snd g)

-- test whether an edge connects two equivalence classes
distinct :: [Class] -> Edge -> Bool
distinct cs e = filter (elem (fst e)) cs /= filter (elem (snd e)) cs

-- find an edge that connects two equivalence classes
select :: [Class] -> [Edge] -> Edge
select cs es = head (filter (distinct cs) es)

-- test whether an edge matches a node class
match :: Edge -> Class -> Bool
match e c = elem (fst e) c || elem (snd e) c

-- merge node equivalence classes given a specific edge
merge :: [Class] -> Edge -> [Class]
merge cs e = concat a : b
  where (a,b) = partition (match e) cs
```

Listing 1: Maze calculation in Haskell (lacking randomization)

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

Section 3: String Search Algorithms

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

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

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

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

Note that the problem formalization details that we are searching for the first occurance of p in t. We could have defined the problem differently, e.g., searching for the last occurance of p in t, or searching for all occurances 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 = FINDANEEDLEINAHAYSTACK, p = NEEDLE

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

N e e d l e
  N e e d l e
    N E e d l e
      N e e d l e
        N e e d l e
          N E E D L E
```


An implementation of naive string search in an imperative language like C is straight forward (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.

```
1  /*
2   * naive-search/search.c --
3   *
4   * Implementation of naive string search in C.
5   */
6
7  #include <stdlib.h>
8  #include "search.h"
9
10 const char *
11 search(const char *haystack, const char *needle)
12 {
13     const char *t, *p, *r;
14
15     for (t = haystack; *t; t++) {
16         for (p = needle, r = t; *r && *p && *r == *p; p++, r++) ;
17         if (! *p) {
18             return t;
19         }
20     }
21
22     return NULL;
23 }
```

Listing 2: Naive string search implemented in C

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., "aaaaaaaaaa" with $n = 10$).
  - Consider a needle of length m which consists of $m - 1$ times the same symbol followed by a single symbol that is different (e.g., "aax" with $m = 3$).
- With $n \gg m$, the number of comparisons needed will be roughly $n \cdot m$.

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

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

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

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

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

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


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 algorithms we would have used 6 alignments. We have performed 7 comparisons in this case while the naive algorithm used 12 comparisons.

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

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

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

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

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

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

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

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

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

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 the 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) if there is no such suffix, skip until the end of the pattern.
- Example: t = `FINDANEEDLEINAHAYSTACK`, p = `NEEDUNEED`

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

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


This example demonstrates case (i) of the good suffix rule.

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

- Example: t = FINDANEEDLEINAHAYSTACK, p = EDISUNEED

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

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


This example demonstrates case (ii) of the good suffix rule.

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

- Example: t = FINDANEEDLEINAHAYSTACK, p = FOODINEED

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

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

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

This example demonstrates case (iii) of the good suffix rule.

The good suffix rule is actually a bit more 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

Boyer-Moore Rules Combined

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

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

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

For further information:

- https://doi.org/10.1145/359146.359148

Section 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!

size n	linear $100n$ µs	quadratic $7n^2$ µs	exponential 2^n µs
1	100 µs	7 µs	2 µs
5	500 µs	175 µs	32 µs
10	1 ms	700 µs	1024 µs
50	5 ms	17.5 ms	13 031.25 d
100	10 ms	70 ms	
1000	100 ms	7 s	
10 000	1 s	700 s	
100 000	10 s	70 000 s	

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).

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 try to use testing techniques in the hope that a good test coverage will reduce errors. But testing never can proof the absense 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 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.

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 known randomized algorithms provide solutions faster than known deterministic solutions. The question for which problems randomized algorithms provide an advantage is an import question investigated in theoretical computer science.

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

Engineering of Software

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

Software engineering (SWE) is the application of engineering to the development of software in a systematic method. The main goal is the production of software with predictable quality and costs. Another definition says software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

For further information:

- https://en.wikipedia.org/wiki/Software_engineering

Part II

Discrete Mathematics

This part will introduce basic elements of discrete mathematics that are relevant for many of the computer science courses.

Section 5: Terminology, Notations, Proofs

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).

Axioms

Definition (axiom)

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

Definition (Peano axioms for natural numbers)

P1 0 is a natural number.

P2 Every natural number has a successor.

P3 0 is not the successor of any natural number.

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

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

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

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

Theorems, Lemma, Corollary

Definition (theorem, lemma, corollary)

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

- There is no clear boundary between what is a theorem, a lemma, or a corollary.
- A proposition for which no proof has been found yet and which is believed to be true is called a *conjecture*.

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

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

Some examples:

- The following statement in mathematical notation

  $$\forall x.\exists y.x = y \Leftrightarrow \neg(x \neq y)$$

  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. Lets assume that the function $s : \mathbb{N} \rightarrow \mathbb{N}$ returns the successor of its argument.

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

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

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

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

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

- 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.

Greek Letters

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

Mathematicians love to use greek letters. And if they run out of greek letters, they love to use roman letters in different writing styles. And to keep reading math fun, every author is free to choose the letters he 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 extend. Different areas of math tend to prefer certain notations and writing styles. As a novice mathematician or programmer, the best advice one can give is to follow the conventions that are used by the seniors around you.

Mathematical Proof

Definition (mathematical proof)

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

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

Proofs Hints

- 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 □.
- Make it easy to understand your proof. A good proof has a clear structure and it is concise.
- 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. □

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

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

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

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

Prove an Implication by its Contrapositive

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

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

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

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

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

Prove an "if and only if" by two Implications

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

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

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

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

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

Proof. We prove x is even implies x^2 is even and vice versa.

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

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.

  Prove P is equivalent to a P' which is equivalent to $[\ldots]$ which is equivalent to Q. □

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

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

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:

  Proof. We prove P by considering the cases $c_1, \ldots, c_N$.
  Case 1: Suppose c_1. Prove of P for c_1.
  ...
  Case N: Suppose c_N. Prove of P for c_N.
  Since P holds for all cases $c1, \ldots c_N$ hold, the statement P 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 $n \in \mathbb{Z}$. □

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

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

  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. Lets consider the case $n+1$:

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

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

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

Section 6: Sets

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., $\{x | x \geq 0 \wedge x < 2^8\}$,
  - stating element-hood using some other statements.
- A set has no order of the elements and every element appears only once.
- The two notations $\{a, b, c\}$ and $\{b, a, a, c\}$ are different *representations* of the same set.

Some popular sets in mathematics:

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

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

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

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

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

Basic Relations between Sets

Definition (basic relations between sets)

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

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

- Obviously:
  - $(A \subseteq B) \wedge (B \subseteq A) \Rightarrow (A \equiv B)$
  - $(A \subseteq B) \Leftrightarrow (B \supseteq A)$

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 $A \cup B = \{x | x \in A \vee x \in B\}$.

Definition (set intersection)

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

Definition (set difference)

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

Some basic properties of set unions:

- $A \cup B = B \cup A$
- $A \cup (B \cup C) = (A \cup B) \cup C$
- $A \subseteq (A \cup B)$
- $A \cup A = A$
- $A \cup \emptyset = A$
- $A \subseteq B \Leftrightarrow A \cup B = B$

Some basic properties of set intersections:

- $A \cap B = B \cap A$
- $A \cap (B \cap C) = (A \cap B) \cap C$
- $A \cap B \subseteq A$
- $A \cap A = A$
- $A \cap \emptyset = \emptyset$
- $A \subseteq B \Leftrightarrow A \cap B = A$

Some basic properties of set differences:

- $A \setminus B \neq B \setminus A$ for $A \neq B$
- $A \setminus A = \emptyset$
- $A \setminus \emptyset = A$

Operations on Sets 2/2

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, $\mathcal{P}(A) = \{S | S \subseteq A\}$.

Definition (cartesian product)

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

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 $f : A \to \mathbb{N}$.

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 straight forward 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

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

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

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

- 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 $\bar{R}$ as $\bar{R} = \{(a, b) \in (A \times B) | (a, b) \notin R\}$.

An good example is the relation of students with their teaching assistants. Let S be the set of students in this course and let T be the set of teaching assistants of this course. Then *is_assigned_to* is a binary relation over $S \times T$ and S is the domain and T is the codomain of this relation.

We sometimes use the notation $dom(R)$ and $codom(R)$ to refer to the domain and the codomain of a relation R.

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 $A = \{a, b, c, d, e, f\}$ and $B = \{1, 2, 3, 4, 5\}$. We define R using the dot graph notation:

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

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

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

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

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

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

Properties of Binary Relations (Endorelations)

Definition

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

- *reflexive* iff $\forall a \in A.(a, a) \in R$
- *irreflexive* iff $\forall a \in A.(a, a) \notin R$
- *symmetric* iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \in R$
- *asymmetric* iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \notin R$
- *antisymmetric* iff $\forall a, b \in A.((a, b) \in R \wedge (b, a) \in R) \Rightarrow a = b$
- *transitive* iff $\forall a, b, c \in A.((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$
- *total* iff $\forall a, b \in A.(a, b) \in R \vee (b, a) \in R$
- *equivalence relation* iff R is reflexive, symmetric, and transitive.

Examples:

- The relation $is\_as\_old\_as$ on the set of persons is reflexive.
- The relation is_older_than on the set of persons is irreflexive.

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

Examples:

- The relation $is\_sibling\_of$ on the set of persons is symmetric.
- The relation is_mother_of on the set of persons is asymmetric.
- The relation $is\_not\_older\_as$ on the set of persons is antisymmetric.
- The relation $is\_ancestor\_of$ on the set of persons transitive.

The relation $is\_as\_old\_as$ is reflexive, symmetric and transitive. Hence it is an equivalence relation.

Note: An equivalence relation induces equivalence classes. Given a set of persons, the $is\_as\_old\_as$ relation induces classes of persons with the same age.

Partial and Strict Order

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 strict partial orders is $\prec$ and a symbol commonly used for non-strict partial orders is $\preceq$.

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 $\mathbb{R}^n$:

$\vec{x} \preceq \vec{y}$ 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.

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

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

Section 8: Functions

Functions

Definition (partial function)

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

Definition (total function)

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

Notation:

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

Function Properties

Definition (injective function)

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

Definition (surjective function)

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

Definition (bijective function)

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

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

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

```
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 often not necessary to give a function a name.
- A function definition of the form $\{(x, y) \in X \times Y | y = E\}$, 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:

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

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

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

- Addition of natural numbers:

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

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

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

For further information:

- 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?
  A: Because 31 Oct = 25 Dec.

For further information:

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

Number Systems in Mathematics

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

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

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

Section 9: Natural Numbers

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:

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

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

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

```
1: function DEC2BIN(n)
2:     s ← ""                                   ▷ 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

```
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:     end while
7:     return n
8: end function
```

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

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

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

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

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

Natural Numbers Literals

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

prefix	example	meaning	description
	42	42_{10}	decimal number
0x	0x42	$42_{16} = 66_{10}$	hexadecimal number
0	042	$42_{8} = 34_{10}$	octal number
0b	0b1000010	$1000010_{2} = 42_{10}$	binary number

- 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.

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

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

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

```c
/*
 * surprises/uint-surprises.c --
 *
 * Surprises with unsigned integers (not only in C).
 */

#include <stdio.h>
#include <stdint.h>

int main()
{
    int max = 321;
    uint8_t n;

    n = UINT8_MAX;
    printf("%d\n", ++n);                /* 127 + 1 = 0 */

    n = 42;
    n += UINT8_MAX;
    printf("%d\n", ++n);                /* 42 + 127 = 42 */

    n -= UINT8_MAX;                     /* 42 - 127 = 42 */
    printf("%d\n", --n);

    for (n = 0; n < max; n++) {           /* endless loop */
        putchar('.');
    }
    putchar('\n');
    return 0;
}
```

Careless choice of fixed precision number ranges can lead to real disasters.

Section 10: Integer Numbers

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 $f : \mathbb{Z} \to \mathbb{N}$ (and the inverse function $f^{-1} : \mathbb{N} \to \mathbb{Z}$):

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

- 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.

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

- We have a fixed number space with n digits and base b to represent integer numbers, that is, we can distinguish at most b^n different integers.
- Lets represent 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: $b = 2, n = 4 : 5_{10} = 0101_2, -5_{10} = 1010_2$

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

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

Two's Complement Fixed Integer Numbers (b complement)

- Like before, we assume a fixed number space with n digits and a base b to represent integer numbers, that is, we can distinguish at most b^n different integers.
- Lets again represent 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 $a'_i = (b-1) - a_i$ and adding 1 to it.
- Example: $b = 2, n = 4 : 5_{10} = 0101_2, -5_{10} = 1011_2$

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

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

Benefits of the two's complement (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:

```
  0010
+ 1010
------
  1100
```

The results 1100_2 is a negative number. Inverting the bits and adding one gives us $0100_2 = 4_{10}$. Hence, the result is -4.

Two's Complement Fixed Integer Number Ranges

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

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

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

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

```
1  /*
2   * surprises/int-surprises.c --
3   *
4   * Surprises with integers (not only in C).
5   */
6
7  #include <stdio.h>
8  #include <stdint.h>
9
10 int main()
11 {
12     int8_t x = 127;
13     uint8_t y = 255;
14     int8_t z = -128;
15
16     printf("%d\n", ++x);              /*  127 + 1 = -128 (!) */
17     printf("%d\n", ++y);              /*  255 + 1 =    0 (!) */
18     printf("%d\n", --z);              /* -128 - 1 =  127 (!) */
19     return 0;
20 }
```

Section 11: Rational and Real Numbers

Rational Numbers

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

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


The equivalent Python code would look like this:

```
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:

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

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

C programmers can use the GNU multiple precision arithmetic library:

```
#include <gmp.h>

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

Real Numbers

- Computer systems usually do not natively represent real numbers, i.e., they cannot compute with real numbers at the hardware level.
- The primary reason is that real numbers like the result of $\frac{1}{7}$ or numbers like π have by definition not a finite representation.
- So the best we can do is to have a finite approximation...
- Since all we have are approximations of real numbers, we *always* make rounding errors 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

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

$$s \times d_0.d_1 d_2 \ldots d_{p-1} \times b^e = s \times \sum_{k=0}^{p-1} d_k b^{-k} \times b^e$$

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 $2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ and the elementary positive charge as $1.602\,176\,634 \times 10^{-19}\,\mathrm{C}$.

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

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

```
1:  function DECF2BINF(f)
2:      s ← ""                                   ▷ s holds a bit string
3:      repeat
4:          f ← f · 2
5:          b ← int(f)                           ▷ int() yields the integer part
6:          append b to the bit string s
7:          f ← frac(f)                          ▷ frac() yields the fractional part
8:      until f = 0                              ▷ may not always be reached
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 ← 0.0                                  ▷ n holds a decimal fraction
3:      while bit string s is not empty do
4:          b ← rightmost bit of the bit string s   ▷ removes the bit from the bit string
5:          f ← (f + b)/2
6:      end while
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:

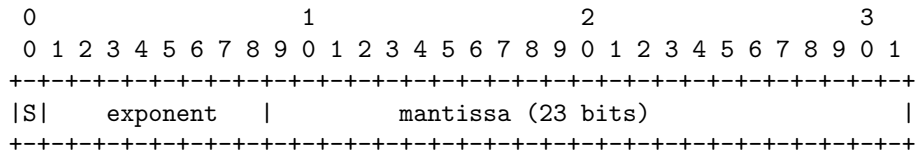
- https://en.wikipedia.org/wiki/Numeric_precision_in_Microsoft_Excel

IEEE 754 Floating Point Formats

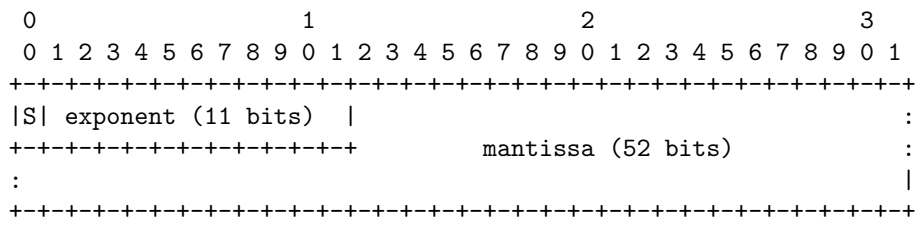
Item	Single precision	Double precision	Quad precision
sign	1 bit	1 bit	1 bit
exponent	8 bit	11 bit	15 bit
mantissa	23 bit	52 bit	112 bit
total size	32 bit	64 bit	128 bit
decimal digits	≈ 7.2	≈ 15.9	≈ 34.0

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

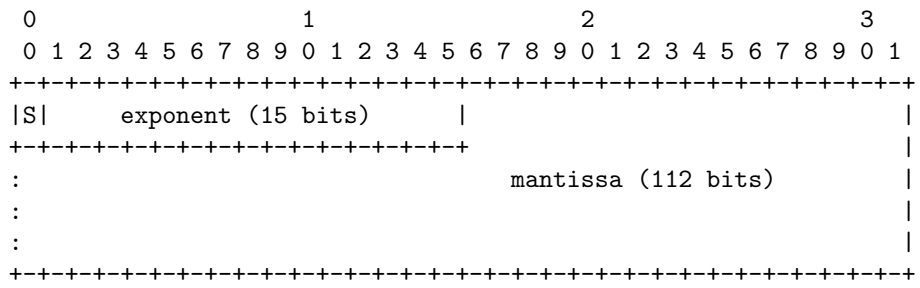
Single precision format:



Double precision format:



Quadruple precision format:



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 $\pm$infinity (inf)
  3. Overflow: returns $\pm$infinity (inf)
  4. Underflow: depends on the operating mode
  5. Inexact: returns rounded result by default
- Note that computations may continue if you hit 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.

Floating Point Surprises

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


```c
/*
 * surprises/double-surprises.c --
 *
 * Surprises with doubles (not only in C).
 */

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

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

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

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

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

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

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

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

For further information:

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

Section 13: International System of Units

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
- https://en.wikipedia.org/wiki/Gimli_Glider

SI Base Units

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

For further information:

- BIPM: “SI Brochure: The International System of Units”, 8th edition, updated 2014
- https://en.wikipedia.org/wiki/SI_base_unit

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:

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

For further information:

- BIPM: “SI Brochure: The International System of Units”, 8th edition, updated 2014
- https://en.wikipedia.org/wiki/SI_derived_unit

Metric Prefixes (International System of Units)

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

Metric Prefixes (International System of Units)

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

Binary Prefixes

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

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

Computers generally access storage using addresses with an address range that is a power of two. Hence, with 30 bits, we can address 2^{30} B $= 1\,073\,741\,824$ 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 B $= 1\,000\,000\,000$ 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

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 rather to other 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.

ASCII Characters and Encoding

- 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.

ASCII Characters and Code Points (decimal)

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

Universal Coded Character Set and Unicode

- The Universal Coded Character Set (UCS) is a standard set of characters defined and maintained by the International Organization of Standardization (ISO).
- As of Unicode 11.0 (June 2018), it contains 137 374 abstract characters, each identified by an unambiguous name and an integer number called its code point.
- The Unicode Consortium produces industry standards based on the UCS for the encoding, representation, and handling of text expressed in most of the world's writing systems.
- 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`.

Unicode Transformation Format UTF-8

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

- 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 $. ☺

Strings

- Let Σ be a non-empty finite set of symbols (or characters), called the alphabet.
- A string (or word) over Σ is any finite sequence of symbols from Σ, including (of course) the empty sequence.
- Typical operations on strings are length(), concatenation(), reverse(), ...
- There are different ways to store strings 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

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:

- https://en.wikipedia.org/wiki/Network_Time_Protocol
- https://en.wikipedia.org/wiki/Precision_Time_Protocol

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://en.wikipedia.org/wiki/Time_zone
- https://www.iana.org/time-zones
- 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.

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

The special letter Z indicates that the date and time is in UTC time. The ISO 8601 standard deals with timezone offsets by specifying the positive or negative offset from UTC time in hours and minutes. This in principle allows us to define +00:00 and -00:00 but the ISO 8601 disallows a negative zero offset. RFC 3339, a profile of ISO 8601, however, defines that -00:00 indicates that the offset is unknown.

For further information:

- https://en.wikipedia.org/wiki/ISO_8601
- 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 digitial 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

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 $\mathbf{X} = \{X_1, X_2, X_3, \ldots\}$.

Definition (Interpretation)

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

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

Boolean $\wedge$ Function (and)

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

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

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

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

- In programming languages like C or C++ (and even Haskell), the operator `&&` is often used to represent the $\wedge$ operation.

Boolean $\vee$ Function (or)

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

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

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

- 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 $\vee$ operation.

Boolean $\neg$ Function (not)

X	$\neg X$
0	1
1	0

- The logical *not* ($\neg$) can be viewed as a unary function that maps a Boolean value to a Boolean value:

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

- The $\neg$ operation simply flips a Boolean value.

- In programming languages like C or C++, the operator `!` is often used to represent the $\neg$ operation (in Haskell you can use the function `not`).

Boolean $\rightarrow$ Function (implies)

X	Y	$X \rightarrow Y$
0	0	1
0	1	1
1	0	0
1	1	1

- The logical *implication* ($\rightarrow$) can be viewed as a function 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.

Boolean $\leftrightarrow$ Function (equivalence)

X	Y	$X \leftrightarrow Y$
0	0	1
0	1	0
1	0	0
1	1	1

- 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 $\dot{\vee}$ Function (exclusive or)

X	Y	$X \dot{\vee} Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

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

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

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

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

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

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

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

X	Y	$X \dot{\vee} Y$	$(X \dot{\vee} Y) \dot{\vee} Y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Boolean ↑ Function (not-and)

X	Y	$X \uparrow Y$
0	0	1
0	1	1
1	0	1
1	1	0

- The logical *not-and* (nand) or ↑ 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 ↑ function is also called Sheffer stroke.

- While we use the functions ∧, ∨, and ¬ 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 not-and is a universal function since it can be used to derive all elementary Boolean functions.

- $X \wedge Y = (X \uparrow Y) \uparrow (X \uparrow Y)$

X	Y	$X \wedge Y$	$X \uparrow Y$	$(X \uparrow Y) \uparrow (X \uparrow Y)$
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

- $X \vee Y = (X \uparrow X) \uparrow (Y \uparrow Y)$

X	Y	$X \vee Y$	$X \uparrow X$	$Y \uparrow Y$	$(X \uparrow X) \uparrow (Y \uparrow Y)$
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	0	0	1

- $\neg X = X \uparrow X$

X	$\neg Y$	$X \wedge X$
0	1	1
1	0	0

Boolean $\downarrow$ Function (not-or)

X	Y	$X \downarrow Y$
0	0	1
0	1	0
1	0	0
1	1	0

- The logical *not-or* (nor) $\downarrow$ 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 $\downarrow$ function is also called Quine arrow.

- The $\downarrow$ (nor) is like $\uparrow$ (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.

- $X \wedge Y = (X \downarrow X) \downarrow (Y \downarrow Y)$
- $X \vee Y = (X \downarrow Y) \downarrow (X \downarrow Y)$
- $\neg X = X \downarrow X$

Section 17: Boolean Functions and Formulas

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.

function	mnemonic	mathematics	engineering	C / C++
and	X and Y	$X \wedge Y$	$X \cdot Y$	`X && Y`
or	X or Y	$X \vee Y$	$X + Y$	`X || Y`
not	not X	$\neg X$	$\overline{X}$	`! X`
implication	X impl Y	$X \rightarrow Y$		
equivalence	X equiv Y	$X \leftrightarrow Y$		`X == Y`
exclusive or	X xor Y	$X \dot{\vee} Y$	$X \oplus Y$	
not and	X nand Y	$X \uparrow Y$		
not or	X nor Y	$X \downarrow Y$		

Boolean Functions

- Elementary Boolean functions $(\neg, \wedge, \vee)$ can be composed to define more complex functions.
- An example of a composed function is

$$\varphi(X, Y) := \neg(X \wedge Y)$$

  which is a function $\varphi : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ and means "first compute the $\wedge$ of X and Y, then apply the $\neg$ on the result you got from the $\wedge$".
- Boolean functions can take a large number of arguments. Here is a function taking three arguments:

$$\varphi(X, Y, Z) := (\neg(X \wedge Y) \vee (Z \wedge Y))$$

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

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

- Implication

$$\rightarrow (X, Y) := \neg X \vee Y$$

- Equivalence

$$\leftrightarrow (X, Y) := (X \wedge Y) \vee (\neg X \wedge \neg Y)$$

- Exclusive or:

$$\dot{\vee}(X, Y) := (X \wedge \neg Y) \vee (\neg X \wedge Y)$$

- Not and:

$$\uparrow (X, Y) := \neg(X \wedge Y)$$

- Not or:

$$\downarrow (X, Y) := \neg(X \vee Y)$$

Boolean Functions

Definition (Boolean function)

A *Boolean function* φ is any function of the type $\varphi : \mathbb{B}^k \to \mathbb{B}$, 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 $\phi(X, Y, Z) := (\neg(X \wedge Y) \vee (Z \wedge Y))$ by filling out the following table:

X	Y	Z	$(\neg(X \wedge Y) \vee (Z \wedge Y))$
0	0	0	1
0	0	1	...
0	1	0	...
0	1	1	...
1	0	0	...
1	0	1	...
1	1	0	...
1	1	1	...

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

1. $(X \wedge Y) = (0 \wedge 0) = 0$
2. $(Z \wedge Y) = (0 \wedge 0) = 0$
3. $\neg(X \wedge Y) = \neg 0 = 1$
4. $(\neg(X \wedge Y) \vee (Z \wedge Y)) = 1 \vee 0 = 1$

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 $X \wedge Y$ and $X + Y$ is used instead of $X \vee Y$ and $\overline{X}$ is used instead of $\neg X$.

$\phi(X, Y, Z) = \overline{(X \cdot Y)} + (Z \cdot Y)$

Syntax of Boolean formulas (aka Boolean expressions)

Definition (Syntax of Boolean formulas)

Basis of inductive definition:

1a Every Boolean variable X_i is a Boolean formula.

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

Induction step:

2a If φ and ψ are Boolean formulas, then $(\varphi \wedge \psi)$ is a Boolean formula.

2b If φ and ψ are Boolean formulas, then $(\varphi \vee \psi)$ is a Boolean formula.

2c If φ 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, $(\neg(X \wedge Y) \vee (Z \wedge Y))$ is a valid Boolean formula and $\neg(X\wedge)$ is not.

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

Furthermore, we may write:

- $(X \rightarrow Y)$ as a shorthand notation for $(\neg X \vee Y)$
- $(X \leftrightarrow Y)$ as a shorthand notation for $((\neg X \vee Y) \wedge (\neg Y \vee X))$

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.

Semantics of Boolean formulas

Definition (Semantics of Boolean formulas)

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

Basis of the inductive definition:

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

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

Because this generalized interpretation $\mathcal{I}^*$ is the same as $\mathcal{I}$ for Boolean variables $X \in D$, we say that $\mathcal{I}^*$ *extends* $\mathcal{I}$ from the domain D to the domain $\Phi(D)$. Following common practice, we will use $\mathcal{I}$ 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 φ and ψ in $\Phi(\mathbf{D})$:

2a

$$\mathcal{I}^*((\varphi \wedge \psi)) = \begin{cases} 1 & \text{if } \mathcal{I}^*(\varphi) = 1 \text{ and } \mathcal{I}^*(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

2b

$$\mathcal{I}^*((\varphi \vee \psi)) = \begin{cases} 1 & \text{if } \mathcal{I}^*(\varphi) = 1 \text{ or } \mathcal{I}^*(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

2c

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

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

Tautology and contradiction

Definition (adapted interpretation)

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

Definition (tautologies and contradictions)

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

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

$$(X \vee \neg X)$$

and the classical example of a contradiction is

$$(X \wedge \neg X).$$

Two elementary facts relating to tautologies and contradictions:

- For any Boolean expression φ, $(\varphi \vee \neg\varphi)$ is a tautology and $(\varphi \wedge \neg\varphi)$ is a contradiction.
- If φ is a tautology, then $\neg\varphi$ is a contradiction and vice versa.

For a complex Boolean formula φ 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 φ. Recall furthermore that if φ 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 φ 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 $\mathcal{I}$ which is adapted to a Boolean formula φ is said to *satisfy* the formula φ if $\mathcal{I}(\varphi) = 1$. A formula φ is called *satisfiable* if there exists an interpretation which satisfies φ.

The following two statements are equivalent characterizations of satisfiability:

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

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

Equivalence of Boolean formulas

Definition (equivalence of Boolean formulas)

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

- 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 $(X \vee Y) \equiv (Y \vee X)$.

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

$$(X \vee Y) \equiv (X \vee Y) \wedge (Z \vee \neg Z)$$

Equivalence laws

Proposition (equivalence laws)

For any Boolean formulas φ, ψ, χ, the following equivalences hold:

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

φ	ψ	$\neg\varphi$	$\neg\psi$	$(\varphi \wedge \psi)$	$\neg(\varphi \wedge \psi)$	$(\neg\varphi \vee \neg\psi)$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

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

Try to simplify the following formula:

$$((X \vee Y) \wedge (\neg Y \wedge Z)) \wedge Z = \ldots = X \wedge \neg Y \wedge Z$$

Alternatively, simplify the formula using a truth table.

Section 19: Normal Forms (CNF and DNF)

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 $(1 \vee 1) \wedge (X_1 \vee 0)$)
  - $X_1 \wedge X_2$ (this is a short form of $(X_1 \vee X_1) \wedge (X_2 \vee X_2)$)
  - $X_1 \vee X_2$ (this is a short form of $(1 \vee 1) \wedge (X_1 \vee X_2)$)
  - $\neg X_1 \wedge (X_2 \vee X_3)$ (this is a short form of $(0 \vee \neg X_1) \wedge (X_2 \vee X_3)$)
  - $(X_1 \vee \neg X_2) \wedge (\neg X_1 \vee X_2)$
- 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 $(0 \wedge 0) \vee (X_1 \wedge 1)$)
  - $X_1 \wedge X_2$ (this is a short form of $(0 \wedge 0) \vee (X_1 \wedge X_2)$)
  - $X_1 \vee X_2$ (this is a short form of $(X_1 \wedge X_1) \vee (X_2 \wedge X_2)$)
  - $(\neg X_1 \wedge X_2) \vee (\neg X_1 \wedge X_3)$
  - $(\neg X_1 \wedge \neg X_2) \vee (X_1 \wedge X_2)$
- 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,

$$\varphi(X,Y,Z) = (\neg X \wedge \neg Y \wedge Z) \vee (\neg X \wedge Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge Z) \vee (X \wedge Y \wedge \neg Z) \vee (X \wedge Y \wedge Z)$$

can be written as

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

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 φ is equivalent to a Boolean formula χ in conjunctive normal form.

Proposition (DNF equivalence)

Every Boolean formula φ is equivalent to a Boolean formula χ 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, $\chi = (1 \vee 1) \wedge (\varphi \vee 0)$ is a formula in CNF that is equivalent to X_i.

$$
\begin{aligned}
\chi &= (1 \vee 1) \wedge (\varphi \vee 0) \\
&= 1 \wedge (\varphi \vee 0) && \text{(domination)} \\
&= \varphi \vee 0 && \text{(identity)} \\
&= \varphi && \text{(identity)} \\
&= X_i
\end{aligned}
$$

1b Let φ 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 $\varphi = (\chi_1 \wedge \chi_2)$ is a conjunction of two formulas χ_1 and χ_2, which we assume to be in CNF. Due to the associativity of $\wedge$ we can drop the parenthesis to obtain $\chi = \chi_1 \wedge \chi_2$ in CNF.

2b Assume that $\varphi = (\chi_1 \vee \chi_2)$ is a disjunction of two formulas χ_1 and χ_2, which we assume to be in CNF.

complete this

2c Assume that $\varphi = \not{\chi}_1$ is a negation of the formula χ_1, which we assume to be in CNF.

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.

X	Y	$X \dot{\vee} Y$
0	0	0
0	1	1
1	0	1
1	1	0

- 2nd row: $\neg X \wedge Y$
- 3rd row: $X \wedge \neg Y$
- $\chi = (\neg X \wedge Y) \vee (X \wedge \neg Y)$

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.

X	Y	$X \dot{\vee} Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

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 $(X \vee Y) \wedge (\neg X \vee \neg Y)$ is indeed the same as $(\neg X \wedge Y) \vee (X \wedge \neg Y)$:

$$\begin{aligned}
(X \vee Y) \wedge (\neg X \vee \neg Y) &= ((X \vee Y) \wedge \neg X) \vee (X \vee Y) \wedge \neg Y) \\
&= (X \wedge \neg X) \vee (Y \wedge \neg X) \vee (X \wedge \neg Y) \vee (Y \wedge \neg Y) \\
&= 0 \vee (Y \wedge \neg X) \vee (X \wedge \neg Y) \vee 0 \\
&= (\neg X \wedge Y) \vee (X \wedge \neg Y)
\end{aligned}$$

Section 20: Complexity of Boolean Formulas

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

Implicants and Prime Implicants

Definition (implicant)

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

Definition (prime implicant)

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

Definition (essential prime implicant)

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

Observations:

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

Example:

$$\begin{aligned}\varphi(X,Y,Z) &= (\neg X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge Z) \vee (X \wedge Y \wedge Z) \\ &= (\neg Y \wedge \neg Z) \vee (X \wedge Z)\end{aligned}$$

- Implicant $(\neg X \wedge \neg Y \wedge \neg Z)$ is not a prime implicant. The first two product terms can be combined since they only differ in one variable:

  $$\begin{aligned}(\neg X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge \neg Z) &= (\neg X \wedge X) \vee (\neg Y \wedge \neg Z) \\ &= 0 \vee (\neg Y \wedge \neg Z) \\ &= (\neg Y \wedge \neg Z)\end{aligned}$$

  The resulting product term $\neg Y \wedge \neg Z$ is still an implicant of φ. In a similar way, Y can be eliminated from the last two product terms.

- $(\neg Y \wedge \neg Z)$ and $(X \wedge Z)$ 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

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

- Combination steps

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

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

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 columnmns 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.

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

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

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

The resulting minimal expression is $\varphi'(W,X,Y,Z) = (X \wedge \neg Y \wedge \neg Z) \vee (W \wedge \neg X) \vee (W \wedge Y)$. The minimal expression φ' uses 6 operations (out of $\{\wedge, \vee\}$). The original expression φ used $8 \cdot 3 + 7 = 31$ operations (out of $\{\wedge, \vee\}$).

Section 21: Boolean Logic and the Satisfiability Problem

Logic Statements

- A common task is to decide whether statements of the form
  if *premises* P_1 **and** ... **and** P_m *hold,* **then** *conclusion* C *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

$$(\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi$$

  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 τ is a tautology if and only if $\tau' = \neg\tau$ is a contradiction. Furthermore, a Boolean formula is a contradiction if and only if it is not satisfiable. Hence, in order to check whether

$$\tau = (\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi \tag{1}$$

  is a tautology, we may check whether

$$\tau' = \neg((\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi) \tag{2}$$

  is unsatisfiable.
- If we show that τ' is satisfiable, we have disproven τ.

Tautology and Satisfiability

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

$$\tau = (\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi = \neg(\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi) \tag{3}$$

$$\tau' = \neg((\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi) = (\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi) \tag{4}$$

- To disprove τ, it is often easier to prove that τ' is satisfiable.
- Note that τ' has a homogenous structure. If we transform the elements $\varphi_1, \ldots, \varphi_m, \psi$ into CNF, then the entire formula is in CNF.
- If τ' is in CNF, all we need to do is to invoke an algorithm that searches for interpretations $\mathcal{I}$ which satisfy a formula in CNF. If there is such an interpretation, τ is disproven, otherwise, if there is no such interpretation, then τ is proven.

Satisfiability Problem

Definition (satisfiability problem)

The satisfiability problem (SAT) is the following computational problem: Given as input a Boolean formula in CNF, compute as output a "yes" or "no" response according to whether the input formula is satisfiable or not.

- It is believed that there is no polynomial time solution for this problem.

There is no known general algorithm that efficiently (means in polynomial time) solves the SAT problem, and it is generally believed that no such algorithm exists. However, this belief has not been proven mathematically. Resolving the question whether SAT has a polynomial-time algorithm is equivalent to answering the P versus NP problem, which is a famous open problem in the theory of computer science.

Note that it is rather trivial to check whether a Boolean formula in DNF is satisfiable since it is sufficient to show that one of the conjunctions is satisfiable (since every conjunction is an implicant). A conjunction is satisfiable if it does not contain X and $\neg X$ for some variable X. Given an arbitrary Boolean formula, the conversion into DNF may unfortunately require exponential time.

By solving the general SAT problem, you will become a famous mathematician and you can secure a one million dollar price.

For further information:

- https://en.wikipedia.org/wiki/Boolean_satisfiability_problem
- https://en.wikipedia.org/wiki/Millennium_Prize_Problems

Part V

Computer Architecture and System Software

This part takes a systems' view on information and communication technology, moving from elementary circuits to processors to operating systems and higher programming languages. This is a major *zoom out* from elementary boolean logic gates to distributed computing and pretty much resembles a zoom out on Google Earth from the level of a brick of a building to a complete view of the earth.

The details we will cover:

1. From logic gates via logic circuits that can add numbers to an elementary machine language of a simple processor.
2. From machine language to abstractions provided by operating systems.
3. From machine and assembly languages up via compilers and interpreters to high-level programming languages.

Given the time constraints, we will only touch on many interesting topics. If you want to dive deeper, consider joining one of the core courses such as Computer Architecture, Operating Systems, or Computer Networks.

Section 22: Logic Gates and Digital Circuits

Recall elementary boolean operations and functions

- Recall the elementary boolean operations AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) as well as the boolean functions XOR ($\dot{\vee}$), NAND ($\uparrow$), and NOR ($\downarrow$).

$$X \dot{\vee} Y := (X \vee Y) \wedge \neg(X \wedge Y)$$
$$X \uparrow Y := \neg(X \wedge Y)$$
$$X \downarrow Y := \neg(X \vee Y)$$

- 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.

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

X	$\neg X$
0	1
1	0

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

X	Y	$X \dot{\vee} Y$
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	$X \uparrow Y$
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	$X \downarrow Y$
0	0	1
0	1	0
1	0	0
1	1	0

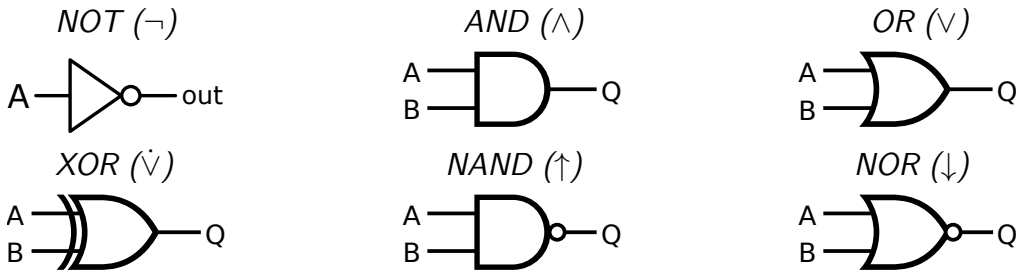
Figure 2: Truth tables for the elementary functions XOR, NAND, and NOR

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:

- http://en.wikipedia.org/wiki/Boolean_algebra

Logic gates implementing logic functions



- 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.

Addition of decimal and binary numbers

```
  2      0010        3      0011        8      1000
+ 5    + 0101      + 3    + 0011      + 3    + 0011
                                        1
---    ------      ---    ------      ---    ------
  7      0111        6      0110       11      1011
```

- 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.

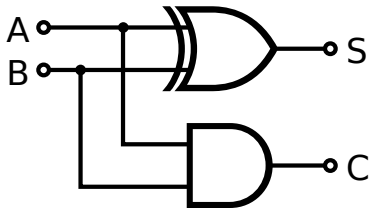
Adding two bits (half adder)

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

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \dot{\vee} B$$

$$C = A \wedge B$$



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:

- http://en.wikipedia.org/wiki/Adder_%28electronics%29

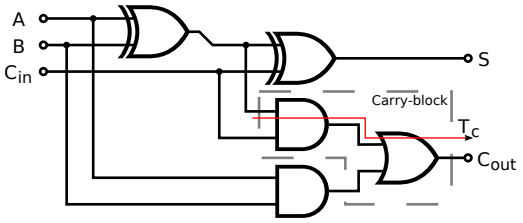
Adding two bits (full adder)

- 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}).

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

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

$$C_{out} = (A \wedge B) \vee (C_{in} \wedge (A \dot{\vee} B))$$



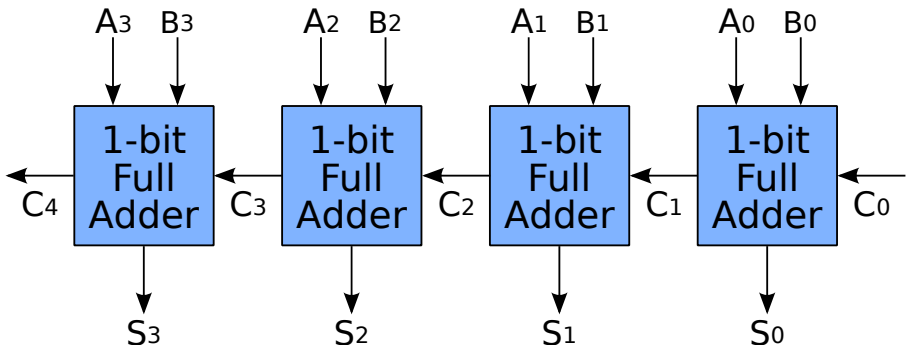
Assuming that every gate has the same gate delay t_g, the full adder works with a constant gate delay of $t_{fa} = 3 \cdot t_g$.

Further information:

- http://en.wikipedia.org/wiki/Adder_%28electronics%29

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 from 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 $t_{ra} = n \cdot t_{ha} = 3n \cdot t_g$. 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 $a_i = 1$ and $b_i = 1$ or it is the result of $c_i = 1$ and either $a_i = 1$ or $b_i = 1$.

$$\begin{aligned} c_{i+1} &= (a_i \wedge b_i) \vee ((a_i \dot{\vee} b_i) \wedge c_i) = g_i \vee (p_i \wedge c_i) \\ g_i &= a_i \wedge b_i \\ p_i &= a_i \dot{\vee} b_i \end{aligned}$$

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.

$$\begin{aligned} c_0 &= c_0 \\ c_1 &= g_0 \vee (p_0 \wedge c_0) \\ c_2 &= g_1 \vee (p_1 \wedge c_1) \\ &= g_1 \vee (p_1 \wedge (g_0 \vee (p_0 \wedge c_0))) \\ &= g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge c_0) \\ c_3 &= g_2 \vee (p_2 \wedge c_2) \\ &= g_2 \vee (p_2 \wedge (g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge c_0))) \\ &= g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge c_0) \\ c_4 &= g_3 \vee (p_3 \wedge c_3) = \ldots \\ &= g_4 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1) \vee (p_3 \wedge p_2 \wedge p_1 \wedge g_0) \vee (p_3 \wedge p_2 \wedge p_1 \wedge p_0 \wedge c_0) \end{aligned}$$

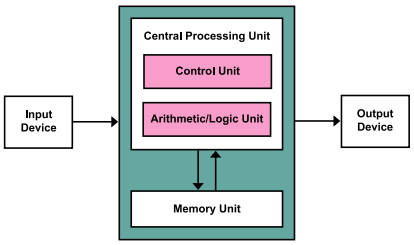
Note that g_i and p_i are exactly the functions of our half adder ($C = g_i$, $S = p_i$). Hence, we can use n half adders to produce $g_0, \ldots, g_{n-1}$ and $p_0, \ldots, p_{n-1}$. We then create a circuit to calculate $c_1, \ldots c_n$ following the expansion scheme above and we finally use n half 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 logartihmically with the number of bits. The price for this improvement is a more complex digital circuit.

Further information:

- http://en.wikipedia.org/wiki/Adder_%28electronics%29

Section 23: Von Neumann Computer Architecture

Von Neumann computer architecture



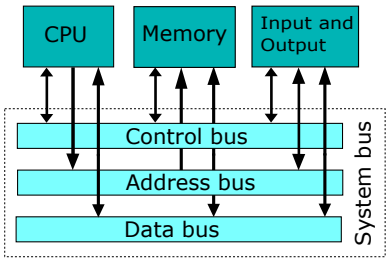
- Control unit contains an instruction register and a program counter
- Arithmetic/logic unit (ALU) performs integer arithmetic and logical operations
- Program instructions and data is stored in a memory unit
- Processor registers provide small amount of storage as part of a central processing unit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

- Mass storage and input/output devices communicate with the central processing unit.
- The memory unit stores both data and program instructions.
- The ALU contains many basic digital circuits, such as a N-bit adder.
- The processor usually operates on binary data with a certain number of bits (8-bit processor, 16-bit processor, 32-bit processor, 64-bit processor).
- The processor is controlled by a clock (usually measured in GHz) that drives the actions a central processing unit is performing.
- Instructions may need one or multiple clock cycles to be carried out.
- A central processing unit may try to “overlap” the execution of instructions (e.g., fetch and decode the next instruction while the current instruction is carried out in the ALU).
- Modern systems often have multiple tightly integrated central processing units (often called cores) in order to perform computations concurrently.

Further information:

- http://en.wikipedia.org/wiki/Von_Neumann_architecture

Computer system bus (data, address, and control)



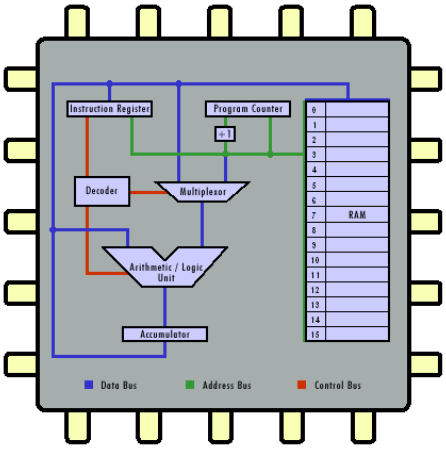
- The *data bus* transports data (primarily between registers and main memory).
- The *address bus* selects which memory cell is being read or written.
- The *control bus* activates components and steers the data flow over the data bus and the usage of the address bus.

- Parallel busses data words in parallel on multiple wires
- Serial busses carry data in bit-serial form over simple wires
- Parallel busses are highly efficient if the distance to cover is small.
- Serial busses 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:

- http://en.wikipedia.org/wiki/Bus_%28computing%29

Simple Central Processing Unit



- Real CPUs usually have multiple registers
- Real CPUs support memory outside of the CPU itself
- Real CPUs have different instruction sets for different privilege levels
- Real CPUs have special digital circuits for floating point arithmetic or cryptographic operations

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?

Instruction cycle (fetch – decode – execute cycle)

```
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/Instruction_cycle
- http://en.wikipedia.org/wiki/Machine_code
- http://en.wikipedia.org/wiki/Assembly_language

Simple Machine Language

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

- 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:

- http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html

Program #1 in our simple machine language

#	Machine Code	Assembly Code	Description
0	001 1 0010	`LOAD #2`	Load the value 2 into the accumulator
1	010 0 1101	`STORE 13`	Store the value of the accumulator in memory location 13
2	001 1 0101	`LOAD #5`	Load the value 5 into the accumulator
3	010 0 1110	`STORE 14`	Store the value of the accumulator in memory location 14
4	001 0 1101	`LOAD 13`	Load the value of memory location 13 into the accumulator
5	011 0 1110	`ADD 14`	Add the value of memory location 14 to the accumulator
6	010 0 1111	`STORE 15`	Store the value of the accumulator in memory location 15
7	111 0 0000	`HALT`	Stop execution

- An animation of the execution of this program can be found here:
  `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.

```
MEM = 0x32 4d 35 4e 2d 6e 4f e0 00 00 00 00 00 00 00 00

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

MEM = 0x32 4d 35 4e 2d 6e 4f e0 00 00 00 00 00 02 05 07
```

Program #2 in our simple machine language

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

- An animation of the execution of this program can be found here:
  `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 00

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

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

Section 24: Interpreter and Compiler

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, Pascal, Fortran, Cobol, ...

Examples of (typically) interpreted programming languages:

- Python, PHP, JavaScript, Perl, Basic, ...

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>:
  4004ac:       55                      push   %rbp
  4004ad:       48 89 e5                mov    %rsp,%rbp
  4004b0:       c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
  4004b7:       c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
  4004be:       8b 45 f8                mov    -0x8(%rbp),%eax
  4004c1:       8b 55 fc                mov    -0x4(%rbp),%edx
  4004c4:       01 d0                   add    %edx,%eax
  4004c6:       89 45 f4                mov    %eax,-0xc(%rbp)
  4004c9:       8b 45 f4                mov    -0xc(%rbp),%eax
  4004cc:       5d                      pop    %rbp
  4004cd:       c3                      retq
  4004ce:       90                      nop
  4004cf:       90                      nop
```


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(%rpb) 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
```

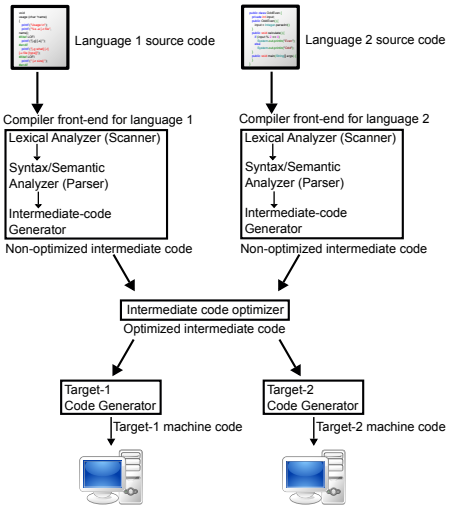

With optimization turned on, the compiler has detected that the expression only depends on constants and thus the expression can already be evaluated at compile time to improve speed at runtime. Furthermore, the compiler has figured out that the variables `a`, `b`, and `c` are not used outside of the 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 usually also possible to ask the compiler to optimize for space.

Compiler



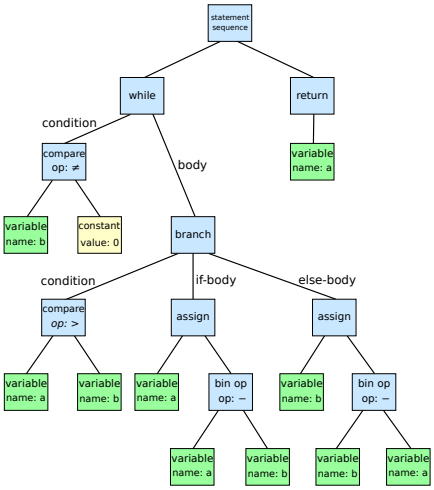
- lexical analysis
  $\Rightarrow$ sequence of token
- syntax analysis
  $\Rightarrow$ parse tree
- semantic analysis
  $\Rightarrow$ abstract syntax tree
- optimization
  $\Rightarrow$ enhanced abstract syntax tree
- code generation
  $\Rightarrow$ object code

- 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 usually 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 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:

- http://en.wikipedia.org/wiki/Compiler

Abstract Syntax Tree Example



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

```
while (b != 0):
    if (a > b):
        a = a - b
    else:
        b = b - a
return a
```

Backus-Naur-Form and Formal Languages

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 ::= and a rule expression
- A rule expression consists of terminal and non-terminal symbols and operators; the empty operator denotes contatenation 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 $\Sigma = \{0, 1, \ldots, 9, x, y, z\}$.

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

Using `<expression>` as a start symbol, the grammar defines a simplified format of expressions. Here is a sample step-by-step deviation:

```
    <expression>
->  <expression> + <term>
->  <term> + <term>
->  <factor> + <term>
->  <factor> + <term> * <factor>
->  <constant> + <term> * <factor>
->  <digit> <constant> + <term> * <factor>
->  <digit> <digit> + <term> * <factor>
->  <digit> <digit> + <factor> * <factor>
->  <digit> <digit> + <constant> * <factor>
->  <digit> <digit> + <digit> * <factor>
->  <digit> <digit> + <digit> * <variable>
->  42+8*x
```

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.

Interpreter

- A basic interpreter parses a statement, executes it, and moves on to the next statement (very similar to a fetch-decode-execute cycle).
- More advanced interpreter do a syntactic analysis to determine syntactic correctness before execution starts.
- Properties:
  - Highly interactive code development (trial-and-error coding)
  - Limited error detection capabilities before code execution starts
  - Interpretation causes a certain runtime overhead
  - Development of short pieces of code can be very fast
- Examples: command interpreter (shells), scripting languages

Interpreters often implement read–eval–print loops (REPLs). REPLs facilitate exploratory programming because the programmer can inspect the result before deciding what 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

[2] Source Code --> Compiler --> Machine Code

[3] Source Code --> Compiler --> Byte Code --> Interpreter

[4] Source Code --> Compiler --> Byte Code --> Compiler --> Machine Code
```

- 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:

- http://en.wikipedia.org/wiki/Register_machine
- http://en.wikipedia.org/wiki/Stack_machine

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).

$\Rightarrow$ Virtual machines were invented in the 1970s and reinvented in the 1990s.

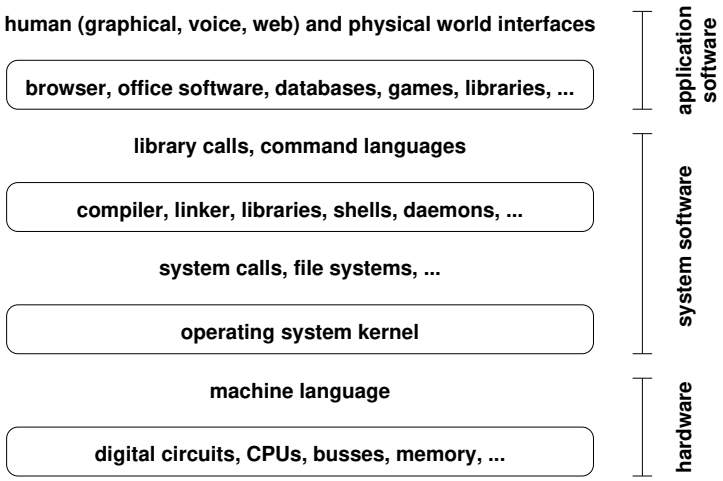
$\Rightarrow$ Virtual machines have been an enabler for cloud computing since they are easy to start / stop / clone / migrate and they separate the software implementing services form the underlying hardware.

- 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.

The CORE modules of the CS bachelor program have been structured along this hierarchy of abstractions:

- The CORE module Technical Computer Science covers the hardware and programming language view (computer architecture and programming languages), the functions of an operating system kernel and concurrent programming (operating systems), and network communication (which is partially implemented as part of operating systems).
- The CORE module Applied Computer Science covers databases and applications using a web frontend to interact with users and other services (databases and web services), how graphics and animations can be programmed (computer graphics), and how software can be produced using engineering principles (software engineering).
- The CORE module Theoretical Computer Science covers the foundations of computer science (formal languages and logic, computability and complexity) and it discusses how to construct secure systems we can rightfully depend on (secure and dependable systems).

Section 25: 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:

```
# ------------------------------------------------------------------------
# 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:                           # This is called by C library's startup code
        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:

```
# ------------------------------------------------------------------------
# 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"
```

OS Abstraction #1: Processes and Process Lifecycle

Definition (process)

An instance of a computer program that is being executed is called a *process*.

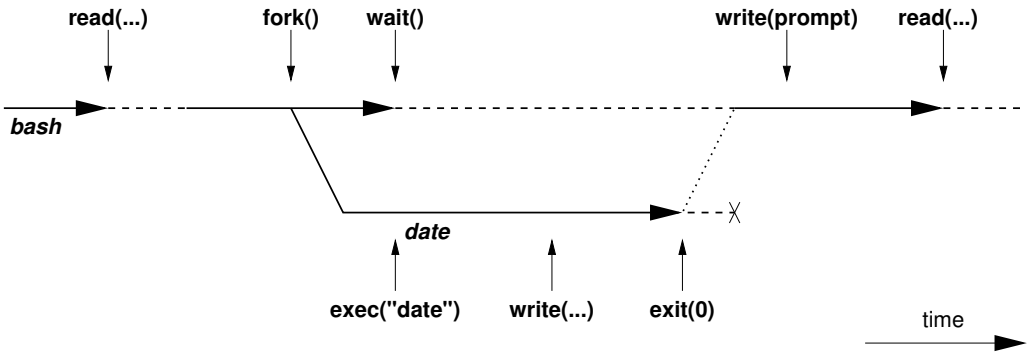
- The OS kernel maintains information about each running process and assigns resources and ensures protection of concurrently running processes.
- In Unix-like Operating Systems
  - a new process is created by "cloning" (forking) an already existing process
  - a process may load a new program image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

$\Rightarrow$ 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.

OS Abstraction #1: Processes and Process Lifecycle



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 the 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 */
    }
}
```



```
#!/usr/bin/env python3.4

import os
import sys

def run(token):
    pid = os.fork()
    if not pid:
        try:
            os.execvp(token[0], token)
        except (OSError) as e:
            print("pysh: %s" % e.strerror)
            os._exit(1)
    return os.wait()[0]

def main():
    while True:
        sys.stdout.write("pysh > ")
        sys.stdout.flush()
        line = sys.stdin.readline()
        if not line or line == "exit\n":
            break
        if line == "\n":
            continue
        run(line.strip().split(' '))
    print("pysh: have a nice day")

if __name__ == "__main__":
    main()
```

OS Abstraction #2: File Systems

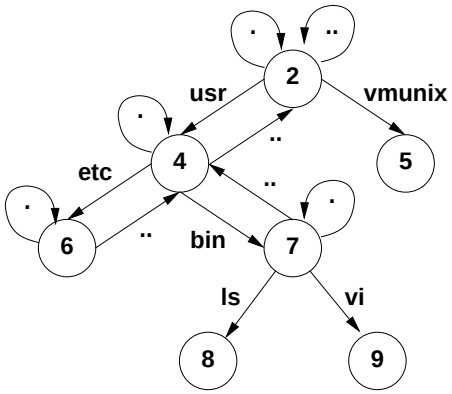
- Files are persistent containers for the storage of data
- Unstructured files contain a sequence of bytes
- Applications interpret the content of a file in a specific way
- Files also have meta data (owner, permissions, timestamps)
- Hierarchical file systems use directories to organize files into a 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:

- http://en.wikipedia.org/wiki/File_system

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 refering 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


```
1  #!/usr/bin/env python3.4
2
3  import os
4  import pwd
5  import grp
6  import time
7  import sys
8
9  def ls(path):
10     if (os.path.isdir(path)):
11         for entry in os.listdir(path):
12             ls(os.path.join(path, entry))
13     else:
14         s = os.stat(path)
15         print("%-8s %-8s %6d %s %s"
16             % (pwd.getpwuid(s.st_uid)[0],
17                grp.getgrgid(s.st_gid)[0],
18                s.st_size,
19                time.strftime("%Y-%m-%dT%H:%M:%S", time.localtime(s.st_mtime)),
20                path))
21
22 for arg in sys.argv[1:]:
23     print(arg)
24     ls(arg)
```

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


```
#!/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

Part VI

Automata and Formal Languages

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them. It is a theory in theoretical computer science and discrete mathematics.

The word automata (the plural of automaton) comes from the Greek word αὐτόματα, which means ”self-acting”. Some automata serve pure theoretical purposes, others are practically used to describe aspects of software systems or to implement certain functionalities.

A formal language is a set of words formed over an alphabet (a set of symbols). A formal language is often defined by means of a formal grammar.

Formal languages theory studies what kind of languages different types of grammars can describe and it studies the problem to decice whether a given sequence of symbols belongs to a formal language or not (a decision problem).

Formal languages and automata theory are closely related since the decision problem (whether a given input belongs to a certain language) can be solved for different classes of languages with different types of automata.

Section 26: Finite State Machines

Finite State Machine

Definition (finite state machine)

A *finite state machine* (FSM) is a quintuple $(\Sigma, S, s_0, \delta, F)$ where:

- Σ is the input alphabet (a finite, non-empty set of symbols)
- S is a finite, non-empty set of states
- s_0 is an initial state ($s_0 \in S$)
- δ is the state-transition function, $\delta : S \times \Sigma \to S$
- F is a possibly empty set of accepting states ($F \subset S$)

- We sometimes say that a FSM *accepts* an input word $w \in \Sigma^*$ if the machine, starting from the state s_0, processes all symbols of w and reaches one of the accepting states in F.

A finite state machine has a finite set of internal states. Starting from an initial state, the machine consumes in every computational step an input symbol. The current input symbol and the current state determine the followup state. The finite state machine stops when all input symbols have been consumed. If the machine stops in a final state, then we say that the finite state machine accepted the input.

In a deterministic finite state machine, every state has exactly one transition for each possible input. In a non-deterministic finite state machine, an input can lead to one, more than one, or no transition for a given state.

Finite state machines can describe many different processes. Some examples:

- Traffic lights can be described using finite state machines. Given an infinite sequence of input pulses, the traffic lights move through a sequence of states representing the sequence of possible traffic light color combinations.
- A computer program may use a state machine in order to determine whether a sequence of input symbols contains a certain pattern.
- The communication between computer systems is often controlled by state machines that determine the next possible steps that can be taken.
- An espresso machine may use a state machine to control the different steps of the coffee brewing process.

Finite State Machine Example ($a^n b^m$)

- The FSM $(\Sigma, S, s_0, \delta, F)$ with $\Sigma = \{\text{a, b}\}$, $S = \{\text{S0, S1, S2, S3}\}$, $s_0 = S0$, $F = \{S2\}$, and

$$\begin{aligned}\delta = \{&((S0, a), S1), ((S0, b), S3),\\ &((S1, a), S1), ((S1, b), S2),\\ &((S2, a), S3), ((S2, b), S2)\}\end{aligned}$$

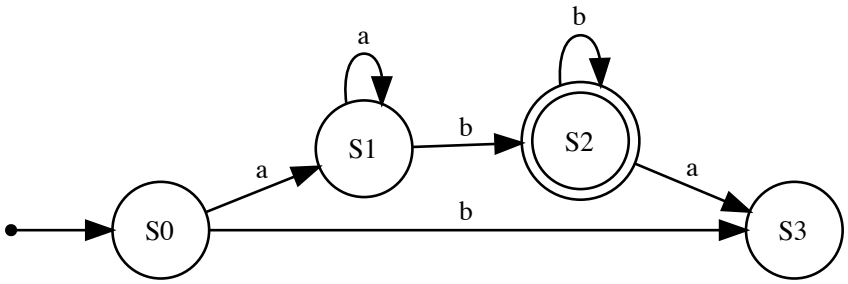
  recognizes all words of the form $\{a^n b^m | n \geq 1, m \geq 1\}$.

- The set of words $w \in \Sigma^*$ accepted by a finite state is called the language recognized by the finite state machine.

An implementation of this finite state machine in Haskell:

```
1  data State = S0 | S1 | S2 | S3
2
3  accepts :: State -> String -> Bool
4  accepts S0 ('a':xs) = accepts S1 xs
5  accepts S0 ('b':xs) = accepts S3 xs
6  accepts S1 ('a':xs) = accepts S1 xs
7  accepts S1 ('b':xs) = accepts S2 xs
8  accepts S2 ('a':xs) = accepts S3 xs
9  accepts S2 ('b':xs) = accepts S2 xs
10 accepts S2 [] = True
11 accepts _ _ = False
12
13 decide :: String -> Bool
14 decide = accepts S0
```

FSM Example ($a^n b^m$) Represented as a Graph



- State machines can be represented as graphs where nodes (circles) represent states, arrows represent state transitions, an arrow pointing from a small black circle indicates the initial state, and accepting states are marked with with a double circle.

The table below shows an execution for the string `aaabbb`. Lines are prefixed with the current state of the pushdown automaton. Lines starting with a vertical bar indicate a transition and the input symbol processed.

```
S0:
| a
S1:
| a
S1:
| a
S1:
| b
S2:
| b
S2:
| b
S2:
```

Finite State Machine Example (integer)

- Consider the problem of deciding whether an input string contains an integer number, that is, whether it consists of at least one digit and an optional '-' at the very beginning (valid numbers would be -1, 0, -42, 42)
- The FSM $(\Sigma, S, s_0, \delta, F)$ with $\Sigma = \{$-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$\}$, $S = \{$S0, S1, S2, S3$\}$, $s_0 = S0$, $F = \{S2\}$, and

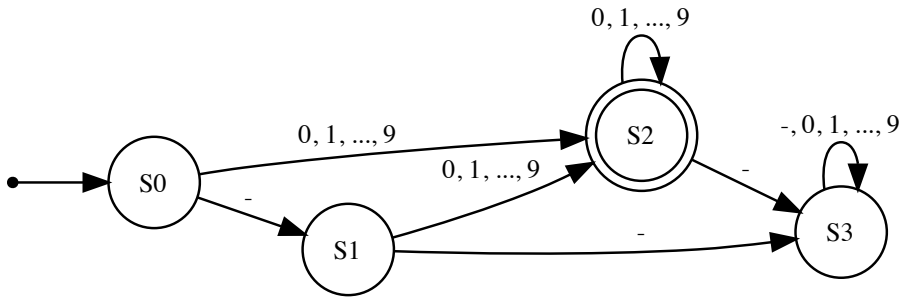
$$\begin{aligned}\delta = \{&((S0, -), S1), ((S0, 0), S2), \ldots, ((S0, 9), S2),\\ &((S1, -), S3), ((S1, 0), S2), \ldots, ((S0, 9), S2),\\ &((S2, -), S3), ((S2, 0), S2), \ldots, ((S2, 9), S2),\\ &((S3, -), S3), ((S3, 0), S3), \ldots, ((S3, 9), S3)\}\end{aligned}$$

  solves this integer number parsing problem.

An implementation of this finite state machine in Haskell:

```
1  import Data.Char (isDigit)
2
3  data State = S0 | S1 | S2 | S3
4
5  accepts :: State -> String -> Bool
6  accepts S0 ('-':xs) = accepts S1 xs
7  accepts S0 (x:xs)
8    | isDigit x = accepts S2 xs
9  accepts S1 ('-':xs) = accepts S3 xs
10 accepts S1 (x:xs)
11   | isDigit x = accepts S2 xs
12 accepts S2 ('-':xs) = accepts S3 xs
13 accepts S2 (x:xs)
14   | isDigit x = accepts S2 xs
15 accepts S2 [] = True
16 accepts _ _ = False
17
18 decide :: String -> Bool
19 decide = accepts S0
```

FSM Example (integer) Represented as a Graph



Section 27: Pushdown Automaton

Pushdown Automaton

Definition (pushdown automaton)

A *pushdown automaton* (PDA) is a 7-tuple $(\Sigma, S, s_0, \Gamma, Z, \delta, F)$ where:

- Σ is the input alphabet (a finite, non-empty set of symbols)
- S is a finite, non-empty set of states
- s_0 is an initial state ($s_0 \in S$)
- Γ is a finite, non-empty stack alphabet
- Z is the initial stack symbol ($Z \in \Gamma$)
- δ is the state-transition function, $\delta : S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow S \times \Gamma^*$
- F is a possibly empty set of accepting states ($F \subset S$)

Pushdown automata can be seen as an extension of finite state machines with a stack (of arbitrary size) on which additional state can be kept. In every step, the pushdown automaton processes and input symbol and depending on the current state and the symbol on the stack, the machine transitions into a followup state and pushes new symbols on the stack.

Since pushdown automata extend finite state machines, pushdown automata can do everything a finite state machine can do. However, since pushdown automata have unlimited memory (the stack), they can solve problems that finite state machines cannot solve.

Pushdown Automaton Example ($a^n b^n$)

- The PDA $(\Sigma, S, s_0, \Gamma, Z, \delta)$ with $\Sigma = \{a, b\}$, $S = \{S0, S1, S2\}$, $s_0 = S0$, $\Gamma = \{A, Z\}$, $F = \{S2\}$, and

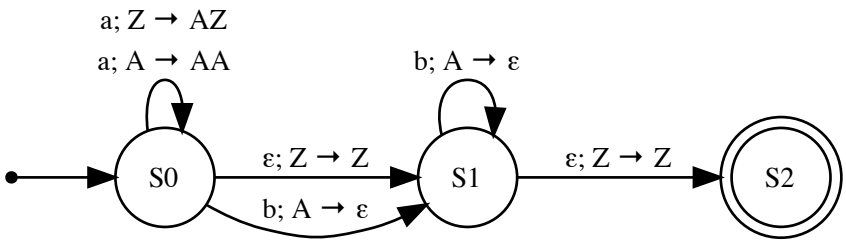
$$\begin{aligned}\delta = \{&(S0, a, Z, S0, AZ),\\ &(S0, a, A, S0, AA),\\ &(S0, \epsilon, Z, S1, Z),\\ &(S0, b, A, S1, \epsilon),\\ &(S1, b, A, S1, \epsilon),\\ &(S1, \epsilon, Z, S2, Z)\}\end{aligned}$$

  recognizes all words of the form $\{a^n b^n | n \geq 1\}$.

An implementation of this pushdown automata in Haskell:

```
1  data State = S0 | S1 | S2
2  data Stack = A | Z
3
4  accepts :: State -> [Stack] -> String -> Bool
5
6  accepts S0 (Z:ss) ('a':xs) = accepts S0 (A:Z:ss) xs
7  accepts S0 (A:ss) ('a':xs) = accepts S0 (A:A:ss) xs
8  accepts S0 (Z:ss) [] = accepts S1 (Z:ss) []
9  accepts S0 (A:ss) ('b':xs) = accepts S1 ss xs
10 accepts S1 (A:ss) ('b':xs) = accepts S1 ss xs
11 accepts S1 (Z:ss) [] = accepts S2 (Z:ss) []
12 accepts S2 _ _ = True
13 accepts _ _ _ = False
14
15 decide :: String -> Bool
16 decide = accepts S0 [Z]
```

PDA Example ($a^n b^n$) Represented as a Graph



- Pushdown automata can be represented as graphs where nodes (circles) represent states, arrows represent state transitions, an arrow pointing from a small black circle indicates the initial state, and accepting states are marked with with a double circle.

There is no common graphical notation for pushdown automata. We label transitions using the input symbol followed by a semicolon followed by the top of the stack symbol and an arrow pointing to the new symbols pushed on the stack.

The table below shows an execution for the string `aaabbb`. Lines are prefixed with the current state of the pushdown automaton and they show the stack growing to the left. Lines starting with a vertical bar indicate a transition and the input symbol processed.

```
S0: Z
| a
S0: ZA
| a
S0: ZAA
| a
S0: ZAAA
| b
S0: ZAA
| b
S1: ZA
| b
S1: Z
|
S2:
```

Section 28: Turing Machines

Turing Machine

Definition (turing machine)

A *Turing machine* (TM) is a 7-tuple $(\Sigma, S, s_0, \Gamma, b, \delta, F)$ where:

- Σ is the set of input symbols ($\Sigma \subseteq \Gamma \setminus \{b\}$)
- S is a finite, non-empty set of states
- s_0 is an initial state ($s_0 \in S$)
- Γ is a finite, non-empty set of tape alphabet symbols
- b is the blank symbol ($b \in \Gamma$)
- δ is the state-transition function, $\delta : (S \setminus F) \times \Gamma \to S \times \Gamma \times \{L, R\}$
- F is a set of accepting states ($F \subset S$)

- The symbol L indicates a left movement, the symbol R a right movement.

A turing machine in each step considers the current state and the current symbol on the tape and then it transitions into a followup state, writes a symbol on the tape, and moves the read/write head either left or right. A turing machine halts if no transition is defined for the current state and the current symbol on the tape. A turing machine also halts if it transitions into a state in F.

Turing Machine Example ($a^n b^n c^n$)

- The TM $(\Sigma, S, s_0, \Gamma, b, \delta, F)$ with $\Sigma = \{a, b, c\}$, $S = \{S0, S1, S2, S3, S4, S5\}$, $s_0 = S0$, $\Gamma = \{a, b, c, A, B, C, \_\}$, $b = \_$, $F = \{S5\}$, and

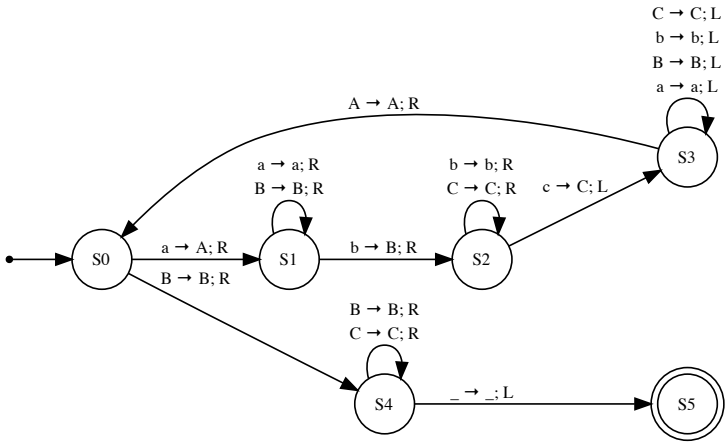
$$\begin{aligned}\delta = \{&(S0, a, S1, A, R), (S0, B, S4, B, R),\\ &(S1, b, S2, B, R), (S1, a, S1, a, R), (S1, B, S1, B, R),\\ &(S2, c, S3, C, L), (S2, b, S2, b, R), (S2, C, S2, C, R),\\ &(S3, A, S0, A, R), (S3, C, S3, C, L), (S3, b, S3, b, L),\\ &(S3, B, S3, B, L), (S3, a, S3, a, L),\\ &(S4, \_, S5, \_, L), (S4, B, S4, B, R), (S4, C, S4, C, R)\}\end{aligned}$$

  recognizes all words of the form $\{a^n b^n c^n | n \geq 1\}$.

An implementation of the Turing machine in Haskell:

```
1  import Prelude hiding (head)
2
3  data State = S0 | S1 | S2 | S3 | S4 | S5 deriving (Show)
4  data Tape = Tape String Int deriving (Show)
5
6  head :: Tape -> Char -> Bool
7  head (Tape xs i) c = xs !! i == c
8
9  left :: Tape -> Tape
10 left (Tape xs i)
11     | i == 0 = Tape ("_" ++ xs) 0
12     | otherwise = Tape xs (i - 1)
13
14 right :: Tape -> Tape
15 right (Tape xs i)
16     | i + 1 == length xs = Tape (xs ++ "_") (i + 1)
17     | otherwise = Tape xs (i + 1)
18
19 write :: Tape -> Char -> Tape
20 write (Tape xs i) c = Tape (take i xs ++ [c] ++ drop (i + 1) xs) i
21
22 accepts :: State -> Tape -> Bool
23 accepts S0 tape
24     | head tape 'a' = accepts S1 (right (write tape 'A'))
25     | head tape 'B' = accepts S4 (right (write tape 'B'))
26 accepts S1 tape
27     | head tape 'b' = accepts S2 (right (write tape 'B'))
28     | head tape 'a' = accepts S1 (right (write tape 'a'))
29     | head tape 'B' = accepts S1 (right (write tape 'B'))
30 accepts S2 tape
31     | head tape 'c' = accepts S3 (left  (write tape 'C'))
32     | head tape 'b' = accepts S2 (right (write tape 'b'))
33     | head tape 'C' = accepts S2 (right (write tape 'C'))
34 accepts S3 tape
35     | head tape 'A' = accepts S0 (right (write tape 'A'))
36     | head tape 'C' = accepts S3 (left  (write tape 'C'))
37     | head tape 'b' = accepts S3 (left  (write tape 'b'))
38     | head tape 'B' = accepts S3 (left  (write tape 'B'))
39     | head tape 'a' = accepts S3 (left  (write tape 'a'))
40 accepts S4 tape
41     | head tape '_' = accepts S5 (left  (write tape '_'))
42     | head tape 'B' = accepts S4 (right (write tape 'B'))
43     | head tape 'C' = accepts S4 (right (write tape 'C'))
44 accepts S5 tape = True
45 accepts _ _ = False
46
47 decide :: String -> Bool
48 decide xs = accepts S0 (Tape xs 0)
```

TM Example ($a^n b^n c^n$) Represented as a Graph



There is no common graphical notation for Turing machines. We label transitions using the symbol under the head, an arrow pointing to the symbol written under the head, and a semicolon followed by the movement indicator (L = left, R = right).

The Turing machine proceeds by replacing an a, a b, and a c with a capital letter and then moving back. This continues until all letters have been replaced and a final check is made that no input letters are left.

The table below shows an execution for the string `aabbcc`. A symbol in brackets indicates the head of the Turing machine and lines are prefixed with the current state of the Turing machine.

```
S0:  _ [a] a  b  b  c  c  _
S1:  _  A [a] b  b  c  c  _
S1:  _  A  a [b] b  c  c  _
S2:  _  A  a  B [b] c  c  _
S2:  _  A  a  B  b [c] c  _
S3:  _  A  a  B [b] C  c  _
S3:  _  A  a [B] b  C  c  _
S3:  _  A [a] B  b  C  c  _
S3:  _ [A] a  B  b  C  c  _
S0:  _  A [a] B  b  C  c  _
S1:  _  A  A [B] b  C  c  _
S1:  _  A  A  B [b] C  c  _
S2:  _  A  A  B  B [C] c  _
S2:  _  A  A  B  B  C [c] _
S3:  _  A  A  B  B [C] C  _
S3:  _  A  A  B [B] C  C  _
S3:  _  A  A [B] B  C  C  _
S3:  _  A [A] B  B  C  C  _
S0:  _  A  A [B] B  C  C  _
S4:  _  A  A  B [B] C  C  _
S4:  _  A  A  B  B [C] C  _
S4:  _  A  A  B  B  C [C] _
S4:  _  A  A  B  B  C  C [_]
S5:  _  A  A  B  B  C [C] _
```

Turing Machine Example (BB-3)

- The busy beaver game consists of designing a halting, binary-alphabet Turing machine, which writes the most 1s on the tape, using only a limited set of states.
- The TM $(\Sigma, S, s_0, \Gamma, b, \delta, F)$ with $\Sigma = \{1\}$, $S = \{S0, S1, S2, S3\}$, $s_0 = S0$, $\Gamma = \{0, 1\}$, $b = 0$, $F = \{S3\}$, and

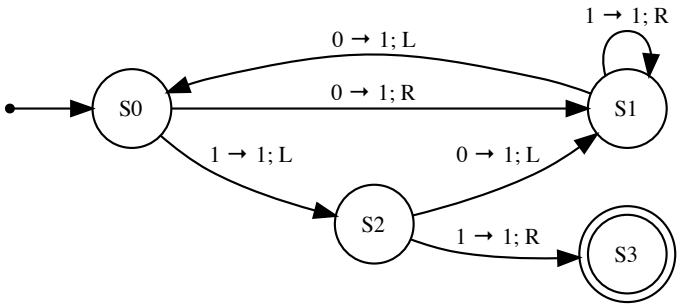
$$\begin{aligned}\delta = \{&(S0, 0, S1, 1, R), (S0, 1, S2, 1, L),\\ &(S1, 0, S0, 1, L), (S1, 1, S1, 1, R),\\ &(S2, 0, S1, 1, L), (S2, 1, S3, 1, R)\}\end{aligned}$$

  is a 3-state busy beaver (BB-3) Turing Machine (the halting state is not counted).

An implementation of a 3-state busy beaver in Haskell:

```
1  import Prelude hiding (head)
2
3  data State = S0 | S1 | S2 | S3 deriving (Show)
4  data Tape = Tape String Int deriving (Show)
5
6  head :: Tape -> Char -> Bool
7  head (Tape xs i) c = xs !! i == c
8
9  content :: Tape -> String
10 content (Tape xs _) = xs
11
12 left :: Tape -> Tape
13 left (Tape xs i)
14     | i == 0 = Tape ("0" ++ xs) 0
15     | otherwise = Tape xs (i - 1)
16
17 right :: Tape -> Tape
18 right (Tape xs i)
19     | i + 1 == length xs = Tape (xs ++ "0") (i + 1)
20     | otherwise = Tape xs (i + 1)
21
22 write :: Tape -> Char -> Tape
23 write (Tape xs i) c = Tape (take i xs ++ [c] ++ drop (i + 1) xs) i
24
25 delta :: State -> Tape -> Tape
26 delta S0 tape
27     | head tape '0' = delta S1 $ right $ write tape '1'
28     | head tape '1' = delta S2 $ left  $ write tape '1'
29 delta S1 tape
30     | head tape '0' = delta S0 $ left  $ write tape '1'
31     | head tape '1' = delta S1 $ right $ write tape '1'
32 delta S2 tape
33     | head tape '0' = delta S1 $ left  $ write tape '1'
34     | head tape '1' = delta S3 $ right $ write tape '1'
35 delta S3 tape = tape
36 delta state tape
37     = error ("tm failed in " ++ show state ++ " with: " ++ show tape)
38
39 run = content (delta S0 (Tape "0" 0))
```

TM Example (BB-3) Represented as a Graph



The table below shows an execution for the string `aabbcc`. A symbol in brackets indicates the head of the Turing machine and lines are prefixed with the current state of the Turing machine.

```
S0:  0  0  0  0 [0] 0  0  0
S1:  0  0  0  0  1 [0] 0  0
S0:  0  0  0  0 [1] 1  0  0
S2:  0  0  0 [0] 1  1  0  0
S1:  0  0 [0] 1  1  1  0  0
S0:  0 [0] 1  1  1  1  0  0
S1:  0  1 [1] 1  1  1  0  0
S1:  0  1  1 [1] 1  1  0  0
S1:  0  1  1  1 [1] 1  0  0
S1:  0  1  1  1  1 [1] 0  0
S1:  0  1  1  1  1  1 [0] 0
S0:  0  1  1  1  1 [1] 1  0
S2:  0  1  1  1 [1] 1  1  0
S2:  0  1  1  1  1 [1] 1  0
```

Section 29: Formal Languages

Formal Language and Formal Grammar

Definition (formal language)

Given an alphabet Σ, a *formal language* L is a subset of Σ^*, i.e., $\Sigma \subseteq L$. An element $w \in L$ is called a word of L.

Definition (formal grammar)

A *formal grammar* G is a tuple (N, Σ, P, S) where

- N is a finite set of non-terminal symbols (disjoint from Σ)
- Σ is a finite set of terminal symbols (disjoint from N)
- P is a finite set of production rules of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \mapsto (\Sigma \cup N)^*$
- $S \in N$ is a distinguished start symbol

Examples:

a) The grammar $G_1 = (N, \Sigma, P, S)$ with
   - $N = \{S, T\}$
   - $\Sigma = \{a, b\}$
   - start symbol S
   - $P = \{S \mapsto aTb, T \mapsto aT, T \mapsto Tb, T \mapsto \epsilon\}$

   defines the language $L(G_1) = \{a^n b^m | n \geq 1, m \geq 1\}$.

b) The grammar $G_2 = (N, \Sigma, P, S)$ with
   - $N = \{S\}$
   - $\Sigma = \{a, b\}$
   - start symbol S
   - $P = \{S \mapsto aSb, S \mapsto ab\}$

   defines the language $L(G_2) = \{a^n b^n | n \geq 1\}$.

c) The grammar $G_3 = (N, \Sigma, P, S)$ with
   - $N = \{S, B\}$
   - $\Sigma = \{a, b, c\}$
   - start symbol S
   - $P = \{S \mapsto aBSc, S \mapsto abc, Ba \mapsto aB, Bb \mapsto bb\}$

   defines the language $L(G_3) = \{a^n b^n c^n | n \geq 1\}$.

Formal Grammar Semantics

Definition (grammar derivation)

Given a formal grammar $G = (N, \Sigma, P, S)$, the binary relation $\underset{G}{\Rightarrow}$ (pronounced as "G derives in one step") on strings in $(\Sigma \cup N)^*$ is defined by

$$x \underset{G}{\Rightarrow} y \text{ iff } \exists u, v, p, q \in (\Sigma \cup N)^* : (x = upv) \wedge (p \mapsto q \in P) \wedge (y = uqv).$$

Let $\overset{*}{\underset{G}{\Rightarrow}}$ denote the reflexive transitive closure of $\underset{G}{\Rightarrow}$.

Definition (language of a grammar)

The language $L(G)$ of $G = (N, \Sigma, P, S)$ is defined as $\{w \in \Sigma^* | S \overset{*}{\underset{G}{\Rightarrow}} w\}$

Operations on Formal Languages

Definition (operations on languages)

Let L_1 and L_2 be formal languages. We defined the following operations:

- $L_1 \cup L_2 = \{w | w \in L_1 \vee w \in L_2\}$ (union)
- $L_1 \cap L_2 = \{w | w \in L_1 \wedge w \in L_2\}$ (intersection)
- $\bar{L_1} = \{w | w \notin L_1\}$ (complement)
- $L_1 L_2 = \{wz | w \in L_1 \wedge z \in L_2\}$ (concatenation)
- $L_1^* = \{\epsilon\} \cup \{wz | w \in L_1 \wedge z \in L_1^*\}$ (Kleene star)

Formal languages as defined so far are rather general. The focus of formal language theory is often on specific subsets of formal languages, so called classes of formal languages. Some widely known classes of formal languages are the following:

- Regular Languages (Type-3 Languages, L_3)
- Context-free Languages (Type-2 Languages, L_2)
- Context-sensitive Languages (Type-1 Languages, L_1)
- Recursively enumeratable Languages (Type-0 Languages, L_0)

These language classes form a hierarchy:

$$L_3 \subset L_2 \subset L_1 \subset L_0$$

This hierarchy is also known as the Chomsky Hierarchy (in reference to Noam Chomsky) and it was developed in the 1950s. Note that the classes of formal languages have different properties concerning the closure to the operations on formal languages.

Further information:

- http://en.wikipedia.org/wiki/Chomsky_hierarchy

Regular Languages

Definition (regular languages)

The collection of regular languages over an alphabet Σ is defined inductively as follows:

- The empty language $\emptyset$, and the empty string language $\{\epsilon\}$ are regular languages.
- For each $a \in \Sigma$, the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then
  - $(A \cup B)$ (union),
  - (AB) (concatenation), and
  - (A^*) (Kleene star)

  are regular languages.
- No other languages over Σ are regular.

In practice, we often use conventions that allow us to save parenthesis. For example, we may simply write abc or (abc) instead of $((ab)c)$ or $(a(bc))$.

The language $L = aa^*bb^* = ((a(a^*))(b(b^*)))$ is the set $\{a^n b^m | n \geq 1, m \geq 1\}$.

Expressions defining regular languages are often used to search for more complex patterns in strings. The pattern, defined by a regular expression, essentially defines a language and the search stops when a word belonging to the language has been found in a given input text. Regular expression understood by search tools often use a slightly different notation:

- The union $(A \cup B)$ is written as `(A|B)`
- The Kleene star (A^*) is written as `A*`
- An expression of the form $(A(A^*))$ is written as `A+`
- An expression of the form $(A \cup \epsilon)$ is written as `A?`
- Some regular expression languages allows repetitions of the form `A{n,m}` where A is matched at least n times but not more than m times
- Parenthesis can be used to group expressions but they can be left out if there is no ambiguity
- Sets of characters can be defined using bracket expressions, e.g, `[0123456789]` or `[0-9]` or `[[:digit:]]`

The Unix `grep` utility is well-known for using regular expressions for searching purposes. But most editors and text processing tools these days understand some form of regular expression syntax. (But be aware that there is no common regular expression syntax — there are actually quite many variations out there.)

Regular Grammars

Definition (right regular grammar)

A formal grammar (N, Σ, P, S) is called a *right regular grammar* iff all the production rules in P are of one of the forms $A \mapsto a$, $A \mapsto aB$, or $A \mapsto \epsilon$ with $A, B \in N$, $a \in \Sigma$, and ϵ denoting the empty word.

Definition (left regular grammar)

A formal grammar (N, Σ, P, S) is called a *left regular grammar* iff all the production rules in P are of one of the forms $A \mapsto a$, $A \mapsto Ba$, or $A \mapsto \epsilon$ with $A, B \in N$, $a \in \Sigma$, and ϵ denoting the empty word.

Definition (regular grammar)

A *regular grammar* is a left or right regular grammar.

The language $L = aa^*bb^*$ can be generated by the (right) regular grammar $G_r = (N, \Sigma, P_r, S)$ with

- $N = \{S, T, U\}$
- $\Sigma = \{a, b\}$
- start symbol S
- $P_r = \{S \mapsto aT, T \mapsto aT, T \mapsto bU, U \mapsto bU, U \mapsto \epsilon\}$.

The language L can also be generated by the (left) regular grammar $G_l = (N, \Sigma, P_l, S)$ with

- $N = \{S, T, U\}$
- $\Sigma = \{a, b\}$
- start symbol S
- $P_l = \{S \mapsto Tb, T \mapsto Tb, T \mapsto Ua, U \mapsto Ua, U \mapsto \epsilon\}$.

Properties of Regular Languages

- A regular language can be defined by a regular expression.
- A regular language can be accepted by a finite state machine.
- A regular language can be generated by a regular grammar.
- ...

$\Rightarrow$ For more properties of regular languages and proofs of the properties, see the course "Formal Languages and Logic".

Context-Free Languages

Definition (context-free grammar)

A formal grammar (N, Σ, P, S) is called a *context-free grammar* iff all productions P are of the form $N \mapsto (\Sigma \cup N)^*$.

Definition (context-free language)

A *context-free language* is a formal language generated by a context-free grammar.

Context-free languages play an important role in computer science.

- The syntax of many programming languages is defined by context-free grammars.
- Many Internet message formats are defined by context-free grammars.
- Context-free grammars are often used to describe expressions.

Example: The context-free grammar $G_2 = (N, \Sigma, P, S)$ with

- $N = \{S\}$
- $\Sigma = \{a, b\}$
- start symbol S
- $P = \{S \mapsto aSb, S \mapsto ab\}$

defines the language $L(G_2) = \{a^n b^n | n \geq 1\}$.

Properties of Context-Free Languages

- A context-free language can be accepted by a pushdown automata.
- A context-free language can be generated by a context-free grammar.
- The set of context-free languages includes the set of regular languages.
- There are context-free languages that are not regular languages.
- ...

$\Rightarrow$ For more properties of context-free languages and proofs of the properties, see the course "Formal Languages and Logic".

Context-Sensitive Languages

Definition (context-sensitive grammar)

A formal grammar (N, Σ, P, S) is called a *context-sensitive grammar* iff all productions P are of the form $\alpha N \beta \mapsto \alpha\gamma\beta$ with $\alpha, \beta \in (\Sigma \cup N)^*$ and $\gamma \in (\Sigma \cup N)^+$

Definition (context-sensitive language)

A *context-sensitive language* is a formal language generated by a context-sensitive grammar.

Example: The context-sensitive grammar $G_3 = (N, \Sigma, P, S)$ with

- $N = \{S, B\}$
- $\Sigma = \{a, b, c\}$
- start symbol S
- $P = \{S \mapsto aBSc, S \mapsto abc, Ba \mapsto aB, Bb \mapsto bb\}$

defines the language $L(G_3) = \{a^n b^n c^n | n \geq 1\}$.

Properties of Context-Sensitive Languages

- A context-sensitive language can be accepted by a Turing machine.
- A context-sensitive language can be generated by a context-sensitive grammar.
- The set of context-sensitive languages includes the set of context-free languages.
- There are context-sensitive languages that are not context-free languages.
- ...

$\Rightarrow$ For more properties of context-sensitive languages and proofs of the properties, see the course "Formal Languages and Logic".

A restricted form of a Turing machine, called a linear bounded automaton, is sufficient to accept context-sensitive languages. A linear bounded automaton is essentially a Turing machine where the tape is limited to a finite contiguous portion of the tape, whose length is a linear function of the length of the initial input.

Turing machines can accept any recursively enumerable formal language, which is a proper superset of all context-sensitive languages. But then there are also non-recursively enumerable languages that not even a Turing machine can accept.

Part VII

Computability and Computational Complexity

Computability theory focuses on the following key questions:

- What does it mean for a function to be computable or noncomputable?
- How can noncomputable functions be classified into a hierarchy based on their level of noncomputability?

Computational complexity theory focuses on the following key question:

- How to classify computational problems according to their inherent difficulty?

Both theories require models of computation that one can work with in a mathematical sense, i.e.,

- models that are simple enough to reason with them formally and
- general enough that the theoretical results apply to real computers.

Abstract automate such as the Turing machine are often used in computability and computational complexity in order to derive results that apply to all Turing equivalent computing machines.

Section 30: Landau Sets and Big O Notation

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:

- $O(g) = \{f | \exists k. f \leq_a k \cdot g\}$
- $\Omega(g) = \{f | \exists k. k \cdot g \leq_a f\}$
- $\Theta(g) = O(g) \cap \Omega(g)$

Commonly Used Landau Sets

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

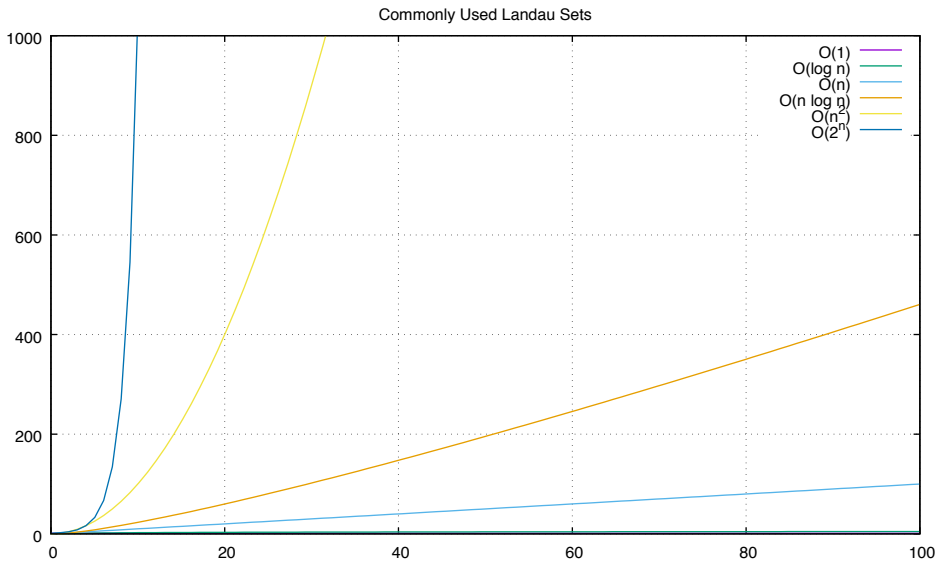
Landau Set	class name	rank
$O(n^2)$	quadratic	5
$O(n^k)$	polynomial	6
$O(k^n)$	exponential	7

Theorem (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

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

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



Landau Set Rules

Theorem (Landau Set Computation Rules)

We have the following computation rules for Landau sets:

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

Examples:

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

Big O Notation (Usage)

- 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.
- Use O classes for worst case complexity, use Ω classes for best case complexity.

Section 31: Computability

Turing Completeness and Equivalence

Definition (Turing complete)

A system of data-manipulation rules (such as a computer's instruction set, a programming language, or a cellular automaton) is said to be *Turing complete* or computationally universal if it can be used to simulate any Turing machine.

Definition (Turing equivalent)

Two computers P and Q are called *Turing equivalent* if P can simulate Q and Q can simulate P.

Church-Turing Thesis

Definition (Church-Turing thesis)

The *Church–Turing thesis* states that a function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

- The Church-Turing thesis is a conjecture.

For further information:

- https://en.wikipedia.org/wiki/Church\OT1\textendashTuring_thesis

Halting Problem

Definition (halting problem)

The halting problem is a decision problem: Given an arbitrary program and an input to the program, decide whether the program will eventually halt when run with that input.

- It is impossible to decide the halting problem. The proof uses a diagonalization argument. Here is an outline of the basic idea...
  1. Assume that the halting problem for any program can be solved by a machine H.
  2. Using H, construct a machine G that goes into an endless loop if H determines that an algorithm halts.
  3. Feed the program G as input to G:
     - If G halts, then H decided that G does not halt, which is a contradiction.
     - If G does not halt, then H decided the G does halt, which is a contradiction.

For further information:

- https://en.wikipedia.org/wiki/Halting_problem

Section 32: Computational Complexity