Introduction to Computer Science

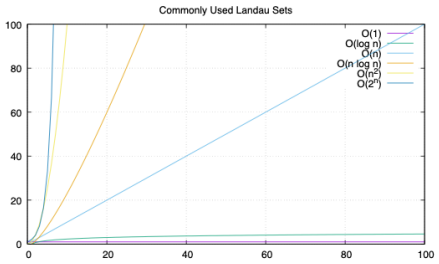
Lecture Notes

Jürgen Schönwälder

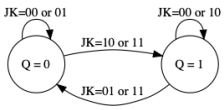
December 6, 2023

Abstract

This memo provides annotated slides for the Computer Science module "Introduction to Computer Science" offered at the Constructor University (formerly Jacobs University Bremen). 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, and Mike Gordon's "Background reading on Hoare Logic".



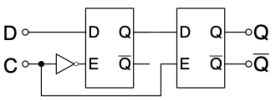
$$\{P\}\ C\ \{Q\}$$

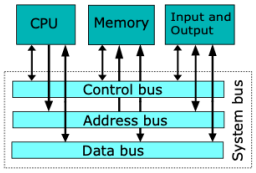



```
module Collatz (collatz, collatzSequence) where

collatz :: Int -> Int
collatz n
    | odd n  = 3 * n + 1
    | even n = n `div` 2

collatzSequence :: Int -> [Int]
collatzSequence n
    | n < 1     = []
    | n == 1    = [1]
    | otherwise = n : collatzSequence (collatz n)
```





property	eq	po	spo	definition	=	≤	<
reflexive	✓	✓		$a \sim a$	✓	✓	
irreflexive			✓	$a \not\sim a$			✓
symmetric	✓			$a \sim b \Rightarrow b \sim a$	✓		
asymmetric			✓	$a \sim b \Rightarrow b \not\sim a$			✓
antisymmetric		✓		$a \sim b \wedge b \sim a \Rightarrow a = b$		✓	
transitive	✓	✓	✓	$a \sim b \wedge b \sim c \Rightarrow a \sim c$	✓	✓	✓

eq = equivalence relation, po = partial order, spo = strict partial order

P1 $0 \in \mathbb{N}$

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

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

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

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

$$\tau = (\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi = \neg(\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi)$$

$$\tau' = \neg((\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi) = (\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi)$$

CONSTRUCTOR UNIVERSITY

https://cnds.jacobs-university.de/courses/ics-2023

Table of Contents

Part I

Algorithms

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

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

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

Section 1: Computer Science and Algorithms

Computer Science

- Computer science the study of computers and how they can be used.
  [Oxford Learner's Dictionary, August 2020]
- Computer science is a branch of science that deals with the theory of computation or the design of computers.
  [Merriam Webster, August 2020]
- Computer science is the study of computation and information. Computer science deals with the theory of computation, algorithms, computational problems and the design of computer systems hardware, software and applications.
  [Wikipedia, August 2020]
- Computer science is the study of computers, including both hardware and software design.
  [Webopedia, August 2020]

James W. McGuffee presents different definitions of Computer Science, and he investigates how they are perceived by students [17]. Peter J. Denning et al. discuss misconceptions about Computer Science [2].

Perhaps the most classic and most comprehensive book series presenting core algorithms and concepts of Computer Science is the series "The Art of Computer Programming' by Donald E. Knuth [9, 12, 10, 11].

Further online information:

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

Algorithm

Definition (algorithm)

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

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

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

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

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

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

Further online information:

- **Wikipedia**: Algorithm

Algorithmic Thinking

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

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

We will train you in algorithmic thinking [4]. 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.

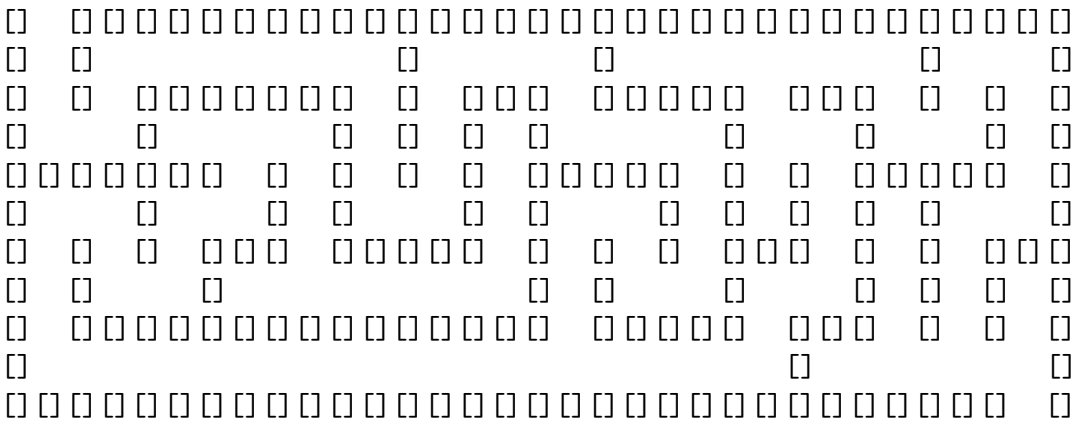
Example: Consider the following problem: Implement a function that returns the square root of a number (on a system that does not have a math library). At first sight, this looks like a reasonably clear definition of the problem. However, on second thought, we discover a number of questions that need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

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.

Further online information:

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

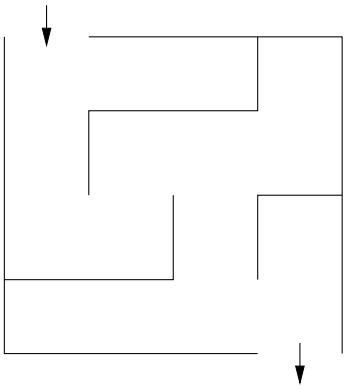
Problem Statement

Problem:

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

Questions:

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



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

What still too often happens is that the 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. . .

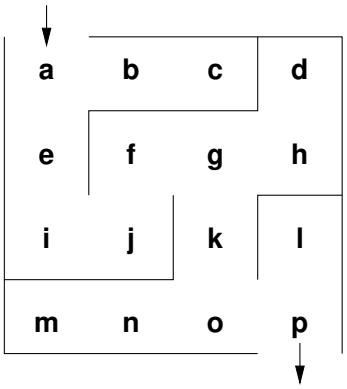
While hacking can be fun, it is often far more effective to think first before opening the editor and starting to write code. It often also helps to discuss the problem with others. Even coding together in pairs of two (pair programming) has been found to lead to better programs. Despite common believes,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

Further online information:

- **Wikipedia**: Pair Programming

Problem Formalization (1/3)

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



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

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

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

And finally, what about mazes that are not rectangular?

Problem Formalization (2/3)

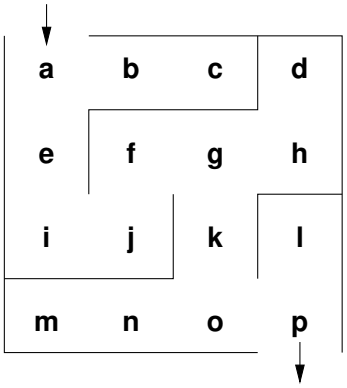
Lets try to formalize the problem in mathematical terms:

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

In the example, we have

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

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



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

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

Further online information:

- **Wikipedia**: Graph

Why use a mathematical formalization?

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

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

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

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

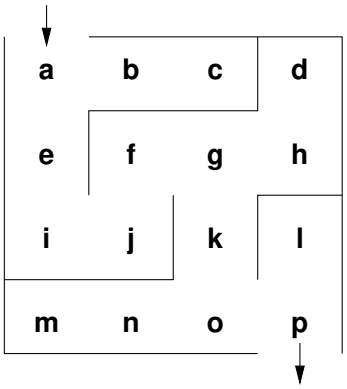
Problem Formalization (3/3)

Definition:

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

Interpretation:

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



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

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

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

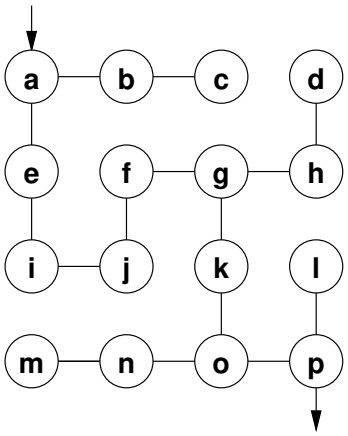
The maze M is constructed on top of a topology graph $G = (V, A)$ representing the adjacency of the rooms:

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

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

Mazes as Graphs (Visualization via Diagrams)

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



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

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

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

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

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

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

Further online information:

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

Mazes as Graphs (Good Mazes)

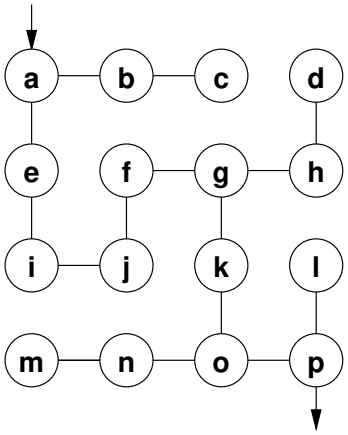
Recall, what is a good maze?

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

Solution:

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

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



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

Note that in graph theory, a spanning tree T 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.

Further online information:

- **Wikipedia**: Spanning Tree

Kruskal's Algorithm (1/2)

General approach:

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

Questions:

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

Kruskal's algorithm was published in 1956 [13] and has since then become one of the standard textbook algorithms for generating spanning trees. Kruskal's algorithm can be used to create minimum spanning trees if the edges of the graph are weighted and in each step an edge with the smallest weight is selected and added to the graph.

Further online information:

- **Wikipedia**: Kruskal's Algorithm

Kruskal's Algorithm (2/2)

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

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

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 by the end of the semester.)

Further online information:

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

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

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

module Maze (maze) where

import Data.List
import Graph

type Class = [Node]

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

-- Take a graph, a list of node equivalence classes, a new (spanning
-- tree) graph (which we are building up) and return a spanning tree
-- graph if only one equivalence is left. Otherwise, find an edge and
-- continue building the spanning tree by merging the classes
-- connected by the edge and adding the edge to the spanning tree.
buildMaze :: Graph -> [Class] -> Graph -> Graph
buildMaze g cs (ns, es)
  | length cs == 1 = (ns, es)
  | otherwise      = buildMaze g (mergeClasses cs e) (ns, e:es)
  where e = findEdge cs (snd g)

-- Given a list of classes and edges, find an edge that connects two
-- equivalence classes. We pick the first edge if there are multiple
-- but this could also be a random selection. The distinct helper
-- function tests where an edge connects two equivalence classes.
findEdge :: [Class] -> [Edge] -> Edge
findEdge cs es = head (filter (distinct cs) es)
  where distinct cs (a, b) = filter (elem a) cs /= filter (elem b) cs

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

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

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

Further online information:

- **Wikipedia**: Maze Generation Algorithm

Section 3: String Search Algorithms

Problem Statement

Problem:

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

Questions:

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

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

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

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

Further online information:

- **Wikipedia**: String Searching Algorithm

Problem Formalization

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

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

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

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

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

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

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

Naive String Search

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

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

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 the text.

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

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

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

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

    return NULL;
}
```

Listing 2: Naive string search implemented in C

Naive String Search Performance

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

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

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

A Haskell implementation of naive string search is shown in Listing 3.

```haskell
{- |
   Module: naive-search/Search.hs

Search for a substring in a text using the naive search algorithm.
-}
module Search (findFirstIn, isPrefixOf) where

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

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

```

Listing 3: Naive string search implemented in Haskell

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

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

- Example: t = FINDANEEDLEINAHAYSTACK, p = HAY

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

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 the unmatched character and the position in the pattern to the number of alignments that can be skipped.

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

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

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

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

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

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

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. We have matched the suffix `NEED` and we have a mismatch of `U` against `A`. Since the pattern contains this suffix again left to the current comparison position, we move the pattern right to align with this suffix.

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

- Example: t = FINDANEEDLEINAHAYSTACK, p = EDISUNEED

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

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. We have matched the suffix NEED and we have a mismatch of U against A. The prefix of the pattern contains the suffix ED of our suffix NEED and hence we move to align this prefix with the matched suffix of our suffix.

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

- Example: t = FINDANEEDLEINAHAYSTACK, p = FOODINEED

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

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. We neither have a matching suffix nor does the prefix match a suffix of the suffix. Hence we skip alignments until the end of the current alignment.

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

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

Boyer-Moore Rules Combined

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

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

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

The Boyer-Moore algorithm was introduced in 1977 [1] and the improvement by Gali for worst cases in 1979 in [5].

Further online information:

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

Section 4: Complexity, Correctness, Engineering

Complexity of Algorithms

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

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

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

Performance and Scaling

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

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

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

Big O Notation (Landau Notation)

Definition (asymptotically bounded)

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

Definition (Landau Sets)

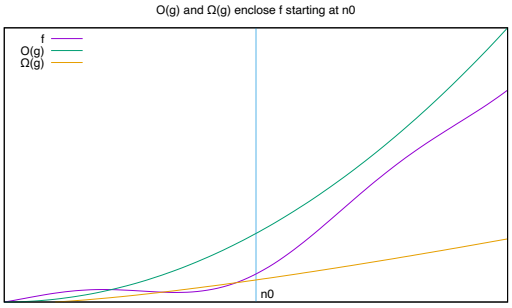
The three *Landau Sets* $O(g), \Omega(g), \Theta(g)$ are defined as follows:

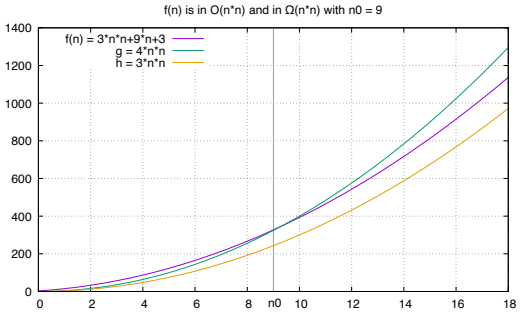
- $O(g) = \{ f \mid \exists k \in \mathbb{N}. f \leq_a k \cdot g \}$
- $\Omega(g) = \{ f \mid \exists k \in \mathbb{N}. k \cdot g \leq_a f \}$
- $\Theta(g) = O(g) \cap \Omega(g)$

Interpretation of the three Landau Sets:

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

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





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

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

Commonly Used Landau Sets

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

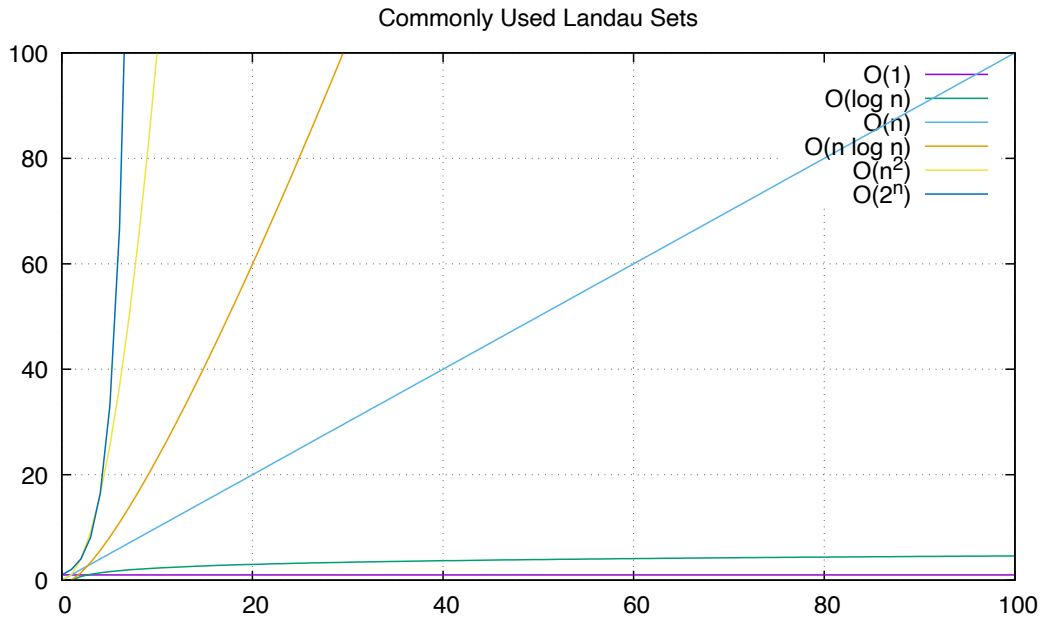
Landau Set	class name	rank
$O(n^2)$	quadratic	5
$O(n^k)$	polynomial	6
$O(k^n)$	exponential	7

Theorem (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

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

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



Landau Set Rules

Theorem (Landau Set Computation Rules)

We have the following computation rules for Landau sets:

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

Examples:

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

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

Correctness of Algorithms and Programs

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

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

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

Partial Correctness and Total Correctness

Definition (partial correctness)

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

Definition (total correctness)

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

In order to talk about the correctness of an algorithm, we need a specification that clearly states the precondition P and the postcondition Q. In other words, an algorithm 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 execution always passing through the same sequence of states.

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

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

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

Randomized Algorithms

Definition (randomized algorithm)

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

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

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

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

Engineering of Software

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

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

Further online information:

- **Wikipedia**: Software Engineering

Part II

Discrete Mathematics

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

Recommended background reading are the first five chapters of the book “Mathematics for Computer Science” by Eric Lehmann, F. Thomson Leighton, and Albert R. Meyer [15]. Students interested to dive deeper into proof methods may want to study Richard Hammack’s “Book of Proof” [6]. Both books are available online and in print.

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

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

Section 5: 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)
- "This sentence is false." (a paradox)

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

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

Axioms

Definition (axiom)

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

Definition (Peano axioms for natural numbers)

P1 0 is a natural number.

P2 Every natural number has a successor.

P3 0 is not the successor of any natural number.

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

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

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

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

Theorems, Lemmata, Corollaries

Definition (theorem, lemma, corollary)

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

Definition (conjecture)

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

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

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

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

for some integer $n > 2$.

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

Predicates

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

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

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

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

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

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

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

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

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

Mathematical Notation

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

Some examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

Greek Letters

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

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

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

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

Mathematical Proof

Definition (mathematical proof)

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

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

Hints for Writing Proofs

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

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

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

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

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

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

Prove an Implication by its Contrapositive

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

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

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

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

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

Prove an "if and only if" by two Implications

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

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

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

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

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

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

Next, we show that if x^2 is even, then x is even. Since x^2 is even, we can write it as $x^2 = 2k$ for some suitable number k, i.e., 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 single chain of implications, namely that $A \Rightarrow B$ and $B \Rightarrow C$ and $C \Rightarrow A$. It is not necessary to prove $B \Rightarrow A$ since this follows from $B \Rightarrow C \Rightarrow A$. Similarly, it is not necessary to prove $C \Rightarrow B$ since this follows form $C \Rightarrow A \Rightarrow B$.

Prove an "if and only if" by a Chain of "if and only if"s

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

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

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

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

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

Breaking a Proof into Cases

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

  Proof. We prove P by considering the cases $c_1, \ldots, c_N$.
  Case 1: Suppose c_1. Prove of P for c_1.
  ...
  Case N: Suppose c_N. Prove of P for c_N.
  Since P holds for all cases $c_1, \ldots c_N$, the statement P holds. □

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

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

- Case 1: Suppose n is even:

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

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

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

- Case 2: Suppose n is odd:

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

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

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

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

Proof by Contradiction

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

  Proof. We prove P by contradiction.

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

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

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

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

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

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

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

Proof by Induction

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

  Proof. We prove P by induction.

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

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

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

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

- Base case:

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

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

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

- Induction step:

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

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

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

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

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

Summary of Proof Techniques

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

Section 6: Sets

Sets

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

Some popular sets in mathematics:

symbol	set	elements
$\emptyset$	empty set	$\{\}$
$\mathbb{N}$	nonnegative integers	$\{0, 1, 2, 3, \dots\}$
$\mathbb{Z}$	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 $R = \{ x \mid x \notin x \}$, then $R \in R \Leftrightarrow R \notin R$.

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

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

Basic Relations between Sets

Definition (basic relations between sets)

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

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

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

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

Operations on Sets 1/2

Definition (set union)

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

Definition (set intersection)

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

Definition (set difference)

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

Some basic properties of set unions:

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

Some basic properties of set intersections:

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

Some basic properties of set differences:

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

Some basic properties of combinations of set operations:

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

Operations on Sets 2/2

Definition (power set)

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

Definition (cartesian product)

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

Cardinality of Sets

Definition (cardinality)

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

Definition (countably infinite)

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

Definition (countable)

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

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

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

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

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

Section 7: Relations

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 focus on binary relations.

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

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

- Entities:
  - $Student$
  - $Instructor$
  - $Course$
  - $Module$
  - $Person$
  - . . .
- 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 often defined using a graphical notation. A standard graphical notation is part of the Unified Modeling Language (UML).

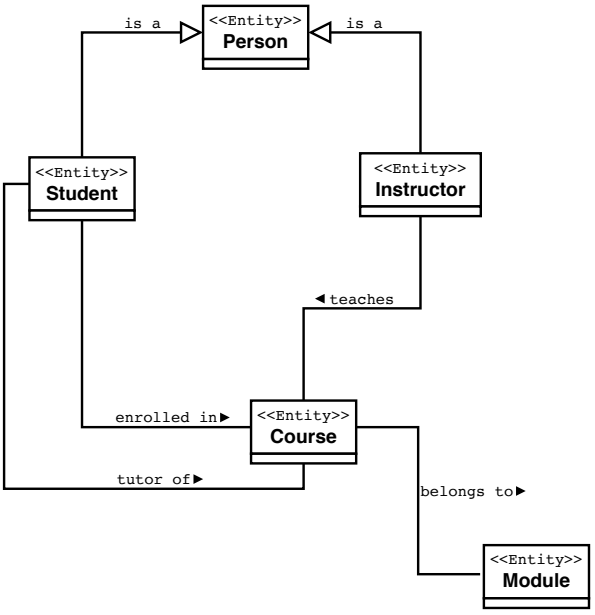


Figure 1: A UML diagram showing relations using a graphical notation

Figure 1 provides a UML diagram representation of relations in a campus information system. The symbols used in UML diagrams have well defined semantics. We do not go into the details here. For now, just note that special line ends (unfilled arrow heads) are used for is_a relations. The reason is that is_a relations are quite common and programming languages or database systems often have special support for representing is_a relations.

Binary Relations

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

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

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

We sometimes use the notation $dom(R)$ 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 graphviz dot notation:

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

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

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

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

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

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

Properties of Binary Relations (Endorelations)

Definition

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

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

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

Examples:

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

Equivalence, Partial Order, and Strict Partial Order

Definition (equivalence relation)

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

Definition (partial order and strict partial order)

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

Definition (linear order)

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

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

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

Examples:

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

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

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

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

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

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

Summary of Properties of Binary Relations

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

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

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

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

Section 8: Functions

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 graphviz dot notation:

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

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

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

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

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

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

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

Operations on Functions

Definition (function composition)

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

Definition (function restriction)

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

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' :: Integral a => a -> Bool
2 odd' = not . even
```

```
1 sumOfOdds :: Integral a => [a] -> a
2 sumOfOdds = sum . filter (not . even)
```

Lambda Notation of Functions

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

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

- Identity function for natural numbers:

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

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

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

- Addition of natural numbers:

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

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

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

Further online information:

- **Wikipedia**: Lambda Calculus

Currying

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

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

Section 9: Algebraic Structures

Algebraic Structures

Definition (algebraic structure)

An *algebraic structure* consists of a nonempty set S (called the underlying set, carrier set or domain), a collection of operations on S (typically binary operations such as addition and multiplication), and a finite set of axioms that these operations must satisfy.

- A branch of mathematics known as *universal algebra* studies algebraic structures.
- A set S is a degenerate algebraic structure having no operations.
- We are interested in algebraic structures with one or multiple defined operations.

Universal algebra enables us to prove theorems for several similar algebraic structures very efficiently. As we will see, basic set theory and boolean logic have many similarities. By identifying an algebraic structure underlying both basic set theory and boolean logic, we can define and prove theorems once that apply to both basic set theory and boolean logic.

Magma

Definition (magma)

A *magma* $(S, *)$ is an algebraic structure consisting of a set S together with a binary operation $*$ satisfying the following property:

$$\forall a, b \in S : a * b \in S \qquad \text{closure}$$

The operation $*$ is a function $* : S \times S \to S$.

- A magma is a very general algebraic structure.
- By imposing additional constraints on the operation $*$, we can define more useful classes of algebraic structures.

Some examples for magmas:

1. $(\mathbb{N}, +)$ (addition over natural numbers)
2. $(\mathbb{N}, \cdot)$ (multiplication over natural numbers)

A more interesting magma is the operation $(a, b, c) * (x, y, z) = (bz - cy, cx - az, ay - bx)$ defined over the set $\mathbb{R}^3$. This magma is interesting since it is a "true" magma, i.e., it is not one of the algebraic structures we define next.

In Haskell, a magma can be represented by the typeclass `Magma`:

```
class Magma a where
  (<>) :: a -> a -> a
```

The `Magma` typeclass is defined in `Data.Magma`. Here is an example implementation of the type class:

```
import Data.Magma

data Shape = Rock | Paper | Scissors deriving (Eq, Show)

instance Magma Shape where
    (<>) :: Shape -> Shape -> Shape
    Rock     <> Rock     = Rock
    Rock     <> Paper    = Paper
    Rock     <> Scissors = Rock
    Paper    <> Paper    = Paper
    Paper    <> Scissors = Scissors
    Paper    <> Rock     = Paper
    Scissors <> Scissors = Scissors
    Scissors <> Rock     = Rock
    Scissors <> Paper    = Scissors
```

Semigroup

Definition (semigroup)

A *semigroup* $(S, *)$ is an algebraic structure consisting of a set S together with a binary operation $* : S \times S \to S$ satisfying the following property:

$$\forall a, b, c \in S : (a * b) * c = a * (b * c) \qquad \text{associativity}$$

- A semigroup extends a magma by requiring that the operation is associative.
- The set of semigroups is a true subset of the set of all magmas.

Some examples for semigroups:

1. $(\mathbb{N}, +)$ (addition over natural numbers)
2. $(\mathbb{N}, \cdot)$ (multiplication over natural numbers)
3. $(\mathbb{Z}, +)$ (addition over integer numbers)
4. $(\mathbb{Z}, \cdot)$ (multiplication over integer numbers)
5. $(\mathbb{Q}, +)$ (addition over rational numbers)
6. $(\mathbb{Q}, \cdot)$ (multiplication over rational numbers)
7. $(\mathbb{R}, +)$ (addition over real numbers)
8. $(\mathbb{R}, \cdot)$ (multiplication over real numbers)

Monoid

Definition (monoid)

A *monoid* $(S, *, e)$ is an algebraic structure consisting of a set S together with a binary operation $* : S \times S \to S$ and an identity element e satisfying the following properties:

$$\forall a, b, c \in S : (a * b) * c = a * (b * c) \qquad \text{associativity}$$
$$\exists e \in S, \forall a \in S : e * a = a = a * e \qquad \text{identity element}$$

The identity element e is also called the neutral element.

- A monoid extends a semigroups by requiring that there is an identity element.
- The set of monoids is a true subset of the set of all semigroups.

Some examples of monoids:

1. $(\mathbb{N}_0, +)$ (addition over natural numbers, identity 0)
2. $(\mathbb{R}, \cdot)$ (multiplication over real numbers, identity 1)
3. $(\mathbb{Z}, \cdot)$ (multiplication over integer numbers, identity 1)

Group

Definition (group)

A *group* $(S, *, e)$ is an algebraic structure consisting of a set S together with a binary operation $* : S \times S \to S$ and an identity element e satisfying the following properties:

$$\forall a, b, c \in S : (a * b) * c = a * (b * c) \quad \text{associativity}$$
$$\exists e \in S, \forall a \in S : e * a = a = a * e \quad \text{identity element}$$
$$\forall a \in S, \exists b \in S : a * b = e \quad \text{inverse element}$$

The element b is called the inverse element of a; it is often denoted as a^{-1}.

- A group extends a monoid by requiring that there is an inverse element.
- The set of groups is a true subset of the set of all monoids.

Some examples of groups:

1. $(\mathbb{Z}, +)$ (addition over integer numbers, identity 0)
2. $(\mathbb{Q}, \cdot)$ (multiplication over rational numbers, identity 1)

Note that $(\mathbb{Z}, \cdot)$ is not a group but a monoid.

Abelian Group

Definition (abelian group)

An *abelian group* $(S, *, e)$ is an algebraic structure consisting of a set S together with a binary operation $* : S \times S \to S$ and an identity element e satisfying the following properties:

$$\forall a, b, c \in S : (a * b) * c = a * (b * c) \quad \text{associativity}$$
$$\exists e \in S, \forall a \in S : e * a = a = a * e \quad \text{identity element}$$
$$\forall a \in S, \exists b \in S : a * b = e \quad \text{inverse element}$$
$$\forall a, b \in S : a * b = b * a \quad \text{commutativity}$$

- An abelian group extends a group by requiring that the operation is commutativ.
- The set of abelian groups is a true subset of the set of all groups.

Some examples of abelian groups:

1. $(\mathbb{Z}, +)$ (addition over integer numbers, identity 0)
2. $(\mathbb{Q}, \cdot)$ (multiplication over rational numbers, identity 1)

Ring

Definition

A *ring* $(S, +, \cdot, 0, 1)$ is an algebraic structure consisting of a set S together with two binary operations $+$ (addition) and $\cdot$ (multiplication) satisfying the following properties:

1. $(S, +, 0)$ is an abelian group with the neutral element 0.
2. $(S, \cdot, 1)$ is a monoid with the neutral element 1.
3. Multiplication is distributive with respect to addition:

$$\forall a, b, c \in S : a \cdot (b + c) = (a \cdot b) + (a \cdot c) \qquad \text{left distributivity}$$
$$\forall a, b, c \in S : (b + c) \cdot a = (b \cdot a) + (c \cdot a) \qquad \text{right distributivity}$$

The operations are called $+$ and $\cdot$ and the neutral elements 0 and 1 since this makes it easier to think about rings. The names are, however, placeholders. The set S can have a complex structure and hence the neutral elements will also have a complex structure. In other words, the number 0 and the neutral element 0 are two very separate concepts and the context tells you what is meant in a concrete situation. The same holds for the number 1 and the neutral element 1.

Note that some authors define a ring differently, i.e., they only require $(S, \cdot)$ to be a semigroup.

Some examples of rings:

1. The algebraic structure $(\mathbb{Z}, +, \cdot, 0, 1)$ is a ring.
2. The finite sets of integer numbers $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ are rings for addition modulo n and multiplication modulo n.
3. The set of all 2×2 matrices is a ring for matrix addition and matrix multiplication operations. The zero elements are:

$$0 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \qquad 1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Field

Definition (field)

A *field* $(S, +, \cdot, 0, 1)$ is an algebraic structure consisting of a set S together with two operations $+ : S \times S \to S$ and $\cdot : S \times S \to S$ satisfying the following properties:

1. $(S, +, 0)$ is a group with the neutral element 0
2. $(S \setminus \{0\}, \cdot, 1)$ is a group with the neutral element 1
3. Multiplication distributes over addition

- A field is a commutative ring where $0 \neq 1$ and all nonzero elements are invertible under multiplication.
- Well known fields are the field of rational numbers, the field of real numbers, or the field of complex numbers.

Like before, operations $+$ and $\cdot$ and the neutral elements 0 and 1 are placeholders.

Some examples of fields:

1. $(\mathbb{Q}, +, \cdot, 0, 1)$ (addition and multiplication over rational numbers)
2. $(\mathbb{R}, +, \cdot, 0, 1)$ (addition and multiplication over real numbers)
3. $(\mathbb{Z}_p, +, \cdot, 0, 1)$ (addition and multiplication modulo p over $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ with p prime)

Homomorphism, Monomorphism, Epimorphism, Isomorphism

Definition (homomorphism)

A *homomorphism* is a structure-preserving map between two algebraic structures of the same type that preserves the operations of the structures.

Definition (monomorphism)

An injective homomorphism is called a *monomorphism*.

Definition (epimorphism)

A surjective homomorphism is called an *epimorphism*.

Definition (isomorphism)

A bijective homomorphism is called an *isomorphism*.

Given two groups, $(G, *)$ and $(H, \star)$, a *group homomorphism* from $(G, *)$ to $(H, \star)$ is a mapping $h : G \to H$ such that for all u and v in G it holds that

$$h(u * v) = h(u) \star h(v)$$

where the group operation on the left side of the equation is that of G and on the right side that of H. The group homomorphism h maps the identity element e_G of G to the identity element e_H of H and it also maps the inverses of G to inverses of H:

$$h(u^{-1}) = (h(u))^{-1}$$

Example: Given the groups $(\mathbb{R}, +)$ and $(\mathbb{R}, \cdot)$, the map $h : \mathbb{R} \to \mathbb{R}$ with $x \mapsto e^x$ is a group homomorphism.

$$\begin{aligned} h(u + v) &= e^{(u+v)} \\ &= e^u \cdot e^v \\ &= h(u) \cdot h(v) \end{aligned}$$

Note that h maps the identity element 0 of $(\mathbb{R}, +)$ to the identity element 1 of $(\mathbb{R}, \cdot)$ and that it also maps inverses:

$$\begin{aligned} h(u^{-1}) &= h(-u) \\ &= e^{-u} \\ &= (e^u)^{-1} \\ &= (h(u))^{-1} \end{aligned}$$

Endomorphism, Automorphism

Definition (endomorphism)

A homomorphism where the domain equals the codomain is called an *endomorphism*.

Definition (automorphism)

An endomorphism which is also an isomorphism is called an *automorphism*.

Part III

Data Representation

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

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

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

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

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

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

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

Numbers can be confusing. . .

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

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

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

Further online information:

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

Number Systems in Mathematics

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

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

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

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

(a) 42 (decimal number)

(b) 2a (hexadecimal number)

(c) 101010 (binary number)

(d) XLII (roman number)

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

(f) (qr code)

Section 10: Natural Numbers

Numeral Systems for Natural Numbers

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

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

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

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

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

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

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

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

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

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

Natural Numbers Literals

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

prefix	example	meaning	description
	42	42_{10}	decimal number
0x	0x42	$42_{16} = 66_{10}$	hexadecimal number
0o	0o42	$42_8 = 34_{10}$	octal number
0b	0b1000010	$1000010_2 = 42_{10}$	binary number
0	042	$42_8 = 34_{10}$	octal number (old)

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

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

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

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

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

Natural Numbers with Fixed Precision

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

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

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

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

```
1  -- Surprises with fixed precision unsigned integers (not only in Haskell)
2
3  import Data.Word
4  import Text.Printf
5
6  w8max = maxBound :: Word8
7  w8min = minBound :: Word8
8
9  main = do
10     printf "%3u + 1 = %3u\n" w8max (w8max + 1)
11     printf "%3u - 1 = %3u\n" w8min (w8min - 1)
12     printf " 42 + %3u + 1 = %3u\n" w8max (42 + w8max + 1)
13     printf " 42 - %3u - 1 = %3u\n" w8max (42 - w8max - 1)
```

This program produces the following output:

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

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

Section 11: Integer Numbers

Integer Numbers

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

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

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

Note how the mathematical definitions of f and f^{-1} directly translate into Haskell code:

```
1  f :: Integer -> Integer
2  f x
3      | x >= 0 = 2 * x
4      | x <  0 = -2 * x -1
5
6  g :: Integer -> Integer
7  g x
8      | even x = x `div` 2
9      | odd  x = -1 * (x+1) `div` 2
10
11 main = print $ map (g . f) [-10..10]
```

Note that Haskell does not really have a type for natural numbers. One reason is that subtraction on natural numbers may lead to results that do not exist in the set of natural numbers.

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

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

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

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

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

Two's Complement Fixed Integer Numbers (b complement)

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

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

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

Benefits of the two's complement for binary numbers:

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

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

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

Two's Complement Fixed Integer Number Ranges

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

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

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

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

```
1  -- Surprises with fixed precision integers (not only in Haskell)
2
3  import Data.Int
4  import Text.Printf
5
6  i8max = maxBound :: Int8
7  i8min = minBound :: Int8
8
9  main = do
10     printf "%3d + 1 = %3d\n" i8max (i8max + 1)
11     printf "%3d - 1 = %3d\n" i8min (i8min - 1)
```

This program produces the following output:

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

Section 12: 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
main = print $ 1%2 + 1%3
```


The following code examples are illustrative.

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 when we use these approximations. If we are not extremely cautious, these rounding errors can *accumulate* badly.
- Numeric algorithms can be analyzed according to how good or bad they propagate rounding errors, leading to the notion of *numeric stability*.

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

Section 13: 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

$$s \cdot d_0.d_1 d_2 \ldots d_{p-1} \cdot b^e = s \cdot \left(\sum_{k=0}^{p-1} d_k b^{-k} \right) \cdot b^e$$

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

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

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

Algorithm 3 Conversion of a decimal fraction f into a 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:      od
7:      return f
8:  end function
```

Floating Point Number Normalization

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

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

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

Further online information:

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

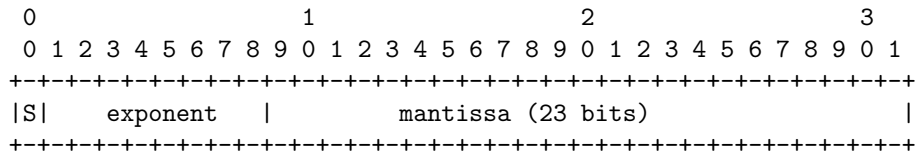
IEEE 754 Floating Point Formats

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

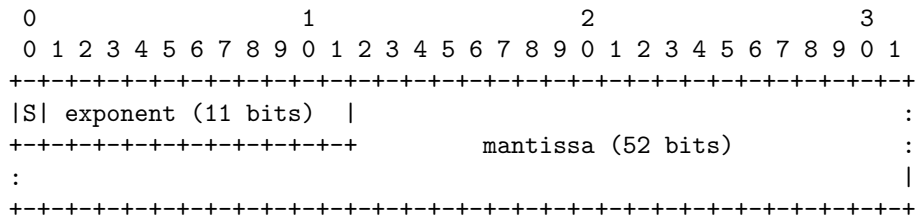
- IEEE 754 is a widely implemented standard for floating point numbers.
- IEEE 754 floating point numbers use the base $b = 2$ and as a consequence decimal numbers such as $1 \cdot 10^{-1}$ cannot be represented precisely.

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

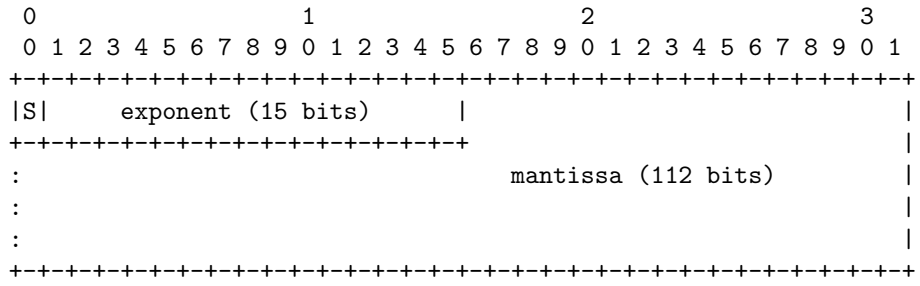
Single precision format:



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 ±infinity (inf)
  3. Overflow: returns ±infinity (inf)
  4. Underflow: depends on the operating mode
  5. Inexact: returns rounded result by default
- Computations may continue if they did produce a special value like nan or inf.
- Hence, it is important to check whether a calculation resulted in a value at all.

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

The special values use special encoding rules:

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

Floating Point Surprises

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


```
1  -- Surprises with double floating point numbers (not only in Haskell)
2
3  import Text.Printf
4
5  main = do
6      printf "2.0 / 0.0 = %g\n" (2.0 / 0.0 :: Double)
7      printf "2.0 / 0.0 + 1.0 = %g\n" (2.0 / 0.0 + 1.0 :: Double)
8      printf "2.0 / 0.0 - 2.0 / 0.0 = %g\n" (2.0 / 0.0 - 2.0 / 0.0 :: Double)
9      printf "(1.0 + 1e20) - 1e20 = %g\n" ((1.0 + 1e20) - 1e20 :: Double)
10     printf "1.0 + (1e20 - 1e20) = %g\n" (1.0 + (1e20 - 1e20) :: Double)
11     printf "0.1 + 0.2 = %g\n" (0.1 + 0.2 :: Double)
12     printf "counting from 0.0 up in steps of 0.1 = %s\n"
13                 $ show $ take 11 $ iterate (+0.1) 0.0
```

Further online information:

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

Section 14: 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 explicit about the units used.
  - NASA lost a Mars climate orbiter (worth $125 million) in 1999 due to a unit conversion error.
  - An Air Canada plane ran out of fuel in the middle of a flight in 1983 due to a fuel calculation error while switching to the metric system.
- There is an International System of Units (SI Units) to help you...

▶ Always be explicit about units.

▶ And always be clear about the unit prefixes.

Further online information:

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

SI Base Units

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.

Further online information:

- BIPM: “SI Brochure: The International System of Units”, 8th edition, updated 2014
- **Wikipedia**: 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	$kg\,m^2\,s^{-3}\,A^{-2}$	resistance
velocity		$m\,s^{-1}$	speed

Further online information:

- BIPM: “SI Brochure: The International System of Units”, 8th edition, updated 2014
- **Wikipedia**: SI Derived Unit

Metric Prefixes (International System of Units)

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

Metric Prefixes (International System of Units)

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

Binary Prefixes

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

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

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

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

Section 15: 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 instead encode bits of information used to control information flow or presentation.
- A *character encoding* is used to represent a set of characters by some kind of encoding system. A single character can be encoded in different ways.

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

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

Further online information:

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

ASCII Characters and Encoding

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

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

ASCII Characters and Code Points (decimal)

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


ASCII control characters:

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

Universal Coded Character Set and Unicode

- The Universal Coded Character Set (UCS) is a standard set of characters defined and maintained by the International Organization of Standardization (ISO).
- The Unicode Consortium produces industry standards based on the UCS for the encoding. Unicode 15.0 (published Sep. 2022) defines 149 186 characters, each identified by an unambiguous name and an integer number called its code point.
- The overall code point space is divided into 17 planes where each plane has $2^{16} = 65536$ code points. The Basic Multilingual Plane (plane 0) contains characters of almost all modern languages, and a large number of symbols.
- Unicode can be implemented using different character encodings. The UTF-32 encoding encodes character code points directly into 32-bit numbers (fixed length encoding). While simple, an ASCII text of size n becomes a UTF-32 text of size $4n$.

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

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

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

In Haskell, characters are Unicode characters. The Unicode code point for a given character can be obtained from the `ord :: Char -> Int` function and the Unicode character of a given code point can be obtained from the `chr :: Int -> Char` function defined in `Data.Char`. The `Data.Char` module defines many useful functions to classify characters or to change the case of characters.

Unicode Transformation Format UTF-8

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

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

The UTF-8 format is widely used when texts are exchanged between programs or over the Internet. It is defined in RFC 3629 [3], which was published in January 1998.

Strings

- Let Σ be a non-empty finite set of symbols (or characters), called the alphabet.
- A string (or word) over Σ is any finite sequence of symbols from Σ, including (of course) the empty sequence.
- Typical operations on strings are length(), concatenation(), reverse(), ...
- There are different ways to store strings 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. C programmers have to make sure that always sufficient memory is allocated to store the terminating byte.

In Haskell, a string is a list of characters. This is neither a time nor a space efficient representation. Hence, when processing larger pieces of text in Haskell, it is advisable to use `Data.Text`.

Section 16: Date and Time

System Time and Clocks

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

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

Another commonly available source of time is the Global Positioning System (GPS).

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

Further online information:

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

Calendar Time

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

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

Further online information:

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

ISO 8601 Date and Time Formats

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

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

The special letter Z indicates that the date and time is in UTC time. The ISO 8601 standard deals with timezone offsets by specifying the positive or negative offset from UTC time in hours and minutes. This in principle allows us to define +00:00 and -00:00 but the ISO 8601 disallows a negative zero offset. RFC 3339 [8], a profile of ISO 8601, however, defines that -00:00 indicates that the time is in UTC and that the local timezone offset is unknown.

Further online information:

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

Part IV

Boolean Algebra

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

- Boolean algebra is the foundation for understanding digitial circuits, the foundation of today's computers. The central processing unit (CPU) of a computer is essentially a very large collection of digital circuits consisting of logic gates. The mathematical foundation of a CPU is Boolean algebra, it helps us to understand how digital circuits can 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 algebra and Boolean logic. More advanced logics and their usage will be covered in more advanced courses.

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

- understand the concept of Boolean variables and their interpretation;
- familar with the basic boolean functions such as $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\underline{\vee}$, $\overline{\wedge}$, $\overline{\vee}$;
- illustrate how Boolean functions relate to Boolean expressions;
- differentiate between the syntax and the semantics of Boolean expressions;
- apply Boolean equivalence laws;
- describe the notion of tautologies and contradictions;
- convert Boolean expressions into conjunctive and disjunctive normal forms;
- obtain Boolean expressions in normal form from truth tables;
- apply the Quine McCluskey algorithm to minimize Boolean expressions;
- describe the representation of basic statements of Boolean logic;
- explain the satisfiability problem and why it is hard to solve in the general case.

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."

Let P denotate the I like Pat and let J denote that I like Joe. In Boolean algebra, the following derivation holds:

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

In this section, we will introduce the laws that enable us to formulate such a derivation.

Section 17: 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 $\neg$ function applied to X is also written as $\overline{X}$.

- 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 :: Bool -> Bool`).

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:

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

- In programming languages like C or C++ (and even Haskell), the operator == is often used to represent the equivalence function as an operation.

Boolean $\veebar$ Function (exclusive or)

X	Y	$X \veebar Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

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

- 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:

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

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

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

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

X	Y	$X \veebar Y$	$(X \veebar Y) \veebar Y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Boolean $\overline{\wedge}$ Function (not-and)

X	Y	$X \overline{\wedge} Y$
0	0	1
0	1	1
1	0	1
1	1	0

- The logical *not-and* (nand) or $\overline{\wedge}$ can be viewed as a function that maps two Boolean values to a Boolean value:

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

- The $\overline{\wedge}$ function is also written using the Sheffer stroke $\uparrow$.

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

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

- $X \wedge Y = (X \overline{\wedge} Y) \overline{\wedge} (X \overline{\wedge} Y)$

X	Y	$X \wedge Y$	$X \overline{\wedge} Y$	$(X \overline{\wedge} Y) \overline{\wedge} (X \overline{\wedge} Y)$
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

- $X \vee Y = (X \overline{\wedge} X) \overline{\wedge} (Y \overline{\wedge} Y)$

X	Y	$X \vee Y$	$X \overline{\wedge} X$	$Y \overline{\wedge} Y$	$(X \overline{\wedge} X) \overline{\wedge} (Y \overline{\wedge} Y)$
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	0	0	1

- $\neg X = X \overline{\wedge} X$

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

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

X	Y	$X \mathbin{\overline{\vee}} Y$
0	0	1
0	1	0
1	0	0
1	1	0

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

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

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

- The $\overline{\vee}$ (nor) is like $\overline{\wedge}$ (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.

- $X \wedge Y = (X \mathbin{\overline{\vee}} X) \mathbin{\overline{\vee}} (Y \mathbin{\overline{\vee}} Y)$
- $X \vee Y = (X \mathbin{\overline{\vee}} Y) \mathbin{\overline{\vee}} (X \mathbin{\overline{\vee}} Y)$
- $\neg X = X \mathbin{\overline{\vee}} X$

Section 18: 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++	C / C++ (bits)
and	X and Y	$X \wedge Y$	$X \cdot Y$	`X && Y`	`&`
or	X or Y	$X \vee Y$	$X + Y$	`X || Y`	`|`
not	not X	$\neg X$	$\overline{X}, X'$	`! X`	`~`
implication	X impl Y	$X \rightarrow Y$			
equivalence	X equiv Y	$X \leftrightarrow Y$		`X == Y`	
exclusive or	X xor Y	$X \veebar Y$	$X \oplus Y$		`^`
not and	X nand Y	$X \barwedge Y$, $X \overline{\wedge} Y$	$\overline{X \cdot Y}$		
not or	X nor Y	$X \barvee Y$, $X \overline{\vee} Y$	$\overline{X + Y}$		

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:

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

- Not and:

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

- Not or:

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

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

Section 19: 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 it is a tautology or a contradiction. One way to find out would be to compute the complete truth table. Recall that each row in a truth table corresponds to one possible interpretation of the variables in 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 a contradiction, which is less costly than computing the entire truth table. In fact, logicians have reason to believe that no faster method exists (but this is an unproven conjecture!).

Satisfying a Boolean formula

Definition (satisfying a Boolean formula)

An interpretation $\mathcal{I}$ which is adapted to a Boolean formula 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:

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

Boolean Equivalence laws

Proposition (equivalence laws)

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

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

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

Section 20: Normal Forms (CNF and DNF)

Literals, Monomials, Clauses

Definition (literals)

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

Definition (monomial)

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

Definition (clause)

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

Conjunctive Normal Form

Definition (conjunctive normal form)

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

- Examples of formulas in CNF:
  - X_1 (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.

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.

Minterms and Maxterms

Definition (minterm)

A *minterm* of a Boolean function $\varphi(X_n, \ldots, X_1, X_0)$ is a monomial $(\hat{X}_n \wedge \ldots \wedge \hat{X}_1 \wedge \hat{X}_0)$ where $\hat{X}_i$ is either X_i or $\neg X_i$. A shorthand notation is m_d where d is the decimal representation of the binary number obtained by replacing all negative literals with 0 and all positive literals with 1 and by dropping the operator.

Definition (maxterm)

A *maxterm* of a Boolean function $\varphi(X_n, \ldots, X_1, X_0)$ is a clause $(\hat{X}_n \vee \ldots \vee \hat{X}_1 \vee \hat{X}_0)$ where $\hat{X}_i$ is either X_i or $\neg X_i$. A shorthand notation is M_d where d is the decimal representation of the binary number obtained by replacing all negative literals with 1 and all positive literals with 0 and by dropping the operator.

For example, the Boolean function

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

can be written as

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

or with the alternative notation as

$$\varphi(X, Y, Z) = m_1 + m_2 + m_4 + m_5 + m_6.$$

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

$$\varphi(X, Y, Z) = M_0 \wedge M_3 \wedge M_7$$

or the alternative notation

$$\varphi(X, Y, Z) = M_0 \cdot M_3 \cdot M_7.$$

The CNF is therefore given by:

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

Obtaining a DNF from a Truth Table

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

X	Y	$X \veebar Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

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

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

Obtaining a CNF from a Truth Table

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

X	Y	$X \veebar Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

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

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

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

Section 21: 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.

With this, the cost of the Boolean expression $((X \vee Y) \wedge (\neg Y \wedge Z)) \wedge Z$ is $C(((X \vee Y) \wedge (\neg Y \wedge Z)) \wedge Z) = 4$ and the cost of the Boolean expression $X \wedge \neg Y \wedge Z$ is $C(X \wedge \neg Y \wedge Z) = 2$. Since

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

and there is no way to further minimize $X \wedge \neg Y \wedge Z$, the cost of the Boolean function φ is $C(\varphi) = 2$.

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 disjunction (sum) of prime implicants that defines the function. (This sum not necessarily includes all prime implicants.)

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

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

Finding Prime Implicants (QM-1)

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

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

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

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

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

Example: Minimize $\varphi(W,X,Y,Z) = m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{14} + m_{15}$

- 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 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 columnns are the original minterms and the rows represent the prime implicants. The marked cells in the table indicate whether a prime implicant covers a minterm. (• = essential, $*$ = covered, ○ = uncovered)

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

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

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

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

Section 22: Boolean Logic and the Satisfiability Problem

Logic Statements

- A common task is to decide whether statements of the following form are true:
  if *premises* P_1 **and** ... **and** P_m *hold,* **then** *conclusion* C *holds*
- The premises P_i and the conclusion C are expressed in some logic formalism, the simplest is Boolean logic (also called propositional logic).
- Restricting us to Boolean logic here, the statement above can be seen as a Boolean formula of the following structure

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

  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

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

  is a tautology, we may check whether

  $$\tau' = \neg((\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi)$$

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

Tautology and Satisfiability

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

$$\tau = (\varphi_1 \wedge \ldots \wedge \varphi_m) \to \psi = \neg(\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi)$$

$$\tau' = \neg((\varphi_1 \wedge \ldots \wedge \varphi_m) \to \psi) = (\varphi_1 \wedge \ldots \wedge \varphi_m \wedge \neg\psi)$$

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

Satisfiability Problem

Definition (satisfiability problem)

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

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

There is no known general algorithm that efficiently (means in polynomial time) solves the SAT problem, and it is generally believed that no such algorithm exists. However, this belief has not been proven mathematically.

Resolving the question whether SAT has a polynomial-time algorithm is equivalent to answering the P versus NP problem, which is a famous open problem in the theory of computer science: The complexity class P contains all problems that are solvable in polynomial time by a deterministic machine and the complexity class NP contains all problems that are solvable in polynomial time by a non-deterministic machine (i.e., a machine that guesses the next best step). Obviously, $P \subseteq NP$. The big question is whether $P = NP$ holds. (Please recall that NP stands for non-deterministic polynomial time.)

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

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

Further online information:

- **Wikipedia**: Boolean satisfiability problem
- **Wikipedia**: Millennium Prize Problems
- **YouTube**: P vs. NP – The Biggest Unsolved Problem in Computer Science

Part V

Computer Architecture

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

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

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

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

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

- describe the relationship between Boolean algebra and digital circuits;
- familiar with the symbols used to draw logic gates and digital circuits;
- explain how a half, full, and ripple adder are constructed;
- outline why a carry-lookahead adder is faster than a ripple adder;
- tell other basic digital circuits such as a magnitude comparator, a multiplexer or a demultiplexer;
- describe the difference between combinational and sequential digital circuits;
- enumerate different kinds of latches and flip-flops;
- detail how latches and flip-flops can be constructed using NAND and NOR gates;
- illustrate the difference between level-triggered and edge-triggered digital circuits;
- sketch the different components of the von Neumann computer architecture;
- contrast the purposes of the control, address, and data bus;
- present the basic structure of a CPU and the fetch-decode-execute cycle;
- recognize programs in machine language and in assembly language;
- demonstrate the notion of function calls and stacks and stack frames;
- summarize the RISC-V base instruction set architecture.

Section 23: Combinational 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 ($\veebar$), NAND ($\barwedge$), and NOR ($\barvee$).

$$X \veebar Y := (X \vee Y) \wedge \neg(X \wedge Y)$$
$$X \barwedge Y := \neg(X \wedge Y)$$
$$X \barvee Y := \neg(X \vee Y)$$

- 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 2: Truth tables for the elementary operations AND, OR, and NOT

X	Y	$X \veebar Y$
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	$X \barwedge Y$
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	$X \barvee Y$
0	0	1
0	1	0
1	0	0
1	1	0

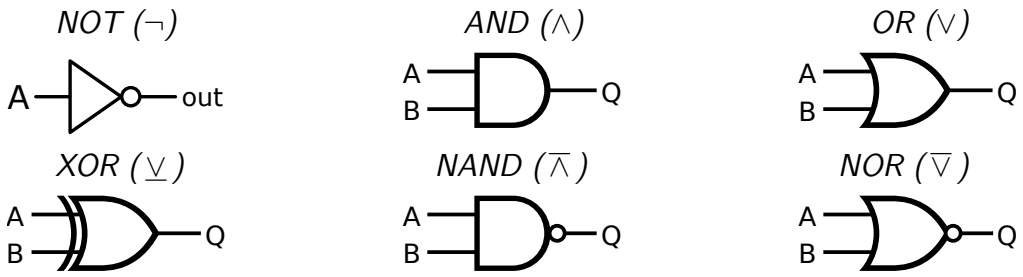
Figure 3: Truth tables for the elementary functions XOR, NAND, and NOR

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 online information:

- **Wikipedia**: 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.

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

There are several web-based tools to design and simulate logic circuits. Some use different symbols, like the European symbols for logic gates.

Further online information:

- **Web**: simulator.io: Web-based logic circuit simulator

Combinational Digital Circuits

Definition (combinational digital circuit)

A *combinational digital circuit* implements a pure boolean function where the result depends only on the inputs.

- Examples of elementary combinational digitial circuits are circuits to add n-bit numbers, to multiply n-bit numbers, or to compare n-bit numbers.
- Combinational digital circuits are pure since their behavior solely depends on the well-defined inputs of the circuit.
- An important property of a combinational digital circuit is its *gate delay*.

Addition of decimal and binary numbers

```
  2      0010        3      0011        8      1000
+ 5    + 0101      + 3    + 0011      + 3    + 0011
                                 11          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.

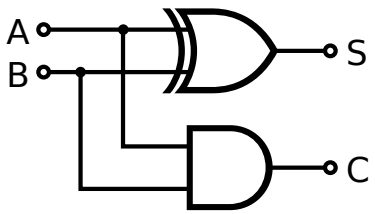
Adding two bits (half adder)

- 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

$$S = A \veebar B$$

$$C = A \wedge B$$



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 online information:

- **Wikipedia**: Adder (electronics)

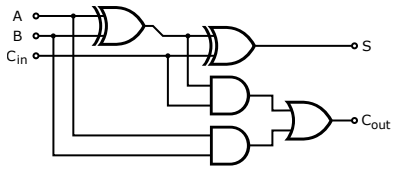
Adding two bits (full adder)

- 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

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

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

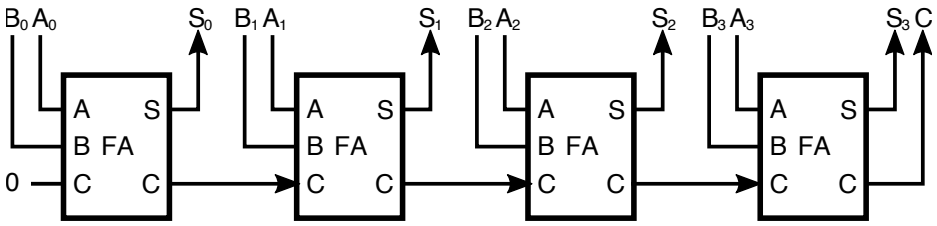


Assuming that every gate has the same gate delay t_g, the full adder works with a constant gate delay of $t_{fa} = 3 \cdot t_g$. Note that the carry in signal only has a gate delay of $2 \cdot t_g$ to the carry out signal.

Further online information:

- **Wikipedia**: Adder (electronics)

Adding N Bits (ripple carry adder)



- And N-bit adder can be created by chaining multiple full adders.
- Each full adder inputs a C_{in}, which is the C_{out} of the previous adder.
- Each carry bit "ripples" to the next full adder.

The layout of a ripple-carry adder is simple. However, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated by the previous full adder. Assuming that every gate has the same gate delay t_g, the n-bit ripple-carry adder has a gate delay of $t_{ra} = n \cdot t_{fa} = 3n \cdot t_g$.

If we built an n-bit ripple-carry adder from elementary logic gates, then the slow path is the carry propagation, which gives us an overall delay of $1 + 2n \cdot t_g$. To see why we get a delay of $1 + 2n \cdot t_g$, lets look at a 4-bit adder taking the inputs $A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3$ and the carry C_0. Using these inputs, we can calculate the following intermediate values using one gate delay:

$$T_0 = (A_0 \veebar B_0)$$
$$U_0 = (A_0 \wedge B_0)$$
$$T_1 = (A_1 \veebar B_1)$$
$$U_1 = (A_1 \wedge B_1)$$
$$T_2 = (A_2 \veebar B_2)$$
$$U_2 = (A_2 \wedge B_2)$$
$$T_3 = (A_3 \veebar B_3)$$
$$U_3 = (A_3 \wedge B_3)$$

It now takes two more gate delays to compute

$$C_1 = ((C_0 \wedge T_0) \vee U_0)$$

and with that it takes two more gate delays to compute

$$C_2 = ((C_1 \wedge T_1) \vee U_1)$$

and with that it takes two more gate delays to compute

$$C_3 = ((C_2 \wedge T_2) \vee U_2)$$

and with that it takes two more gate delays to compute

$$C_4 = ((C_3 \wedge T_3) \vee U_3).$$

The delay of an n-bit ripple-carry adder grows linearly with the number of bits. For a 64-bit adder, we get a gate delay of 129 gate times, which is not nice.

Adding N Bits (carry-lookahead adder)

- A carry-lookahead adder uses a special circuit to calculate the carry bits faster.
- The carry bit c_{i+1} 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 $b_i = 1$:

$$c_{i+1} = (a_i \wedge b_i) \vee ((a_i \veebar b_i) \wedge c_i)$$

- We can write this as follows:

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

- Half adder can be used to (i) add the input bits and to feed the carry-lookahead circuit and to (ii) finally add the carry bits.

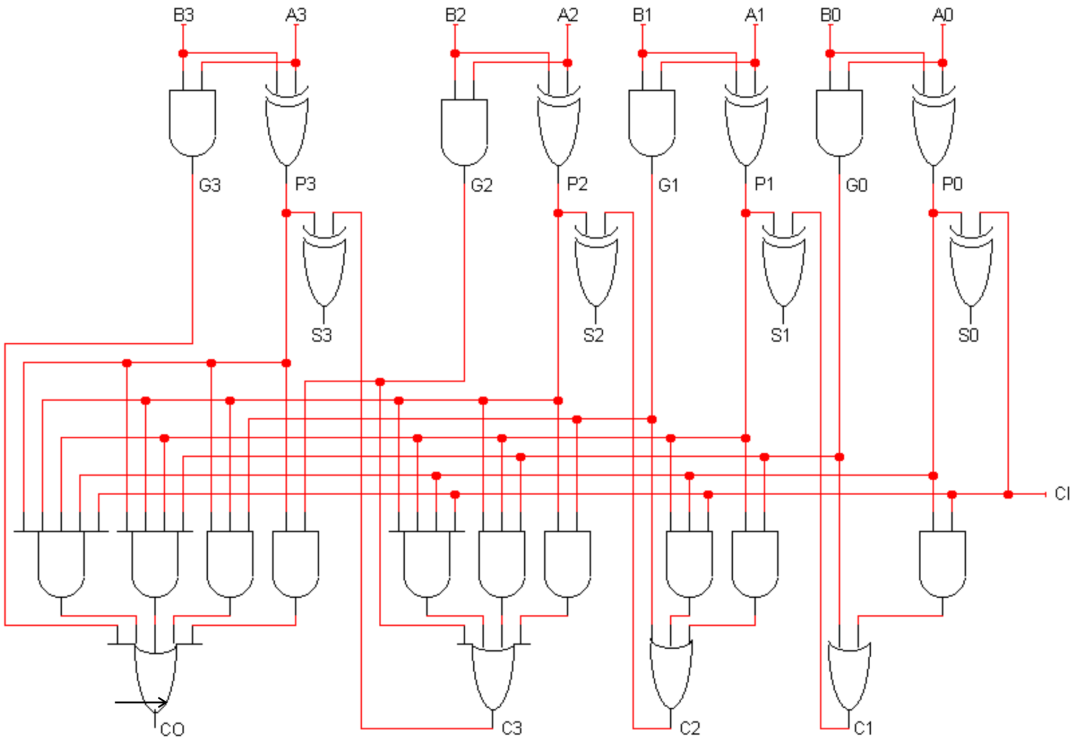
It is possible to design adders that reduce the gate delay. A classic example is the carry lookahead adder. Lets consider how the carry bit works: The carry bit is either the result of adding $a_i = 1$ and $b_i = 1$ or it is the result of $c_i = 1$ and either $a_i = 1$ or $b_i = 1$.

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

The function g_i "generates" a carry bit and the function p_i "propagates" a carry bit. The equation above gives us a recursive definition how c_i can be calculated. With 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 online information:

- **Wikipedia**: Adder (electronics)

Multiplication of Decimal and Binary Numbers

```
11 * 13          1011 * 1101
-------          -----------
    11               1011
+    33          +    1011
-------          +     0000
    143          +      1011
                 -----------
                   10001111
```

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

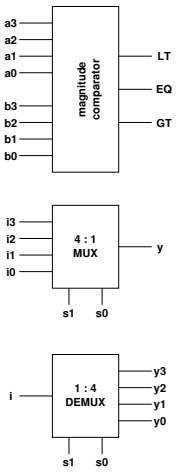
The multiplication of a two bits is simply a logical and operation.

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

Straight-forward digital multiplication circuits are conceptually simply but relatively complex in terms of the number of gates needed. Optimizations can be used for special cases. For example, multiplication by 2 is simply a left shift of bits. We will not study multiplication circuits further here. Students interested to dive deeper may want to lookup the Wallace tree fast multiplier [19].

Magnitude Comparator, Multiplexer, Demultiplexer

- An n-bit magnitude comparator takes two n-bit numbers a and b and outputs whether $a < b$, $a > b$, or $a = b$.
- An n-bit multiplexer has $n = 2^m$ inputs $i_0, i_1, \ldots, i_n$ and m selection inputs $s_0, s_1, \ldots, s_m$ and it outputs on y the input i_x where x is the number indicated by the selection inputs.
- An n-bit demultiplexer has $n = 2^m$ outputs $y_0, y_1, \ldots, y_n$, an input i and m selection inputs $s_0, s_1, \ldots, s_m$. It outputs the input i on the output y_x where x is the number indicated by the selection inputs.



This slide shows some additional combinational circuits that are often used as building blocks for more advanced digital circuits. The design of these circuits is relatively straight-forward and can be done as a little exercise.

Section 24: Sequential Digital Circuits

Sequential Digital Circuits

Definition (sequential digital circuit)

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

Definition (asynchronous sequential digital circuit)

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

Definition (synchronous sequential digital circuit)

A sequential digitial circuit is *synchronous* if the state of the circuit and the results can change only at discrete times in response to a clock.

Basic Properties of Memory

Memory should have at least three properties:

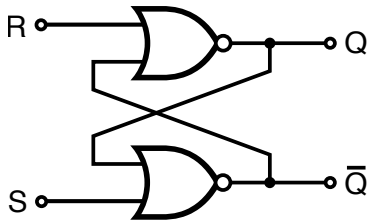
1. It should be able to hold a value.
2. It should be possible to read the value that was saved.
3. It should be possible to change the value that was saved.

We start with the simplest case, a one-bit memory:

1. It should be able to hold a single bit.
2. It should be possible to read the bit that was saved.
3. It should be possible to change the bit that was saved.

SR Latch using NOR Gates

S	R	Q	$\bar{Q}$	Q	$\bar{Q}$
0	0	0	1	0	1
0	0	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1



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

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

$$Q_{next} = R \bar{\vee} \bar{Q}_{current}$$
$$\bar{Q}_{next} = S \bar{\vee} Q_{current}$$

Lets try to understand the behavior of this circuit by looking at the different possible cases:

- $S = 0 \wedge R = 0$: The values of Q and $\bar{Q}$ do not change.

$$Q_{next} = R \bar{\vee} \bar{Q}_{current} = 0 \bar{\vee} \bar{Q}_{current} = Q_{current}$$
$$\bar{Q}_{next} = S \bar{\vee} Q_{current} = 0 \bar{\vee} Q_{current} = \bar{Q}_{current}$$

- $S = 1 \wedge R = 0$: The value of $\bar{Q}$ becomes 0 and subsequently the value of Q becomes 1.

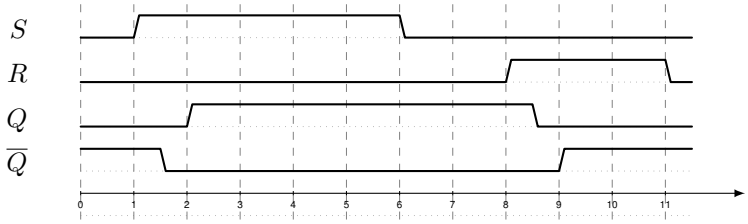
$$\bar{Q}_{next} = S \bar{\vee} Q_{current} = 1 \bar{\vee} Q_{current} = 0$$
$$Q_{next} = R \bar{\vee} \bar{Q}_{current} = 0 \bar{\vee} \bar{Q}_{current} = 1$$

- $S = 0 \wedge R = 1$: The value of Q becomes 1 and subsequently the value of $\bar{Q}$ becomes 0.

$$Q_{next} = R \bar{\vee} \bar{Q}_{current} = 1 \bar{\vee} \bar{Q}_{current} = 0$$
$$\bar{Q}_{next} = S \bar{\vee} Q_{current} = 0 \bar{\vee} Q_{current} = 1$$

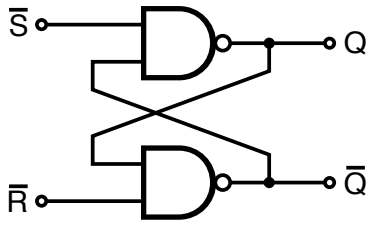
- $S = 1 \wedge R = 1$: This may lead to oscillation and hence this input should not occur.

To understand sequential circuits, it is useful to look at timing diagrams. The following diagram shows the timing at the gate level.



SR Latch using NAND Gates

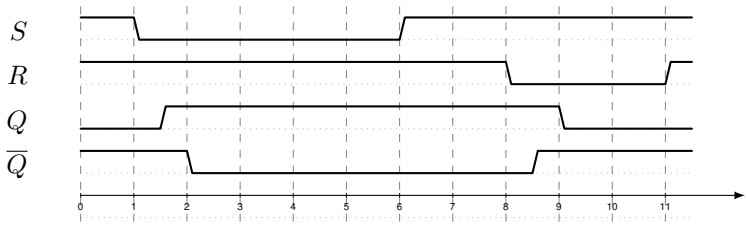
$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	Q	$\bar{Q}$
1	1	0	1	0	1
1	1	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1



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

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

The following diagram shows the timing at the gate level.

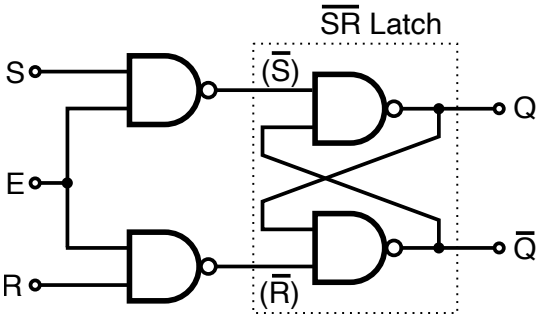


Further online information:

- **YouTube**: Latches and Flip-Flops 1 - The SR Latch

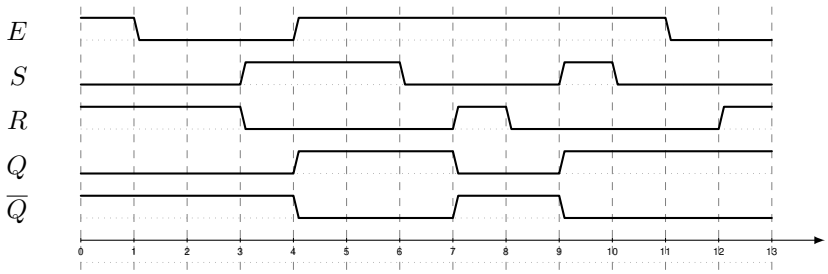
Gated SR Latch using NAND Gates

E	S	R	Q	$\bar{Q}$	Q	$\bar{Q}$
0	x	x	0	1	0	1
0	x	x	1	0	1	0
1	0	0	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	1	0	1
1	0	1	1	0	0	1
1	1	0	0	1	1	0
1	1	0	1	0	1	0



- The control input E enables the latch, the latch does not change while E is low.

Basic latches permanently follow changes in the inputs. Since this is sometimes not desirable, it is possible to introduce an additional enable (E) input. As long as $E = 0$, the latch does not change state.

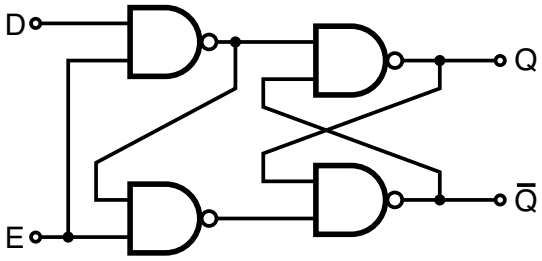


Further online information:

- **YouTube**: Latches and Flip-Flops 2 - The Gated SR Latch

Gated D Latch using NAND Gates

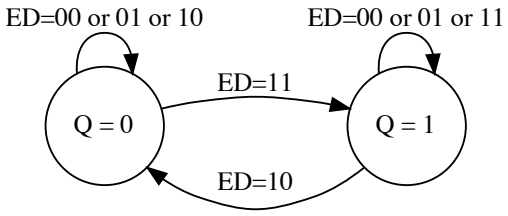
E	D	Q	$\bar{Q}$	Q	$\bar{Q}$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
1	0	x	x	0	1
1	1	x	x	1	0



- There is no illegal input combination anymore
- There is no input combination anymore for "keep and do not change"

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

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



Further online information:

- **YouTube**: Latches and Flip-Flops 3 - The Gated D Latch

Level-triggered versus Edge-triggered

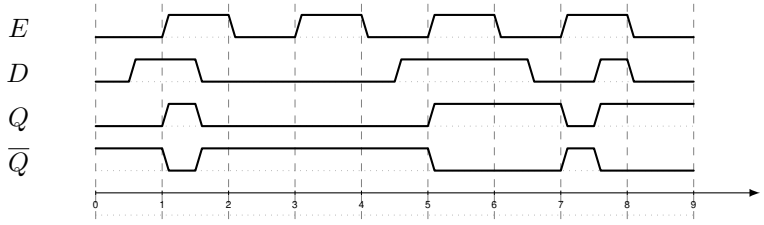
Definition (level-triggered)

A digital circuit is called *level-triggered* if changes of the inputs affect the output as long as the clock input is high (i.e., as long as the circuit is enabled).

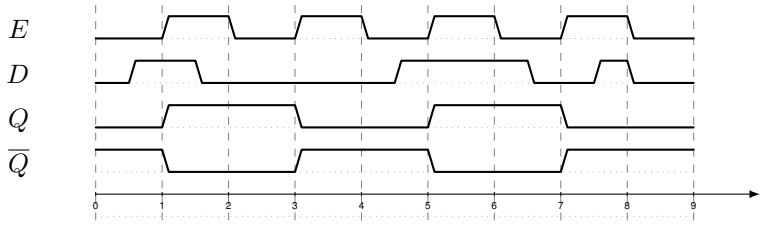
Definition (edge-triggered)

A digital circuit is called *edge-triggered* if the inputs affect the output only when the clock input transitions from low to high (positive edge-triggered) or from high to low (negative edge-triggered).

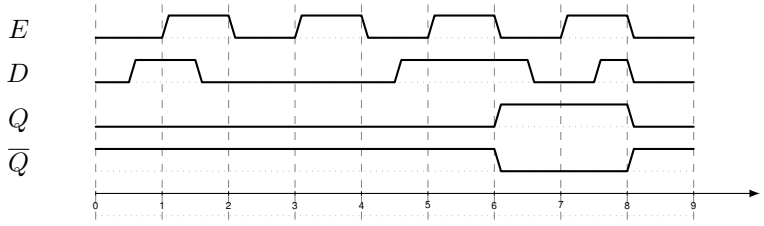
Level-triggered D latch:



Positive edge-triggered D latch:



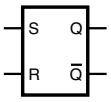
Negative edge-triggered D latch:



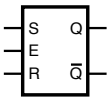
Further online information:

- **YouTube**: Latches and Flip-Flops 4 - The Clocked D Latch

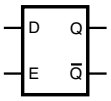
Latch Types and Symbols



SR Latch

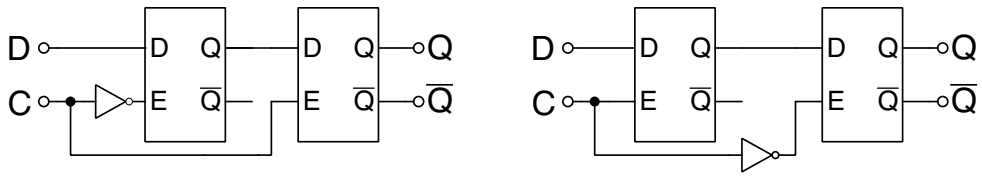


Gated SR Latch



Gated D Latch

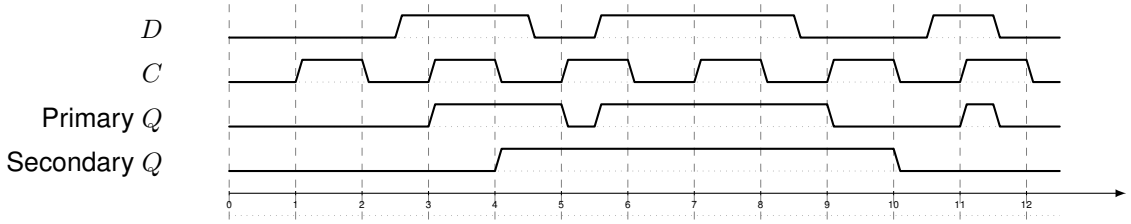
D Flip-Flop (positive and negative edge-triggered)



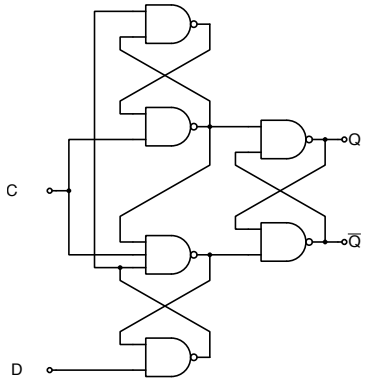
- Edge-triggered D flip-flops propagate changes from the primary to the secondary latch on either the rising edge of a clock or the falling edge of the clock.
- Positive edge-triggered: state changes when the clock becomes high
- Negative edge-triggered: state changes when the clock becomes low

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

Timing diagram for a negative edge-triggered D flip-flop:



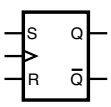
A positive edge-triggered D flip-flop can be constructed out of three SR latches using a circuit only using NAND gates.

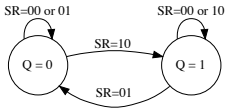


Further online information:

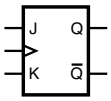
- **YouTube**: Latches and Flip-Flops 5 – D Type Flip Flop

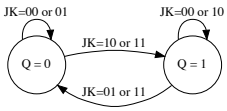
Flip-Flop Types, State Diagrams, and Symbols



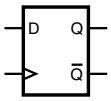


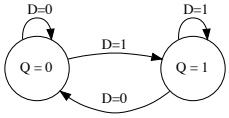
SR Flip-Flop



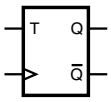


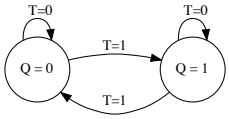
JK Flip-Flop





D Flip-Flop





T Flip-Flop

All flip-flops are edge triggered on the clock input C.

- SR flip-flop:

  If the S input is asserted, then the bit stored in the flip-flop is set. If the R input is asserted, then the bit stored in the flip-flop is cleared. If the S and R inputs are both asserted, then the flip-flop behaviour is undefined. If neither S nor R is set, then the bit stored in the flip-flop does not change.

- JK flip-flop:

  If the J input is asserted, then the bit stored in the flip-flop is set. If the K input is asserted, then the bit stored in the flip-flop is cleared. If the J and K inputs are both asserted, then the bit stored in the flip-flop is toggled, i.e., it moves from 0 to 1 or from 1 to 0. If neither J nor K is set, then the bit stored in the flip-flop does not change.

- D flip-flop:

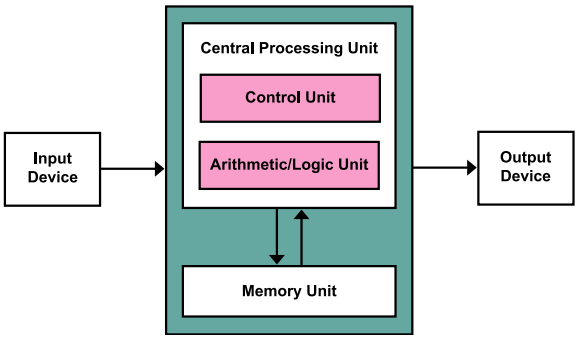
  If the D input is asserted, then the bit stored in the flip-flop is set. If the D input is not asserted, then the bit stored in the flip-flop is cleared.

- T flip-flop:

  The T flip-flop toggles the bit stored in the flip-flop. If the T input is asserted, the bit stored in the flip-flop is toggled. Otherwise, the bit does not change.

Section 25: 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
- Processor registers provide small amount of storage as part of a central processing unit

- The memory unit stores data and program instructions
- The central processing unit (CPU) carries out the actual computations

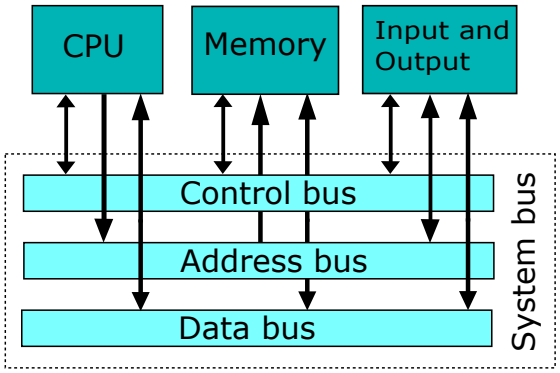
- Mass storage and input/output devices communicate with the central processing unit.
- The memory unit stores both data and program instructions.
- The ALU contains many basic digital circuits, such as N-bit adder and comparator.
- The processor usually operates on binary data with a certain number 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 performed by the central processing unit.
- 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.

The von Neumann architecture is quite popular but there have been other proposals. Most widely know is the Harvard architecture, which separates instruction memory from data memory.

Further online information:

- **Wikipedia**: 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.

- The Peripheral Component Interconnect (PCI) is an example of a CPU external system bus (exists in multiple versions)

Parallel busses carry data words in parallel using multiple wires. Parallel busses are highly efficient if the distance between the components is small. Close to the central processing unit, busses are usually parallel. Examples of parallel busses:

- PATA: Parallel Advanced Technology Attachment (primarily used to connect hard drives, also known as IDE)
- SCSI: Small Computer System Interface (SCSI) (primarily used to connect external storage devices)
- PCI: Peripheral Component Interconnect (primarily used internally to interconnect computer components)

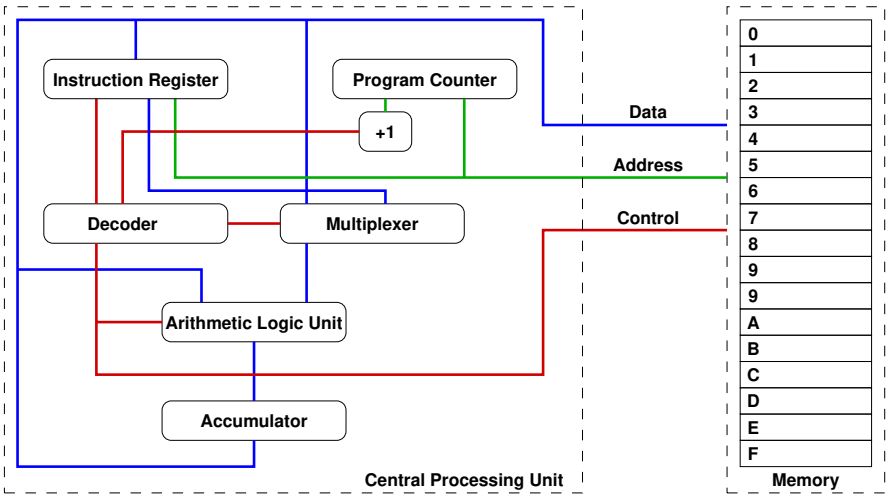
Serial busses carry data in bit-serial form over simple wires. Serial busses are highly efficient if the distance to cover is long. Examples of serial busses:

- SATA: Serial ATA (primarily used to connect hard drives)
- PCI-E: Peripheral Component Interconnect Express (serial bus designed to replace PCI)
- RS-232: Recommended Standards 232 (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 online information:

- **Wikipedia**: Bus (computing)

Simple Central Processing Unit



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.

Real CPUs usually have multiple registers, they provide vary instruction sets and registers to support different privilege levels, and they have special units for floating point arithmetic or cryptographic operations.

Instruction cycle (fetch – decode – execute cycle)

```
while True:
    instruction = fetch();
    advance_program_counter();
    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 online information:

- **Wikipedia**: Instruction cycle
- **Wikipedia**: Machine code
- **Wikipedia**: Assembly language
- **YouTube**: Inside the CPU - Computerphile
- **YouTube**: Fetch Decode Execute Cycle in more detail

Simple Machine Language

Op-code	Instruction	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 online information:

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

Program #1 in our simple machine language

Cell	Machine Code	Assembler	Description
0	001 1 0010	`LOAD #2`	acc = 2
1	010 0 1101	`STORE 13`	m[13] = acc
2	001 1 0101	`LOAD #5`	acc = 5
3	010 0 1110	`STORE 14`	m[14] = acc
4	001 0 1101	`LOAD 13`	acc = m[13]
5	011 0 1110	`ADD 14`	acc += m[14]
6	010 0 1111	`STORE 15`	m[15] = acc
7	111 0 0000	`HALT`	stop

- What is an 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

Cell	Machine Code	Assembler	Description
0	001 1 0101	`LOAD #5`	acc = 5
1	010 0 1111	`STORE 15`	m[15] = acc
2	001 1 0000	`LOAD #0`	acc = 0
3	101 0 1111	`EQUAL 15`	if acc == m[15] skip
4	110 1 0110	`JUMP #6`	pc = 6
5	111 0 0000	`HALT`	halt
6	011 1 0001	`ADD #1`	acc += 1
7	110 1 0011	`JUMP #3`	pc = 3

- What is an 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
```

Improvements of the Simple Machine Language

- We can improve our simple machine language in a number of ways:
  1. Moving to a 16-bit instruction format, where 6 bits are used to identify op-codes (allowing up to 64 instructions), 2 bits are used for different addressing modes, and 8 bits are used as operands (giving us an address space of 256 memory cells).
  2. Adding additional registers so that intermediate results do not always have to be written back to memory.
  3. Adding indirect addressing modes using base registers in order to move to 16-bit address spaces (65536 memory cells).
  4. Adding support for function calls and returns to modularize assembly code and to support code reuse.
- Some processors often use variable length instruction formats where an instruction may take one or several bytes to encode the instruction.

Call Stacks and Stack Frames

Definition (call stack and stack frames)

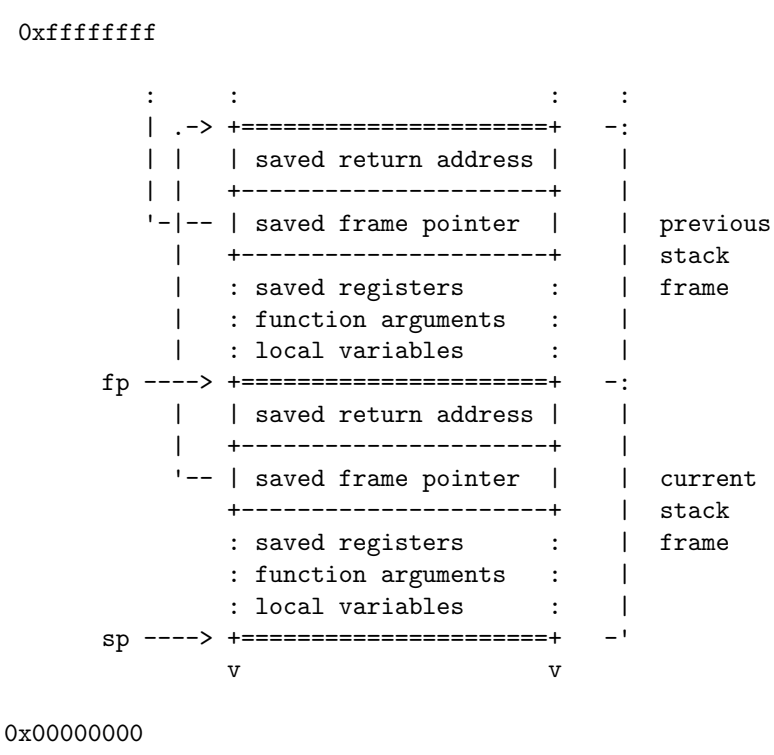
A *call stack* holds a stack frame for every active function call. A *stack frame* provides memory space to store

1. the return address to load into the program counter when the function returns,
2. local variables that exist during an active function call,
3. arguments that are passed into the function or results returned by the function.

To support function calls, a CPU needs to provide

- a register pointing to the top of the call stack (stack pointer),
- a register pointing to the start of the current stack frame (frame pointer),
- a mechanism to call a function (allocating a new stack frame on the call stack),
- an mechanism to return from a function (deleting a stack frame from the stack).

The drawing below shows a typical call stack growing downwards from large addresses towards small addresses. Each active function call results in a stack frame where the return address, a frame pointer, saved registers, function arguments and local variables are stored. Stack frames enable nested and recursive function calls. They also provide a very efficient way to allocate memory for local variables.



Function call prologue and epilogue

Definition (function call prologue)

A *function call prologue* is a sequence of machine instructions at the beginning of a function that prepare the stack and the registers for use in the function.

Definition (function call epilogue)

A *function call epilogue* is a sequence of machine instructions at the end of a function that restores the stack and CPU registers to the state they were in before the function was called.

Definition (stack and frame pointer)

The *stack pointer* is a register pointing to the beginning of the function call stack. The *frame pointer* is a register pointing to the beginning of the current stack frame.

RISC-V Instruction Set Architecture

Name	Description	Version	Status
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
M	Integer Multiplication and Division	2.0	Ratified
A	Atomic Instructions	2.1	Ratified
F	Single-Precision Floating-Point	2.2	Ratified
D	Double-Precision Floating-Point	2.2	Ratified
Zicsr	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
C	Compressed Instructions	2.0	Ratified
...	...		

- Open source RISC instruction set architecture (UC Berkeley, 2010)
- Meanwhile owned by RISC-V International, a Swiss nonprofit business association

The RISC-V base 32-bit integer ISA [18] supports 32 registers, each register being 32 bit long, plus a 32-bit program counter. The 64-bit integer ISA expands all registers to 64-bit. The following table summarizes how the registers are commonly used.

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-7	t0-2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments / return value
x12-17	a2-7	Function arguments
x18-27	s2-s11	Saved registers
x28-31	t3-6	Temporaries
pc	pc	Program counter

The saved registers s0 to s11 are typically preserved across function calls, while the argument registers a0 to a7 and the temporary registers t0 to t6 are not.

The ISA supports a minimalistic set of instructions. Assemblers often support additional pseudo instructions that make assembler code more readable but essentially expand to ISA instructions. For example, the `nop` (no operation) assembler instruction translates into `addi x0, x0, 0`, which says add 0 to `zero` and store the result into `zero`.

Further online information:

- **YouTube**: emulsiV: A visual simulator for the RISC-V instruction set
- **Web**: emulsiV: A visual simulator for the RISC V instruction set

Inst	Name	Description (C)	Note
add	ADD	`rd = rs1 + rs2`	
sub	SUB	`rd = rs1 - rs2`	
xor	XOR	`rd = rs1 ^ rs2`	
or	OR	`rd = rs1 | rs2`	
and	AND	`rd = rs1 & rs2`	
sll	Shift Left Logical	`rd = rs1 << rs2`	
srl	Shift Right Logical	`rd = rs1 >> rs2`	
sra	Shift Right Arith*	`rd = rs1 >> rs2`	msb-extends
slt	Set Less Than	`rd = (rs1 < rs2) ? 1 : 0`	
sltu	Set Less Than (U)	`rd = (rs1 < rs2) ? 1 : 0`	zero-extends
addi	ADD Immediate	`rd = rs1 + imm`	
xori	XOR Immediate	`rd = rs1 ^ imm`	
ori	OR Immediate	`rd = rs1 | imm`	
andi	AND Immediate	`rd = rs1 & imm`	
slli	Shift Left Logical Imm	`rd = rs1 << imm[0:4]`	
srli	Shift Right Logical Imm	`rd = rs1 >> imm[0:4]`	
srai	Shift Right Arith Imm	`rd = rs1 >> imm[0:4]`	msb-extends
slti	Set Less Than Imm	`rd = (rs1 < imm)?1:0`	
sltiu	Set Less Than Imm (U)	`rd = (rs1 < imm)?1:0`	zero-extends
lb	Load Byte	`rd = M[rs1+imm][0:7]`	zero-extends
lh	Load Half	`rd = M[rs1+imm][0:15]`	zero-extends
lw	Load Word	`rd = M[rs1+imm][0:31]`	
lbu	Load Byte (U)	`rd = M[rs1+imm][0:7]`	
lhu	Load Half (U)	`rd = M[rs1+imm][0:15]`	
sb	Store Byte	`M[rs1+imm][0:7]  = rs2[0:7]`	
sh	Store Half	`M[rs1+imm][0:15] = rs2[0:15]`	
sw	Store Word	`M[rs1+imm][0:31] = rs2[0:31]`	
beq	Branch ==	`if (rs1 == rs2) PC += imm`	
bne	Branch !=	`if (rs1 != rs2) PC += imm`	
blt	Branch $<$	`if (rs1 <  rs2) PC += imm`	
bge	Branch $\geq$	`if (rs1 >= rs2) PC += imm`	
bltu	Branch $<$ (U)	`if (rs1 <  rs2) PC += imm`	zero-extends
bgeu	Branch $\geq$ (U)	`if (rs1 >= rs2) PC += imm`	zero-extends
jal	Jump And Link	`rd = PC+4; PC += imm`	
jalr	Jump And Link Reg	`rd = PC+4; PC = rs1 + imm`	
lui	Load Upper Imm	`rd = imm << 12`	
auipc	Add Upper Imm to PC	`rd = PC + (imm << 12)`	
ecall	Environment Call	Transfer control to OS	
ebreak	Environment Break	Transfer control to debugger	

RV32I Base Integer Instructions (derived from https://github.com/jameslzhu/riscv-card).

RISC-V Software and Hardware

- Software
  - GNU compiler toolchain for RISC V (C compiler, assembler, linker)
  - Linux running on RISC V
  - RISC V emulation using qemu
  - Educational software (simulators)
- Hardware
  - Boards similar to Arduino
  - Boards similar to Raspberry Pi
  - BeagleV similar to Beagle Boards
  - ESP32-C embedded WLAN MCU
  - Open source FPGA implementations

RISC V processors are not yet performing at the speed of established mobile/desktop/server CPUs (arm, x86) that have been optimized over several decades. However, having an open ISA enables research on RISC-V technology and industrial implementations are catching up.

Below is a hello world program written in C that can be compiled to run on the emulsiV RISC-V simulator.

```
1  /*
2   * emulsiv/hello.c
3   */
4
5  #define EMULSIV_OUT_ADDR        0xc0000000
6
7  void putc(char c)
8  {
9      char *out = (char *) (EMULSIV_OUT_ADDR);
10     *out = c;
11 }
12
13 int print(char *s)
14 {
15     int i;
16
17     for (i = 0; s[i]; i++) {
18         putc(s[i]);
19     }
20     return i;
21 }
22
23 void main(void)
24 {
25     print("Hello Emulsiv!\n");
26 }
```

Comiling the C code using gcc 8.3.0 with optimizations turned off gives us the following RISC-V assembler code. Comments were added manually to make it easy to understand the assembler code and how it relates to the C code.

```
1  print:
2          addi    sp,sp,-48      ; sp = sp - 48      ; allocate stack frame
3          sw      ra,44(sp)      ; mem[sp+44] = ra   ; save the return address
4          sw      s0,40(sp)      ; mem[sp+40] = sfp  ; save frame pointer
5          addi    s0,sp,48       ; sfp = sp + 48     ; set new frame pointer
6          sw      a0,-36(s0)     ; mem[sfp-36] = a0  ; save argument s
7          sw      zero,-20(s0)   ; mem[sfp-20] = 0   ; initialize variable i
8          j       .L2            ; pc = .L2          ; jump to .L2
9  .L1:
10         lw      a5,-20(s0)     ; a5 = sfp[-20]     ; load i into a5
11         lw      a4,-36(s0)     ; a4 = sfp[-36]     ; load s into a4
12         add     a5,a4,a5       ; a5 = a4 + a5      ; calc s+i
13         lbu     a5,0(a5)       ; a5 = mem[a5]      ; load mem[s+i]
14         mv      a0,a5          ; a0 = a5           ; load argument
15         call    putc           ;                   ; call putc()
16         lw      a5,-20(s0)     ; a5 = sfp[-20]     ; load i
17         addi    a5,a5,1        ; a5 = a5 + 1       ; increment i
18         sw      a5,-20(s0)     ; mem[sfp-20] = a5  ; store i
19 .L2:
20         lw      a5,-20(s0)     ; a5 = sfp[-20]     ; load i into a5
21         lw      a4,-36(s0)     ; a4 = sfp[-36]     ; load s into a4
22         add     a5,a4,a5       ; a5 = a4 + a5      ; calc s+i
23         lbu     a5,0(a5)       ; a5 = mem[a5]      ; load mem[s+i]
24         bne     a5,zero,.L4    ; pc = .L1 if a5 != 0
25         lw      a5,-20(s0)     ; a5 = sfp[-20]     ; load i into a5
26         mv      a0,a5          ; a0 = a5           ; load return value
27         lw      ra,44(sp)      ; ra = mem[sp+44]   ; restore return address
28         lw      s0,40(sp)      ; sfp = mem[sp+40]  ; restore frame pointer
29         addi    sp,sp,48       ; sp = sp + 48      ; deallocate stack frame
30         jr      ra             ;                   ; jump to return address
```

Obviously, the assembler code is not very smart since it carries out redundant machine instructions. The code changes significantly once compiler optimizations are enabled.

Part VI

System Software

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

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

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

- explain the difference between compiled and interpreted programming languages;
- illustrate some optimizations performed by an optimizing compiler;
- describe the processing phases of a compiler;
- define multi-language and multi-target compilers;
- elaborate the purpose of an abstract syntax tree;
- use the backus-naur form to define some simple formal languages;
- outline the basic working principle of an interpreter;
- enumerate the differences between virtual machines and emulators;
- name services and abstractions provided by an operating system;
- understand the process lifecycle on POSIX operating systems;
- explain filesystems concepts and file system operations;
- motivate the relevance of different inter-process communication services.

Section 26: 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, Rust, Go, Pascal, Fortran, Cobol, ...

Examples of (typically) interpreted programming languages:

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

Many (typically) interpreted languages are not executed by pure interpreters anymore these days. Modern interpreters often compile the source code into an intermediate byte-code format, which is executed by a byte-code interpreter. However, this internal compilation step is usually transparent for the user; the languages keep their highly interactive programming interface, and they do not require an explicit compilation step to be initiated by the programmer.

RISC-V machine code (gcc 12.2.0 without optimizations)

```
/* C source code */

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

    return c;
}
```

```
00000044 <main>:
  44:   fe010113              addi    sp,sp,-32
  48:   00812e23              sw      s0,28(sp)
  4c:   02010413              addi    s0,sp,32
  50:   00500793              li      a5,5
  54:   fef42623              sw      a5,-20(s0)
  58:   00200793              li      a5,2
  5c:   fef42423              sw      a5,-24(s0)
  60:   fec42703              lw      a4,-20(s0)
  64:   fe842783              lw      a5,-24(s0)
  68:   00f707b3              add     a5,a4,a5
  6c:   fef42223              sw      a5,-28(s0)
  70:   fe442783              lw      a5,-28(s0)
  74:   00078513              mv      a0,a5
  78:   01c12403              lw      s0,28(sp)
  7c:   02010113              addi    sp,sp,32
  80:   00008067              ret
```


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, which is stored in the shell variable `$?`. You should see the number `7` printed to the terminal.

The `objdump` utility can be used to disassemble the machine code generated by the `gcc` compiler. Disassembling means the translation of binary machine code into a mnemonic representation humans can read more easily. Here is a description of the disassembled machine instructions:

```
main:
        addi    sp,sp,-32       ; allocate a new stack frame
        sw      s0,28(sp)       ; save old frame pointer
        addi    s0,sp,32        ; setup the new frame pointer
        li      a5,5            ; load 5 into register a5
        sw      a5,-20(s0)      ; save a5 in the stack frame
        li      a5,2            ; load 2 into register a5
        sw      a5,-24(s0)      ; save a5 in the stack frame
        lw      a4,-20(s0)      ; load 5 from the stack frame into a4
        lw      a5,-24(s0)      ; load 2 from the stack frame into a5
        add     a5,a4,a5        ; add a4 and a5 and place the result in a5
        sw      a5,-28(s0)      ; save a5 in the stack frame
        lw      a5,-28(s0)      ; load a5 from the stack frame
        mv      a0,a5           ; copy a5 to a2 (a2 returns the result)
        lw      s0,28(sp)       ; restore the old frame pointer
        addi    sp,sp,32        ; deallocate the stack frame
        jr      ra              ; return from function call
```

RISC-V machine code (gcc 12.2.0 with optimizations)

```
/* C source code */

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

    return c;
}
```

```
00000044 <main>:
  44:   00700513                li      a0,7
  48:   00008067                ret
```

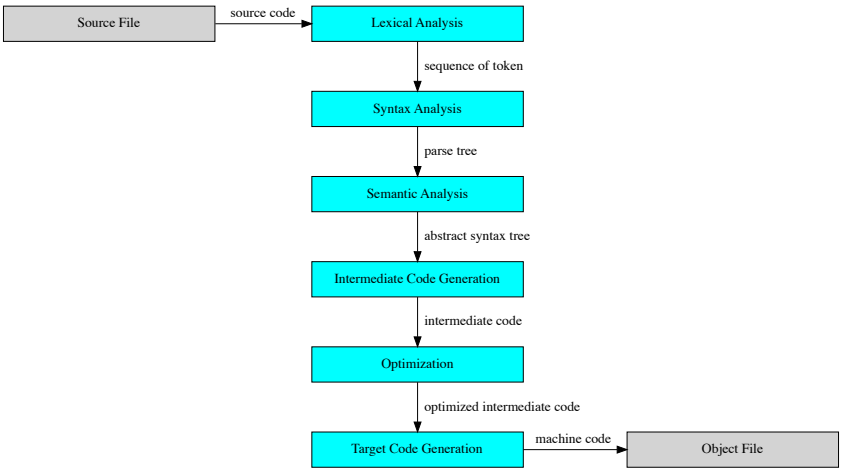

With optimization enabled, the compiler detected that the expression only depends on constants and thus the expression can be evaluated at compile time instead of runtime. Furthermore, the compiler determined that the variables `a`, `b`, and `c` are not used outside of the function `main()` and hence it is not necessary to store any values in the stack frame and since no other function is called from `main()`, it is not necessary to have a stack frame at all.

As a result, the function now translates into two instructions (8 bytes of machine code): one to load the constant 7 into register `a0` and one to return from the function. The previous version used 16 instructions (64 bytes of machine code).

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

Most of the time, compilers optimize for speed but it is also possible to ask the compiler to optimize for space. In this example, we got both a speed and a size improvement. In some cases, however, machine code may become longer in order to achieve better execution time (e.g., inlining of functions or loop unrolling).

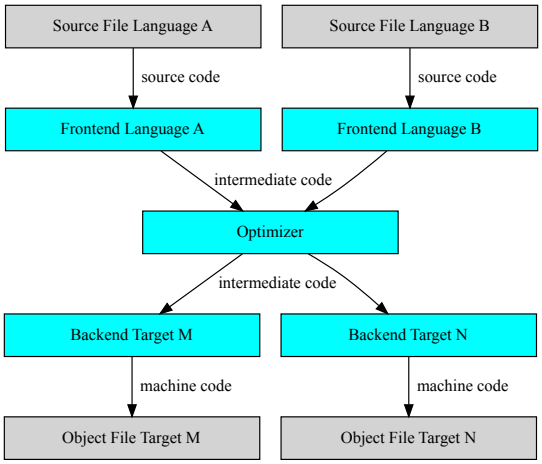
Phases of a Compiler



A compiler translates source code to machine code in several phases:

1. The *lexical analysis* phase converts a sequence of characters into a sequence of tokens. The tokens can have attributes, such as the name of an identifier or the value of a constant. The format of the different tokens is often specified using regular expressions.
2. The *syntax analysis* phase turns the sequence of tokens into a parse tree. This is done by matching the sequence of tokens against syntax rules. The syntactic structure of programming languages is commonly defined using a formal grammar, which is a set of rules defining the structure of syntactically correct programs. A program is syntactically correct if the syntax analysis is able to derive a syntax tree conforming to the grammar of the programming language. (A syntactically correct program is considered to be a word of the language of all syntactically correct programs.)
3. The *semantic analysis* phase adds semantic information to the parse tree, resulting in an abstract syntax tree. Semantic annotations concern for example type information and the determination whether the program is using types correctly.
4. The *intermediate code generation* phase converts the abstract syntax tree into an intermediate code representation. The intermediate code is abstract and not specific to any target machine languages. For example, intermediate code may assume that there is an arbitrary large number of registers.
5. The *optimization* phase optimizes the intermediate code. Optimizations include the removal of unreachable code, inlining of short frequently used code, unrolling of loops, moving invariant code out of loops, evaluating constant expressions, and many more things.
6. The *target code generation* phase converts the intermediate code into code for the target machine languages. Compilers often generate assembler code that is finally converted by an assembler into machine code. During the target code generation phase, the final allocation of registers or the exact format of stack frames is decided. Some optimizations that are specific to a given target machine language happen at this stage.

Multi-language and Multi-target Compiler



Multi-language compiler support multiple language frontends that produce a common abstract syntax tree. Multi-language compiler enable the reuse of the compiler backend phases, most importantly the code optimization and code generation phases. The construction of multi-language compiler is, however, non-trivial since the intermediate formats need to be able to express all features of all programming languages supported.

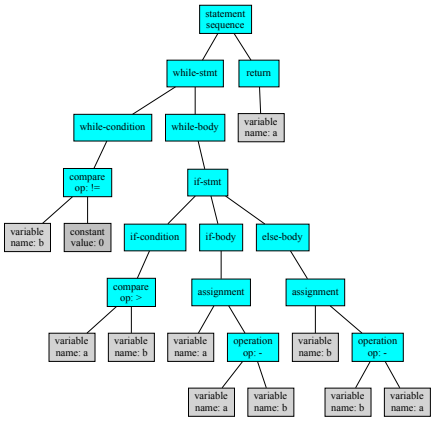
Multi-target compiler generate code for multiple target machine architectures. This can be achieved by integrating code generators for different machine languages. Modern compiler usually support multiple targets.

Two popular generic compiler infrastructures are the GNU compiler collection (gcc) provided by the GNU project and the LLVM compiler infrastructure provided by the LLVM Project. The GNU compiler project started around 1987, the LLVM project is much more recent, it started around 2004 [14].

Further online information:

- **Wikipedia**: Compiler
- **Web**: GNU Compiler Collection
- **Web**: LLVM Compiler Infrastructure

Abstract Syntax Tree Example



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

```
int gcd(int a, int b)
{
    while (b != 0) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```


From a compiler perspective, programmers are simply busy people writing trees down in a linear fashion...

Backus-Naur-Form and Formal Languages

The syntax of programming languages is commonly 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 $\Sigma = \{0, 1, \ldots, 9, x, y, z, +, *, (,)\}$.

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

```
    <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 valid word of the language. This is used by compilers to test whether a given program text is a (syntactically) valid word of the language.

The BNF of the Haskell language can be found in Section 10.5 of the Haskell 2010 report [16].

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 which expression to provide for the next read. The read–eval–print loop is more interactive than the edit-compile-run-debug cycle of compiled languages.

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

Compiler and Interpreter

```
[1] Source Code --> Interpreter

[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 online information:

- **Wikipedia**: Register machine
- **Wikipedia**: Stack machine

Virtual Machines and Emulators

Definition (virtual machine)

A virtual machine (VM) replicates the computer architecture and functions of the underlying real physical computer.

Definition (emulator)

An emulator emulates the functions of one computer system (the guest system) on another computer system (the host system) where the emulated guest system can be different from the underlying host system.

- Terminology is not very consistent, people commonly refer to the software running on an emulator as a virtual machine.

Virtual machines were invented in the 1970s (mainframes) and reinvented in the 1990s (personal computers). Virtual machines and emulators have facilitated the development of cloud computing since they are easy to start / stop / clone / migrate and they separate the software implementing services form the underlying hardware.

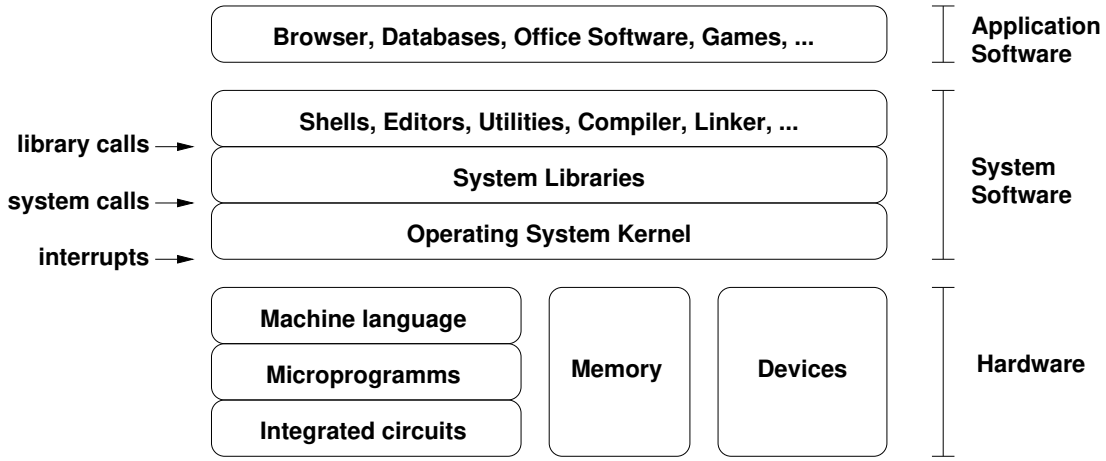
- The software virtualizing the underlying hardware is called a hypervisor.
- Full virtualization: Virtual machines run a complete operating system inside of the virtual machine. The hypervisor virtualizes all aspects of the underlying hardware (e.g., VmWare).
- Operating-system level virtualization: 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 online information:

- **Wikipedia**: Emulator
- **Wikipedia**: Virtual machine

Section 27: Operating Systems

Hardware vs. System Software vs. Application Software



The operating system kernel provides services to application programs. Applications use services provided by the kernel by making system calls. The operating system kernel controls all hardware components and mediates between the hardware and application programs. The kernel executes at a different privilege level and uses hardware assisted mechanisms to isolate applications from each other. Application programs are written against the system call interface (and associated libraries). Some 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.

Below is a hello world program in RISC-V assembler language using Linux system calls:

```
/*
 * Writes "Hello World" to the console using only system calls. Runs on
 * rv64 Linux only. To assemble and run:
 *
 * gcc -c hello-rv64-syscall.s && ld -static hello-rv64-syscall.o && ./a.out
 */

        .global _start
        .text
_start:
        # write(1, message, 12)
        addi    a0, zero, 1             # file handle 1 is stdout
        la      a1, message             # address of string to write
        addi    a2, zero, 12            # number of bytes to write
        addi    a7, zero, 64            # system call 64 is write
        ecall                           # invoke write system call
        # exit(0)
        addi    a0, zero, 0             # we want return code 0
        addi    a7, zero, 93            # system call 93 is exit
        ecall                           # invoke exit system call
message:
        .ascii  "Hello World\n"
```

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)

Below is a hello world in RISC-V assembler language using a Linux system library function:

```
/*
 * Writes "Hello World" to the console using a C library. Runs on Linux
 * rv64 Linux any other rv64 system that does not use underscores for
 * symbols in its C library. To assemble and run:
 *
 * gcc -static hello-rv64-libc.s && ./a.out
 */

        .global main
        .text
main:                                   # called by C library's startup code
        addi    sp, sp, -16             # allocate stack frame
        sd      ra, 8(sp)               # save return address
        sd      s0, 0(sp)               # save frame pointer
        addi    s0, sp, 16              # establish new frame pointer
        la      a0, message             # first parameter passed in a0
        jal     puts                    # call puts(message)
        addi    a0, zero, 0             # we want return code 0
        ld      ra, 8(sp)               # restore return address
        ld      s0, 0(sp)               # restore frame pointer
        addi    sp, sp, 16              # deallocate stack frame
        ret                             # return to C library code
message:
        .asciz "Hello World"            # asciz puts a 0 byte at the end
```

Further online information:

- **Wikipedia**: Operating system

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 code.

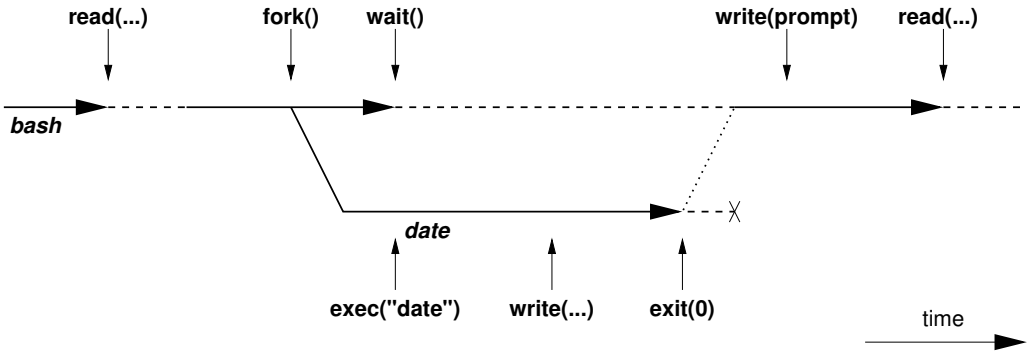
Relevant Linux command line tools to inspect the processes running on a system:

- 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.

Further online information:

- **Wikipedia**: Process

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 then waits for the clone (child process) to finish by invoking the `wait()` system call. The child process uses the `exec()` system call to replace the current process image with the process image of the `date` program. The child process finally exits by calling `exit()`. The process stays around until the exit code has been delivered to the parent process.

Description of the system calls:

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

OS Abstraction #1: Processes and Process Lifecycle

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



```
1  module Main where
2
3  import System.IO
4  import System.Exit (exitSuccess)
5  import System.Posix.Process (forkProcess, getProcessStatus, executeFile)
6  import Control.Monad (forever, when)
7
8  runCmd :: [String] -> IO ()
9  runCmd [] = return ()
10 runCmd args = do
11     pid <- forkProcess $ executeFile (head args) True (tail args) Nothing
12     getProcessStatus True False pid
13     return ()
14
15 exitCmd :: IO ()
16 exitCmd = do
17     exitSuccess
18
19 nextCmd :: IO ()
20 nextCmd = do
21     putStr "hsh > "
22     done <- isEOF
23     when done exitCmd
24     line <- getLine
25     when (line == "exit") exitCmd
26     runCmd (words line)
27
28 main :: IO ()
29 main = do
30     hSetBuffering stdout NoBuffering    -- make stdout unbuffered
31     forever nextCmd
```

OS Abstraction #2: File Systems

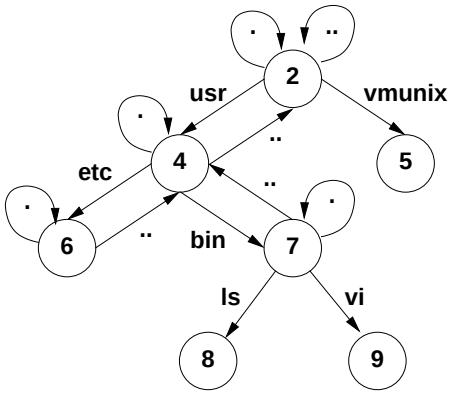
- Files are persistent containers for the storage of data
- Unstructured files contain a sequence of bytes
- Applications interpret the content of a file in a specific way
- Files also have meta data (owner, permissions, timestamps)
- Hierarchical file systems use directories to organize files into a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

- **Wikipedia**: File systems

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


```haskell
module Main where

import Control.Monad
import System.Environment
import System.Directory
import System.FilePath.Posix

ls :: FilePath -> IO ()
ls path = do
     dirExists <- doesDirectoryExist path
     when dirExists $ do
         all <- filter (`notElem` [".", ".."]) <$> getDirectoryContents path
         mapM_ putStrLn all
     unless dirExists $ putStrLn path

main :: IO ()
main = do
    args <- getArgs
    mapM_ ls args
```

OS Abstraction #3: Communication

- Primitives to support 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 an abstraction for communication between processes over the Internet
  - Internet communication starts with resolving names to (numeric) addresses
  - In many cases, a connection is then established between a client and a server
  - Data exchanges require encoding and decoding functions to achieve interoperability
  - Network protocols implemented in the kernel handle the communication flow


```haskell
{-# LANGUAGE OverloadedStrings #-}

module Main where

import qualified Control.Exception as E
import qualified Data.ByteString.Char8 as C
import Network.Socket
import Network.Socket.ByteString (recv, sendAll)

tcpClient :: HostName -> ServiceName -> (Socket -> IO a) -> IO a
tcpClient host port client = withSocketsDo $ do
    addr <- resolve
    E.bracket (open addr) close client
  where
    resolve = do
        let hints = defaultHints { addrSocketType = Stream }
        head <$> getAddrInfo (Just hints) (Just host) (Just port)
    open addr = do
        sock <- socket (addrFamily addr) (addrSocketType addr) (addrProtocol addr)
        connect sock $ addrAddress addr
        return sock

httpGet :: HostName -> ServiceName -> IO ()
httpGet host port = tcpClient host port $ \s -> do
    sendAll s $ C.pack ("GET / HTTP/1.0\r\nHost:" ++ host ++ "\r\n\r\n")
    msg <- recv s 1024
    C.putStrLn msg

main :: IO ()
main = do
    httpGet "cnds.jacobs-university.de" "http"
```

Further online information:

- **Wikipedia**: Inter-process communication

Part VII

Software Correctness

We generally want our software to be correct. In this part, we define software correctness and we study a classic technique, called the Floyd-Hoare logic, to prove the correctness of programs written in a simple imperative programming language.

Once we have developed the necessary ideas to prove the correctness of programs, we will investigate ways to automate some of the software verification process. The treatment of Floyd-Hoare triples and Floyd-Hoare logic is largely based on Mike Gordon's execellent "Background reading on Hoare Logic".

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

- recall the difference between partial and total correctness;
- articulate Floyd-Hoare triples and preconditions and postconditions;
- elaborate the role and importance of formal software specifications;
- describe the rules of the Hoare logic for program verification;
- illustrate precondition strengthening and postcondition weakening;
- explain the relevance of the weakest precondition and the strongest postcondition;
- sketch how annotations help to (partially) automate software verification;
- demonstrate how loop invariants are used for partial correctness proofs;
- outline how loop variants are used for total correctness proofs..

Section 28: Software Specification

Formal Specification and Verification

Definition (formal specification)

A *formal specification* uses a formal (mathematical) notation to provide a precise definition of what a program should do.

Definition (formal verification)

A *formal verification* uses logical rules to mathematically prove that a program satisfies a formal specification.

- For many non-trivial problems, creating a formal, correct, and complete specification is a problem by itself.
- A bug in a formal specification leads to programs with verified bugs.

Processors (CPUs) are designed using hardware description languages and we commonly assume that processors are correct. However, efficient designs introduce significant complexity and this increases the likelihood of introducing bugs. Fixing design flaws after silicon has been produced and deployed is very expensive. A classic example is the Intel Pentium FDIV bug discovered in 1994. Many more processor design flaws have been discovered since then.

Modern compilers are also very complex software designs, but still they tend to be super reliable for most users. One reason is that they are tested extensively and serious bugs can be fixed relatively quickly. This is a fundamental difference to systems that are difficult to fix once deployed. Creating certified correct compilers is still a very complex undertaking but required in some parts of the industry where safety-critical code must be written.

Further online information:

- **Wikipedia**: Pentium FDIV bug

Floyd-Hoare Triple

Definition (hoare triple)

Given a state that satisfies precondition P, executing a program C (and assuming it terminates) results in a state that satisfies postcondition Q. This is also known as the "Hoare triple":

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

- Invented by Charles Anthony ("Tony") Richard Hoare with original ideas from Robert Floyd (1969).
- Hoare triple can be used to specify what a program should do.
- Example:

$$\{X = 1\}\ X := X + 1\ \{X = 2\}$$

The classic publication introducing Hoare logic is [7]. Tony Hoare has made several other notable contributions to computer science: He invented the basis of the Quicksort algorithm (published in 1962) and he has developed the formalism Communicating Sequential Processes (CSP) to describe patterns of interaction in concurrent systems (published in 1978).

P and Q are logcial expressions on program variables. They are written using standard mathematical notation and logical operators. The predicate P defines the subset of all possible states for which a program C is defined. Similarly, the predicate Q defines the subset of all possible states for which the program's result is defined.

It is possible that different programs satisfy the same specification:

$$\{X = 1\}\ Y := 2\ \{X = 1 \wedge Y = 2\}$$

$$\{X = 1\}\ Y := 2 * X\ \{X = 1 \wedge Y = 2\}$$

$$\{X = 1\}\ Y := X + X\ \{X = 1 \wedge Y = 2\}$$

Partial Correctness and Total Correctness

Definition (partial correctness)

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

Definition (total correctness)

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

The distinction between partial correctness and total correctness is of fundamental importance. Total correctness requires termination, which is generally impossible to prove in an automated way as this would require to solve the famous halting problem. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

A definition of the form $\{P\}\ C\ \{Q\ \}$ usually provides a partial correctness specification. We use the notation $[\ P\ ]\ C\ [\ Q\ ]$ for a total correctness specification.

Further online information:

- **YouTube**: How This One Question Breaks Computers

Hoare Notation Conventions

1. The symbols V, V_1, ..., V_n stand for arbitrary variables. Examples of particular variables are X, Y, R etc.
2. The symbols E, E_1, ..., E_n stand for arbitrary expressions (or terms). These are expressions like $X + 1$, $\sqrt{2}$ etc., which denote values (usually numbers).
3. The symbols S, S_1, ..., S_n stand for arbitrary statements. These are conditions like $X < Y$, $X^2 = 1$ etc., which are either true or false.
4. The symbols C, C_1 , ..., C_n stand for arbitrary commands of our programming language; these commands are described on the following slides.

- We will use lowercase letters such as x and y to denote auxiliary variables (e.g., to denote values stored in variables).

We are focusing in the following on a purely imperative programming model where a global set of variables determines the current state of the computation. A subset of the variables are used to provide the input to an algorithm and another subset of the variables provides the output of an algorithm.

Note that we talk about a programming language consisting of commands and we use the term statements to refer to conditions. This may be a bit confusing since programming languages often call our commands statements and they may call our statements conditions.

Hoare Assignments

- Syntax: $V := E$
- Semantics: The state is changed by assigning the value of the expression E to the variable V. All variables are assumed to have global scope.
- Example: $X := X + 1$

Hoare Skip Command

- Syntax: *SKIP*
- Semantics: Do nothing. The state after executing the *SKIP* command is the same as the state before executing the *SKIP* command.
- Example: *SKIP*

The $SKIP$ command does nothing. It is still useful since it allows us to construct a single conditional command.

Hoare Command Sequences

- Syntax: $C_1; \ldots; C_n$
- Semantics: The commands $C_1, \ldots, C_n$ are executed in that order.
- Example: $R := X; X := Y; Y := R$

The example sequence shown above swaps the content of X and Y. Note that it has a side-effect since it also assigns the initial value of X to R. A specification of the swap program as a Floyd-Hoare triple would be the following:

$$\{ X = x \wedge Y = y \}\ R := X; X := Y; Y := R\ \{ X = y \wedge Y = x \}$$

Since the program does not involve any loops, we can easily specify total correctness as well:

$$[X = x \wedge Y = y]\ R := X; X := Y; Y := R\ [X = y \wedge Y = x]$$

Hoare Conditionals

- Syntax: *IF S THEN* C_1 *ELSE* C_2 *FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Example: *IF* $X < Y$ *THEN* $M := Y$ *ELSE* $M := X$ *FI*

Note that we can use $SKIP$ to create conditional statements without a $THEN$ or $ELSE$ branch:

IF S THEN C ELSE SKIP FI

IF S THEN SKIP ELSE C FI

Hoare While Loop

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- Example: *WHILE* $\neg(X = 0)$ *DO* $X := X - 2$ *OD*

Our notation uses a convention that was popular in the 1970s to denote the end of a programming language construct by repeating a keyword with the letters reversed. An early programming language using this notation was Algol 68. You find similar syntactic ideas in Bourne shells (if / fi, case / esac). More modern languages tend to use curly braces instead or they require suitable indentation to find the end of a command sequence.

Termination can be Tricky

```
1: function COLLATZ(X)
2:     while X > 1 do
3:         if (X%2) ≠ 0 then
4:             X ← (3 · X) + 1
5:         else
6:             X ← X/2
7:         end if
8:     end while
9:     return X
10: end function
```

- Collatz conjecture: The program will eventually return the number 1, regardless of which positive integer is chosen initially.

This program calculates the so called Collatz sequence. The Collatz conjecture says that no matter what value of $n \in \mathbb{N}$ you start with, the sequence will always reach 1. For example, starting with $n = 12$, one gets the sequence $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$.

Further online information:

- **Wikipedia**: Collatz conjecture
- **YouTube**: The Simplest Math Problem No One Can Solve - Collatz Conjecture

Specification can be Tricky

- Specification for the maximum of two variables:

$$\{\mathbf{T}\}\ C\ \{\ Y = max(X, Y)\ \}$$

- C could be:

```
IF X > Y THEN Y := X ELSE SKIP FI
```

- But C could also be:

```
IF X > Y THEN X := Y ELSE SKIP FI
```

- And C could also be:

```
Y := X
```

- Use auxiliary variables x and y to associate Q with P:

$$\{\ X = x \wedge Y = y\ \}\ C\ \{\ Y = max(x, y)\ \}$$

Obviously, multiple programs can satisfy a given specification:

$$\{\ X = 1\ \}\ Y := 2\ \{\ X = 1 \wedge Y = 2\ \}$$

$$\{\ X = 1\ \}\ Y := X + 1\ \{\ X = 1 \wedge Y = 2\ \}$$

$$\{\ X = 1\ \}\ Y := 2 * X\ \{\ X = 1 \wedge Y = 2\ \}$$

A slightly more complex example (factorial):

Precondition: $\{X > 0 \wedge X = x\}$
1: $F := 1$
2: **while** $X > 0$ **do**
3: $\quad F := F \cdot X$
4: $\quad X := X - 1$
5: **od**
Postcondition: $\{F = x!\}$

Section 29: Software Verification

Floyd-Hoare Logic

- Floyd-Hoare Logic is a set of inference rules that enable us to formally proof partial correctness of a program.
- If S is a statement, we write $\vdash S$ to mean that S has a proof.
- The axioms of Hoare logic will be specified with a notation of the following form:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

- The conclusion S may be deduced from $\vdash S_1, \ldots, \vdash S_n$, which are the hypotheses of the rule.
- The hypotheses can be theorems of Floyd-Hoare logic or they can be theorems of mathematics.

So far we have discussed the formal specification of software using preconditions and postconditions and we have introduced a simple imperative programming language consisting essentially of variables, expressions and variable assignments, a conditional command, a loop command, and command sequences. The next step is to define inference rules that allow us to make inferences over the commands of this simple programming language. This will give us a formal framework to prove that a program processing input satisfying the precondition will produce a result satisfying the postcondition.

Floyd-Hoare logic is a deductive proof system for Floyd-Hoare triples. It can be used to extract verification conditions (VCs), which are proof obligations or proof subgoals that must be proven so that $\{ P \}\, C \,\{ Q \}$ is true.

Precondition Strengthening

- If P implies P' and we have shown $\{P'\}\ C\ \{Q\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash P \to P', \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

- Example: Since $\vdash X = n \to X + 1 = n + 1$, we can strengthen

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

to

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

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

Precondition strengthening applied to the assignment axiom gives us a triple that feels more intuitive. But keep in mind that $\vdash \{\ X = n\ \}\ X := X + 1\ \{\ X = n + 1\ \}$ has been derived by combining the assignment axiom with precondition strengthening.

Postcondition Weakening

- If Q' implies Q and we have shown $\{P\}\ C\ \{Q'\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \quad \vdash Q' \to Q}{\vdash \{P\}\ C\ \{Q\}}$$

- Example: Since $X = n + 1 \to X > n$, we can weaken

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

to

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

The postcondition Q is weaker than Q' ($Q' \to Q$) if the set of states $\{s|s \vdash Q'\} \subseteq \{s|s \vdash Q\}$.

Weakest Precondition

Definition (weakest precondition)

Given a program C and a postcondition Q, the *weakest precondition* $wp(C, Q)$ denotes the largest set of states for which C terminates and the resulting state satisfies Q.

Definition (weakest liberal precondition)

Given a program C and a postcondition Q, the *weakest liberal precondition* $wlp(C, Q)$ denotes the largest set of states for which C leads to a resulting state satisfying Q.

- The "weakest" precondition P means that any other valid precondition implies P.
- The definition of $wp(C, Q)$ is due to Dijkstra (1976) and it requires termination while $wlp(C, Q)$ does not require termination.

In Hoare Logic, we can usually define many valid preconditions. For example, all of the following are valid Hoare triples:

$$\vdash \{\ X = 1\ \}\ X := X + 1\ \{\ X > 0\ \}$$

$$\vdash \{\ X > 0\ \}\ X := X + 1\ \{\ X > 0\ \}$$

$$\vdash \{\ X > -1\ \}\ X := X + 1\ \{\ X > 0\ \}$$

Obviously, the second preconditions is weaker than the first since $X = 1$ implies $X > 0$. With a similar argument, the third precondition is weaker than the second since $X > 0$ implies $X > -1$. How does the precondition $X = 0$ compare to the second and third alternative?

The weakest liberal precondition for $X := X + 1$ and the postcondition $X > 0$ is:

$$wlp(X := X + 1, X > 0) = (X > -1)$$

Since we can assume that the assignment always terminates in this specific case, we have:

$$wp(X := X + 1, X > 0) = wlp(X := X + 1, X > 0) = (X > -1)$$

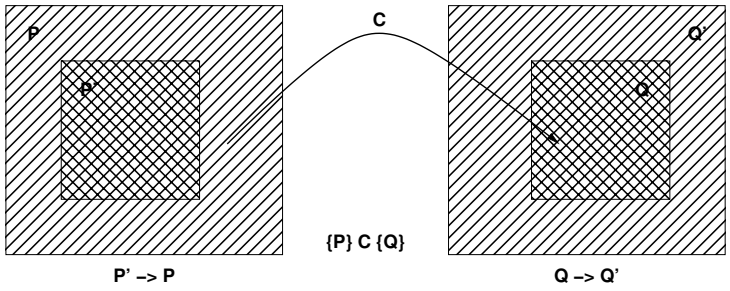
Strongest Postcondition

Definition (stronges postcondition)

Given a program C and a precondition P, the *strongest postcondition* $sp(C, P)$ has the property that $\vdash \{ P \} C \{ sp(C, P) \}$ and for any Q with $\vdash \{ P \} C \{ Q \}$, we have $\vdash sp(C, P) \to Q$.

- The "strongest" postcondition Q means that any other valid postcondition is implied by Q (via postcondition weakening).

Our goal is to find the weakest precondition and the strongest postcondition. This translates into finding the largest set of states that are valid inputs of the computation C and the smallest set of possible result states. Finding $P = wp(C, Q)$ makes it easy to cover any P' for which $P' \to P$ holds. Similarly, finding $Q = sp(C, P)$ makes it easy to cover any Q' for which $Q \to Q'$ holds.



Assignment Axiom

- Let $P[E/V]$ (P with E for V) denote the result of substituting the expression E for all occurances of the variable V in the statement P.
- An assignment assigns a variable V an expression E:

$$\vdash \{ P[E/V] \} V := E \{ P \}$$

- Example:

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

The assignment axiom kind of works backwards. In the example, we start with P, which is $\{X = n+1\}$. In P, we substitute E, which is $X + 1$, for V, which is X. This gives us $\{X + 1 = n + 1\}$.

Note that the term E is evaluated in a state where the assignment has not yet been carried out. Hence, if a statement P is true after the assignment, then the statement obtained by substituting E for V in P must be true before the assignment.

Two common erroneous intuitions:

1. $\vdash \{P\}\ V := E\ \{P[V/E]\}$

   This has the consequence $\vdash \{ X = 0 \} X := 1 \{ X = 0 \}$ since $X = 0[X/1]$ is equal to $X = 0$ (since 1 does not occur in $X = 0$).

2. $\vdash \{P\}\ V := E\ \{P[E/V]\}$

   This has the consequence $\vdash \{ X = 0 \} X := 1 \{ 1 = 0 \}$ since one would substitute X with 1 in $X = 0$.

Warning: An important assumption here is that expressions have no side effects that modify the program state. The assignment axiom depends on this property. (Many real-world programming languages, however, do allow side effects.) To see why side effects cause problems, consider an expression $(C; E)$ that consists of a command C and an expression E, e.g. $(Y := 1; 2)$. With this, we would get $\vdash \{ Y = 0 \} X := (Y := 1; 2) \{ Y = 0 \}$ (the substitution would not affect Y).

Specification Conjunction and Disjunction

- If we have shown $\{ P_1 \} C \{ Q_1 \}$ and $\{ P_2 \} C \{Q_2\}$, then $\{ P_1 \wedge P_2 \} C \{ Q_1 \wedge Q_2 \}$ holds as well:

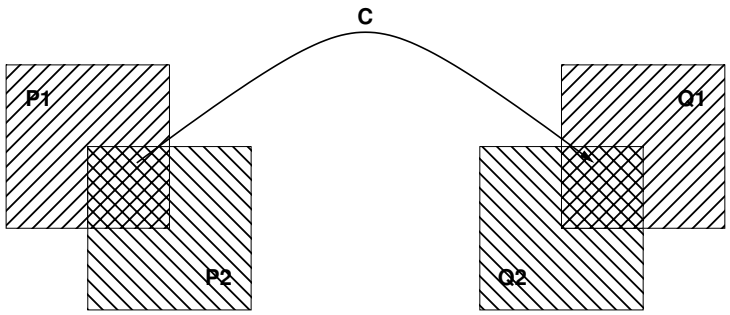
$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

- We get a similar rule for disjunctions:

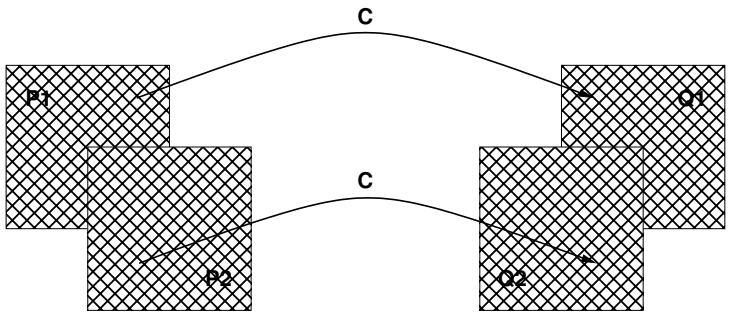
$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

- These rules allows us to prove $\vdash \{ P \} C \{ Q_1 \wedge Q_2 \}$ by proving both $\vdash \{ P \} C \{ Q_1 \}$ and $\vdash \{ P \} C \{ Q_2 \}$.

The rules can be easily understood by looking at the sets of states satisfying the predicates. The conjunction $P_1 \wedge P_2$ translates to the intersection of the corresponding sets of states and in the same way the conjunction $Q_1 \wedge Q_2$ translates to the intersection of the corresponding sets of states. The rule then follows directly from the fact that $\{ P_1 \wedge P_2 \} C \{ Q_1 \wedge Q_2 \}$ maps from the intersection to the intersection.



The disjunction $P_1 \vee P_2$ translates to the union of the sets of states associated with P_1 and P_2 and the disjunction $Q_1 \vee Q_2$ to the union of the sets of states associated with Q_1 and Q_2. In order to cover the entire space, it is necessary that both $\{ P_1 \} C \{ Q_1 \}$ and $\{ P_2 \} C \{ Q_2 \}$ are true.



Skip Command Rule

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

$$\frac{}{\vdash \{P\}\ SKIP\ \{P\}}$$

Sequence Rule

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

$$\frac{\vdash \{P\}\ C_1\ \{R\}, \quad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

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

$$\frac{\vdash \{P\}\ C_1\ \{R_1\}, \vdash \{R_1\}\ C_2\ \{R_2\}, \ \ldots, \vdash \{R_{n-1}\}\ C_n\ \{Q\}}{\vdash \{P\}\ C_1; C_2; \ldots; C_n\ \{Q\}}$$

Example (swapping two numbers):

Precondition: $\{\ X = x \wedge Y = y\ \}$
1: $R := X$
2: $X := Y$
3: $Y := R$
Postcondition: $\{\ X = y \wedge Y = x\ \}$

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

(i) $\vdash \{\ X = x \wedge Y = y\ \}\ R := X\ \{\ R = x \wedge Y = y\ \}$ (assignment axiom)

(ii) $\vdash \{\ R = x \wedge Y = y\ \}\ X := Y\ \{\ R = x \wedge X = y\ \}$ (assignment axiom)

(iii) $\vdash \{\ R = x \wedge X = y\ \}\ Y := R\ \{\ Y = x \wedge X = y\ \}$ (assignment axiom)

(iv) $\vdash \{\ X = x \wedge Y = y\ \}\ R := X; X := Y\ \{\ R = x \wedge X = y\ \}$ (sequence rule for (i) and (ii))

(v) $\vdash \{\ X = x \wedge Y = y\ \}\ R := X; X := Y; Y := R\ \{\ Y = x \wedge X = y\ \}$ (sequence rule for (iv) and (iii))

Conditional Command Rule

- Syntax: *IF S THEN* C_1 *ELSE* C_2 *FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Conditional Rule:

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \quad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ \{Q\}}$$

Consider the following specification and program (max):

Precondition: $\{\ X = x \wedge Y = y\ \}$
1: **if** $X \geq Y$ **then**
2: $\quad M := X$
3: **else**
4: $\quad M := Y$
5: **fi**
Postcondition: $\{\ M = max(x, y)\ \}$

In order to prove the partial correctness of this program, we have to prove the correctness of the two assignments under the statement $X \geq Y$ being either true or false. The application of the assignment axiom gives us the following two statements:

$$\{\ X = x \wedge Y = y \wedge X \geq Y\ \}\ M := X\ \{\ M = x \wedge X = x \wedge Y = y \wedge X \geq Y\ \}$$

$$\{\ X = x \wedge Y = y \wedge X < Y\ \}\ M := Y\ \{\ M = y \wedge X = x \wedge Y = y \wedge X < Y\ \}$$

The definition of $max(x, y)$ we are going to use is the following:

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

This gives us the following implications:

$$M = x \wedge X = x \wedge Y = y \wedge X \geq Y \rightarrow M = max(x, y)$$

$$M = y \wedge X = x \wedge Y = y \wedge X < Y \rightarrow M = max(x, y)$$

Postcondition weakening gives us:

$$\{\ X = x \wedge Y = y \wedge X \geq Y\}\ M := X\ \{M = max(x, y)\ \}$$

$$\{\ X = x \wedge Y = y \wedge X < Y\}\ M := Y\ \{M = max(x, y)\ \}$$

Applying the conditional rule, we get:

$$\{\ X = x \wedge Y = y\ \}\ \text{IF}\ X \geq Y\ \text{THEN}\ M := X\ \text{ELSE}\ M := Y\ \text{FI}\ \{\ M = max(x, y)\ \}$$

While Command Rule

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- While Rule:

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ \{P \wedge \neg S\}}$$

P is an invariant of C whenever S holds. Since executing C preserves the truth of P, executing C any numbner of times also preserves the truth of P.

Finding invariants is the key to prove the correctness of while loops. The invariant should

- say what has been done so far together with what remains to be done;
- hold at each iteration of the loop;
- give the desired result when the loop terminates.

Example (factorial):

Precondition: $\{\ Y = 1 \wedge Z = 0 \wedge X = x \wedge X \geq 0\ \}$
1: **while** $Z \neq X$ **do**
2: $\quad Z := Z + 1$
3: $\quad Y := Y * Z$
4: **od**
Postcondition: $\{\ Y = x!\ \}$

We need to find an invariant P such that:

- $\{\ P \wedge Z \neq X\ \}\ Z := Z + 1;\ Y := Y * Z\ \{\ P\ \}$ (while rule)
- $Y = 1 \wedge Z = 0 \rightarrow P$ (precondition strengthening)
- $P \wedge \neg(Z \neq X) \rightarrow Y = X!$ (postcondition weakening)

The invariant $Y = Z!$ serves the purpose:

- $Y = Z! \wedge Z \neq X \rightarrow Y \cdot (Z + 1) = (Z + 1)!$
  $\{\ Y \cdot (Z + 1) = (Z + 1)!\ \}\ Z := Z + 1\ \{\ Y \cdot Z = Z!\ \}$ (assignment axiom)
  $\{\ Y \cdot Z = Z!\ \}\ Y := Y * Z\ \{\ Y = Z!\ \}$ (assignment axiom)
  $\{\ Y = Z!\ \}\ Z := Z + 1;\ Y := Y * Z\ \{\ Y = Z!\ \}$ (sequence rule)
- $Y = 1 \wedge Z = 0 \rightarrow Y = Z!$ since $0! = 1$
- $Y = Z! \wedge \neg(Z \neq X) \rightarrow Y = X!$ since $\neg(Z \neq X)$ is equivalent to $Z = X$

Section 30: Automation of Software Verification

Proof Automation

- Proving even simple programs manually takes a lot of effort
- There is a high risk to make mistakes during the process
- General idea how to automate the proof:
  - (i) Let the human expert provide annotations of the specification (e.g., loop invariants) that help with the generation of proof obligations
  - (ii) Generate proof obligations automatically (verification conditions)
  - (iii) Use automated theorem provers to verify some of the proof obligations
  - (iv) Let the human expert prove the remaining proof obligations (or let the human expert provide additional annotations that help the automated theorem prover)
- Step (ii) essentially compiles an annotated program into a conventional mathematical problem.

Consider the following program:

Precondition: $\{\top\}$

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

Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

Annotations

- Annotations are required
  - (i) before each command C_i (with $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$, where C_i is not an assignment command and
  - (ii) after the keyword *DO* in a *WHILE* command (loop invariant)
- The inserted annotation is expected to be true whenever the execution reaches the point of the annotation.
- For a properly annotated program, it is possible to generate a set of proof goals (verification conditions).
- It can be shown that once all generated verification conditions have been proved, then $\vdash \{P\}\ C\ \{Q\}$.

We add suitable annotations:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\quad \{X = Y \cdot Q + R\}$
6: $\quad R := R - Y$
7: $\quad Q := Q + 1$
8: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

This should (ideally automatically) lead to the following proof obligations (verification conditions):

1. $\top \rightarrow (X = X \wedge 0 = 0)$
2. $(R = X \wedge Q = 0) \rightarrow (X = Y \cdot Q + R)$
3. $(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \wedge R < Y)$
4. $(X = Y \cdot Q + R \wedge Y \leq R) \rightarrow (X = Y \cdot (Q + 1) + (R - Y)$

Generation of Verification Conditions

- Assignment $\{P\}\ V := E\ \{Q\}$:
  Add verification condition $P \to Q[E/V]$.
- Conditions $\{P\}$ *IF* S *THEN* C_1 *ELSE* C_2 *FI* $\{Q\}$
  Add verification conditions generated by $\{P \wedge S\}\ C_1\ \{Q\}$ and $\{P \wedge \neg S\}\ C_2\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ \{R\}\ C_n\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{R\}$ and $\{R\}\ C_n\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ V := E\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{Q[E/V]\}$
- While loops $\{P\}$ *WHILE* S *DO* $\{R\}\ C$ *OD* $\{Q\}$
  Add verification conditions $P \to R$ and $R \wedge \neg S \to Q$
  Add verificiation conditions generated by $\{R \wedge S\}\ C\ \{R\}$

Starting with the annotated example:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\quad \{X = Y \cdot Q + R\}$
6: $\quad R := R - Y$
7: $\quad Q := Q + 1$
8: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

According to the second sequence rule, we have to generate VCs for the while loop and the sequence consisting of the initial assignments. The initial assignments reduce to $\top \to (X = X \wedge 0 = 0)$ as follows:

$$\{\top\}\ R := X;\ Q := 0\ \{R = X \wedge Q = 0\}$$
$$\{\top\}\ R := X\ \{R = X \wedge 0 = 0\}$$
$$\top \to (X = X \wedge 0 = 0)$$

The while loop rule gives us the following two VCs

$$(R = X \wedge Q = 0) \to (X = Y \cdot Q + R)$$
$$(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \to (X = Y \cdot Q + R \wedge R < Y)$$

and the VC generated as follows:

$$\{X = Y \cdot Q + R \wedge Y \leq R\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R\}$$
$$\{X = Y \cdot Q + R \wedge Y \leq R\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R\}$$
$$(X = Y \cdot Q + R \wedge Y \leq R) \to (X = Y \cdot (Q + 1) + (R - Y))$$

Total Correctness

- We assume that the evaluation of expressions always terminates.
- With this simplifying assumption, only *WHILE* commands can cause loops that potentially do not terminate.
- All rules for the other commands can simply be extended to cover total correctness.
- The assumption that expression evaluation always terminates is often not true. (Consider recursive functions that can go into an endless recursion.)
- We have so far also silently assumed that the evaluation of expressions always yields a proper value, which is not the case for a division by zero.
- Relaxing our assumptions for expressions is possible but complicates matters significantly.

If C does not contain any while commands, then we have the simple rule:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [1/4]

- Assignment axiom

$$\vdash [P[E/V]]\ V := E\ [P]$$

- Precondition strengthening

$$\frac{\vdash P \to P', \quad \vdash [P']\ C\ [Q]}{\vdash [P]\ C\ [Q]}$$

- Postcondition weakening

$$\frac{\vdash [P]\ C\ [Q'], \quad \vdash Q' \to Q}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [2/4]

- Specification conjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \wedge P_2]\ C\ [Q_1 \wedge Q_2]}$$

- Specification disjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \vee P_2]\ C\ [Q_1 \vee Q_2]}$$

- Skip command rule

$$\frac{}{[P]\ SKIP\ [P]}$$

Rules for Total Correctness [3/4]

- Sequence rule

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

- Conditional rule

$$\frac{\vdash [P \wedge S]\ C_1\ [Q], \quad \vdash [P \wedge \neg S]\ C_2\ [Q]}{\vdash [P]\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ [Q]}$$

Rules for Total Correctness [4/4]

- While rule

$$\frac{\vdash [P \wedge S \wedge E = n]\ C\ [P \wedge (E < n)], \quad \vdash P \wedge S \rightarrow E \geq 0}{\vdash [P]\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ [P \wedge \neg S]}$$

  E is an integer-valued expression
  n is an auxiliary variable not occuring in P, C, S, or E

- A prove has to show that a non-negative integer, called a *variant*, decreases on each iteration of the loop command C.

We show that the while loop in the following program terminates.

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: **while** $Y \leq R$ **do**
4: $\quad R := R - Y$
5: $\quad Q := Q + 1$
6: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

We apply the while rule with

$$P = Y > 0$$
$$S = Y \leq R$$
$$E = R$$

and we have to show the following to be true:

1. $[P \wedge S \wedge E = n]\ R := R - Y; Q := Q + 1\ [P \wedge (E < n)]$

   This follows from the following derivation:

$$[P \wedge S \wedge E = n]\ R := R - Y; Q := Q + 1\ [P \wedge (E < n)]$$
$$[Y > 0 \wedge Y \leq R \wedge R = n]\ R := R - Y; Q := Q + 1\ [Y > 0 \wedge (R < n)]$$
$$Y > 0 \wedge Y \leq R \wedge R = n \rightarrow Y > 0 \wedge (R < n)[Q + 1/Q][R - Y/R]$$
$$Y > 0 \wedge Y \leq R \wedge R = n \rightarrow Y > 0 \wedge ((R - Y) < n)$$

2. $P \wedge S \rightarrow E \geq 0$

   This follows from:

$$P \wedge S \rightarrow E \geq 0$$
$$Y > 0 \wedge Y \leq R \rightarrow R > 0$$

Generation of Termination Verification Conditions

- The rules for the generation of termination verificiation conditions follow directly from the rules for the generation of partial correctness verificiation conditions, except for the while command.
- To handle the while command, we need an additional annotation (in square brackets) that provides the variant expression.
- For while loops of the form $\{P\}$ *WHILE S DO* $\{R\}$ $[E]$ *C OD* $\{Q\}$ add the verification conditions

$$P \to R$$
$$R \wedge \neg S \to Q$$
$$R \wedge S \to E \geq 0$$

and add verificiation conditions generated by $\{R \wedge S \wedge (E = n)\}\ C\ \{R \wedge (E < n)\}$

Annotated example including the variant annotation for termination verification rule generation:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\quad \{X = Y \cdot Q + R\}$
6: $\quad [R]$
7: $\quad R := R - Y$
8: $\quad Q := Q + 1$
9: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

The while loop rule gives use the following termination VCs

$$(R = X \wedge Q = 0) \to (X = Y \cdot Q + R)$$
$$(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \to (X = Y \cdot Q + R \wedge R < Y)$$
$$(X = Y \cdot Q + R \wedge (Y \leq R)) \to R \geq 0$$

and the VC generated as follows:

$$\{X = Y \cdot Q + R \wedge Y \leq R \wedge R = n\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R \wedge R < n\}$$
$$\{X = Y \cdot Q + R \wedge Y \leq R \wedge R = n\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R \wedge R < n\}$$
$$(X = Y \cdot Q + R \wedge Y \leq R \wedge R = n) \to (X = Y \cdot (Q + 1) + (R - Y) \wedge (R - Y) < n)$$

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

$Y = 0$:

$$((X = R \wedge 0 \leq R \wedge R = n) \to (X = R \wedge R < n)) \to \bot$$

$Y < 0$:

$$((X = Y \cdot Q + R \wedge Y \leq R \wedge R = n) \to (X = Y \cdot (Q + 1) + (R - Y) \wedge (R - Y) < n)) \to \bot$$

Termination and Correctness

- Partial correctness and termination implies total correctness:

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

- Total correctness implies partial correctness and termination:

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

References

[1] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[2] Peter J. Denning, Matti Tedre, and Pat Yongpradit. Misconceptions about computer science. *Communications of the ACM*, 60(3), February 2017.

[3] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, Alis Technologies, November 2003.

[4] G. Futschek. Algorithmic Thinking: The Key for Understanding Computer Science. In *Informatics Education – The Bridge between Using and Understanding Computers, Lecture Notes in Computer Science 4226*. Springer, 2006.

[5] Zvi Galil. On improving the worst case running time of the boyer-moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, September 1979.

[6] R. Hammack. *Book of Proof*. 3 edition, 2019.

[7] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[8] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339, Clearswift Corporation, Sun Microsystems, July 2002.

[9] D. E. Knuth. *The Art of Computer Programming*, volume 1, Fundamental Algorithms. Addison Wesley, 3 edition, 1997.

[10] D. E. Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching. Addison Wesley, 2 edition, 1998.

[11] D. E. Knuth. *The Art of Computer Programming*, volume 4a, Combinatorial Algorithms. Addison Wesley, 1 edition, 2011.

[12] D. E. Knuth. *The Art of Computer Programming*, volume 2, Semi Numerical Algorithms. Addison Wesley, 3 edition, 2014.

[13] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1), February 1956.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[15] E. Lehmann, F.T. Leighton, and A.R. Meyer. *Mathematics for Computer Science*. MIT Open Courseware, 2019.

[16] S. Marlow. Haskell 2010 Language Report. Technical Report Haskell 2010, Haskell Community, 2010.

[17] James W. McGuffee. Defining computer science. *ACM SIGSE Bulletin*, 32(2), June 2020.

[18] SiFive Inc., University of California at Berkeley. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, 20191213 edition, December 2019.

[19] C.S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, February 1964.