Introduction to Computer Science '2020

Jürgen Schönwälder

Jacobs University Bremen

December 25, 2020

JACOBS
UNIVERSITY

https://cnds.jacobs-university.de/courses/ics-2020

Intended Learning Outcomes

- explain basic concepts such as the correctness and complexity of algorithms (including the big O notation);
- illustrate basic concepts of discrete math (sets, relations, functions);
- recall basic proof techniques and use them to prove properties of algorithms;
- explain the representation of numbers (integers, floats), characters and strings, and date and time;
- summarize basic principles of Boolean algebra and Boolean logic;
- describe how Boolean logic relates to logic gates and digital circuits;
- outline the basic structure of a von Neumann computer;
- explain the execution of machine instructions on a von Neumann computer;
- describe the difference between assembler languages and higher-level programming languages;
- define the differences between interpretation and compilation;
- illustrate how an operating system kernel supports the execution of programs;
- determine the correctness of simple programs;
- write simple programs in a pure functional programming language.

Topics and Timeline

- The "lecture" part of the module is structured into several parts:

I	Introduction	2 weeks
II	Discrete Mathematics	2 weeks
III	Number Systems, Units, Characters, Date and Time	2 weeks
IV	Boolean Algebra and Logic	3 weeks
V	Computer Architecture	2 weeks
VI	System Software	1 week
VII	Software Correctness	2 weeks

- The "programming" part of the module is introducing you to the functional programming language Haskell and runs over all 14 weeks

Assessment

- Module achievement (during the semester):
  - 50% of 10 (weekly) assignments correctly solved
  - 2 additional (weekly) assignments can be used to makeup points
  - Students without module achievement are not allowed to sit for the exam
  - Submit homework solutions regularly from the beginning
- Written examination (December 2020 and/or January 2021):
  - Duration: 120 min (closed book)
  - Scope: All intended learning outcomes of the module
- You can audit the course. To earn an audit, you have to pass a short oral interview about key concepts introduced in the course at the end of the semester.

Assignments

- We will post weekly homework assignments
- Assignments reinforce what has been discussed in class
- Assignments will be small individual assignments (but may take time to solve)
- Solving assignments will prepare you for the written examination
- Solutions must be submitted individually via Moodle
- Teaching assistants will review the assignments
- Assignments will tell you whether you understood the material
- Consider forming study groups. It helps to discuss questions and course material in study groups or to explore different directions to solve an assignment. However, solutions must be individual submissions. (Discuss the general problem in a study group, workout the details of the solution yourself.)

Study Groups

- I strongly suggest to form study groups.
- It helps to discuss questions and course material in study groups, in particular when you are getting stuck.
- Discussions in a study group can help more quickly understand what is demanded by a problem.
- Study group members may try different approaches to solve a problem and you can benefit from that.
- However, note that submissions must be individual solutions.
- It is OK to sketch a possible solution in a study group, then work out the details of the solution yourself.

Code of Academic Integrity

- Jacobs University has a “Code of Academic Integrity”
  - this is a document approved by the Jacobs community
  - you have signed it during enrollment
  - it is a law of the university, we take it seriously
- It mandates good behaviours from faculty and students and it penalizes bad ones:
  - honest academic behavior (e.g., no cheating)
  - respect and protect intellectual property of others (e.g., no plagiarism)
  - treat all Jacobs University members equally (e.g., no favoritism)
- It protects you and it builds an atmosphere of mutual respect
  - we treat each other as reasonable persons
  - the other’s requests and needs are reasonable until proven otherwise
  - if others violate our trust, we are deeply disappointed (may be leading to severe and uncompromising consequences)

Academic Integrity Committee (AIC)

- The Academic Integrity Committee is a joint committee by students and faculty.
- Mandate: to hear and decide on any major or contested allegations, in particular,
  - the AIC decides based on evidence in a timely manner,
  - the AIC makes recommendations that are executed by academic affairs,
  - the AIC tries to keep allegations against faculty anonymous for the student.
- Every member of Jacobs University (faculty, student, . . .) can appeal any academic integrity allegations to the AIC.

Cheating

- There is no need to cheat, cheating prevents you from learning
- Useful collaboration versus cheating:
  - You will be required to hand in your own original code/text/math for all assignments
  - You may discuss your homework assignments with others, but if doing so impairs your ability to write truly original code/text/math, you will be cheating
  - Copying from peers, books or the Internet is plagiarism unless properly attributed
- What happens if we catch you cheating?
  - We will confront you with the allegation (you can explain yourself)
  - If you admit or are silent, we impose a grade sanction and we notify the student records office
  - Repeated infractions are reported to the AIC for deliberation
- Note: Both active cheating (copying from others) and passive cheating (allowing others to copy) are penalized equally

Deadlines

- Deadlines will be strict (don't bother to ask for extensions)
- Make sure you submit the right document. We grade what was submitted, not what could have been submitted.
- Submit early — avoid last minute changes or software/hardware problems.
- Official excuses by the student records office will extend the deadlines, but not more than the time covered by the excuse.
- A word on medical excuses: Use them when you are ill. Do not use them as a tool to gain more time.
- You want to be taken serious if you are seriously ill. Misuse of excuses can lead to a situation where you are not taken too serious when you deserve to be taken serious.

Culture of Questions, Answers, and Explanations

- Answers to questions require an explanation even if this is not stated explicitly
    - A question like 'Does this algorithm always terminate?' can in principle be answered with 'yes' or 'no'.
    - We expect, however, that an explanation is given why the answer is 'yes' or 'no', even if this is not explicitly stated.
- Answers should be written in your own words
    - Sometimes it is possible to find perfect answers on Wikipedia or Stack Exchange or in good old textbooks.
    - Simply copying the answer of someone else is plagiarism.
    - Copying the answer and providing the reference solves the plagiarism issue but usually does not show that you understood the answer.
    - Hence, we want you to write the answer in your own words.
    - Learning how to write concise and precise answers is an important academic skill.

Culture of Interaction

- I am here to help you learn the material.
- If things are unclear, ask questions.
- If I am going too fast, tell me.
- If I am going too slow, tell me.
- Discussion in class is most welcome – don't be shy.
- Discussion in tutorials is even more welcome – don't be shy.
- If you do not understand something, chances are pretty high your neighbor does not understand either.
- Don't be afraid of asking teaching assistants or myself for help and additional explanations.

Study Material and Forums

- There is no required textbook.
- The slides and lecture notes are available on the course web page.
  `https://cnds.jacobs-university.de/courses/ics-2020`
- We will use the Jacobs Moodle system for assignments etc.
  `https://moodle.jacobs-university.de/`
- General questions should be asked on the Moodle forum.
  - Faster responses since many people can answer
  - Better responses since people can collaborate on the answer
- For individual questions, send me email or come to see me at my office (or talk to me after class or wherever you find me).

Tools

- You will need a computer to follow this course (any notebook will do)
- Get used to standard software tools:
  - Good and powerful editors such as `emacs` or `vim`
  - Unix-like operating systems such as `Linux` (e.g., Ubuntu)
  - Learn how to use a command interpreter (shell) like `bash`
  - Learn to write structured documents using LaTeX (great for typesetting math)
  - Learn how to maintain an agenda and TODO items (managing your time)
  - Get familiar with version control systems (e.g., `git`)
- Learn how to touch-type (typing without having to look at the keys)
- Learn how to maintain a healthy work life balance
  - Getting enough sleep is important for your brain to be effective
  - A workout may result in the idea missing if you are stuck on a problem

Links to Further Useful Information

- Sign up for the `cs-students@lists.jacobs-university.de` mailing list, a low volume list used primarily for announcements to all CS students:

  `http://lists.jacobs-university.de/mailman/listinfo`

- The handbooks can be found on the registrar's web page:

  `http://student-records.user.jacobs-university.de/undergraduate-program-handbooks/`

- The policies can be found on this web page:

  `https://www.jacobs-university.de/academic-policies`

Part 1: Introduction

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]

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?

Algorithmic Thinking

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

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

Section 2: Maze Generation Algorithms

Maze (33 x 11)

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

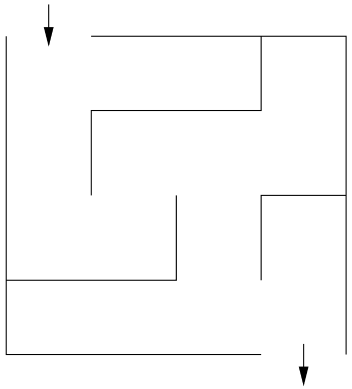
Problem Statement

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:

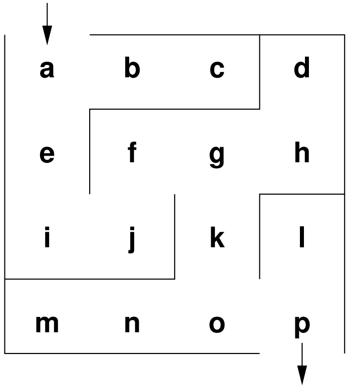
- How do we approach this problem?
- Are there other properties that make a maze a "good" or a "challenging" maze?



Hacking. . .

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?



Problem Formalization (2/3)

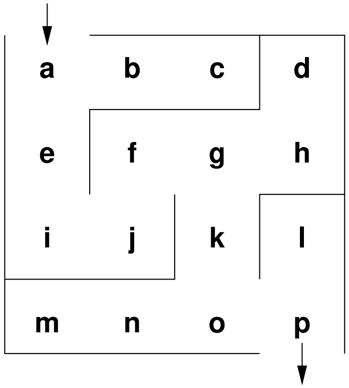
Lets try to formalize the problem in mathematical terms:

- 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).



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"

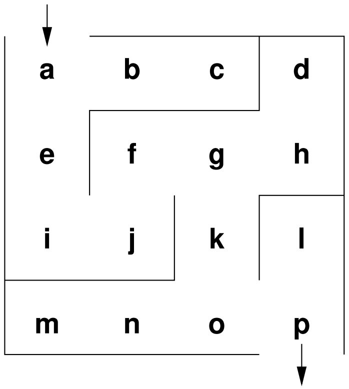
Problem Formalization (3/3)

Definition:

- A maze is a graph $G = (V, E)$ with two special nodes, the start node S and the exit node X.

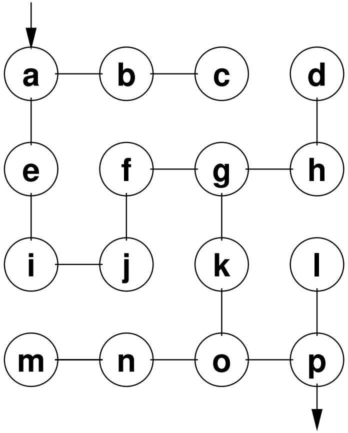
Interpretation:

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



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.



Mazes as Graphs (Good Mazes)

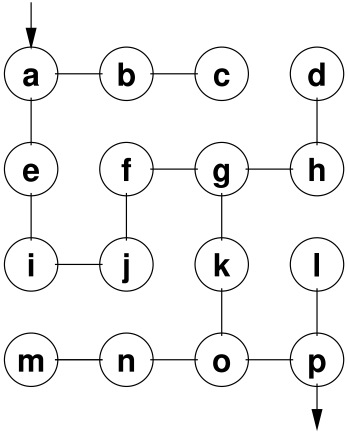
Recall, what is a good maze?

- 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.



Kruskal's Algorithm (1/2)

General approach:

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

Questions:

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

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.

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

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?

Problem Formalization

- Let Σ be a finite set, called an alphabet.
- Let k denote the number of elements in Σ.
- Let Σ^* be the set of all words that can be created out of Σ (Kleene closure of Σ).
- Let $t \in \Sigma^*$ be a (possible long) text and $p \in \Sigma^*$ be a (typically short) pattern.
- Let n denote the length of t and m denote the length of p.
- We assume that $n \gg m$.
- Find the first occurance of p in t.

Naive String Search

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

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

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

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 $n \cdot m$.

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
```

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?

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
```

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
```

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?

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.

Section 4: Complexity, Correctness, Engineering

Complexity of Algorithms

- Questions:
  - Which maze generation algorithm is faster?
  - Is there a fastest maze generation algorithm?
  - What happens if we consider mazes of different sizes or dimensions?
  - Instead of measuring execution time (which depends on the speed of the computer hardware), can we have a more neutral notion of "fast"?
- Computer science is about analyzing the complexity of algorithms.
- Complexity is an abstract measure of computational effort (time complexity) and memory usage (space complexity).

Performance and Scaling

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

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

Big O Notation (Landau Notation)

Definition (asymptotically bounded)

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

Definition (Landau Sets)

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

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

Commonly Used Landau Sets

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

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

Theorem (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

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

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

Landau Set Rules

Theorem (Landau Set Computation Rules)

We have the following computation rules for Landau sets:

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

Examples:

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

Correctness of Algorithms and Programs

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

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.

Deterministic Algorithms

Definition (deterministic algorithm)

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

- Some factors that make an algorithm non-deterministic:
  - external state
  - user input
  - timers
  - random values
  - hardware errors

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).

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.

Part 2: Discrete Mathematics

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)

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.

Theorems, Lemmata, Corollaries

Definition (theorem, lemma, corollary)

An important true proposition is called a *theorem*. A *lemma* is a preliminary proposition useful for proving other propositions (usually theorems) and a *corollary* is a 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.

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 $n = p + q$.

Mathematical Notation

Notation	Explanation
$P \land Q$	logical and of propositions P and Q
$P \lor Q$	logical or of propositions P and Q
$\lnot P$	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

Greek Letters

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

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

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. $\square$

Prove an Implication by its Contrapositive

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

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

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. $\square$

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. $\square$

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 $c_1, \dots, c_N$.

  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 $c_1, \dots c_N$, the statement P holds. $\square$

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. $\square$

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.

Section 6: Sets

Sets

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

Basic Relations between Sets

Definition (basic relations between sets)

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

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

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

Operations on Sets 1/2

Definition (set union)

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

Definition (set intersection)

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

Definition (set difference)

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

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

Definition (cartesian product)

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

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.

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 $X_1 \times \ldots \times X_k$

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.

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.

Properties of Binary Relations (Endorelations)

Definition

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

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

Equivalence, 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.

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	eq	po	spo	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	✓	✓	✓	$a \sim b \wedge b \sim c \Rightarrow a \sim c$	✓	✓	✓

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

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 $(x, y) \in f$.

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.)

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.

Lambda Notation of Functions

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

Currying

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

Part 3: Number Systems, Units, Characters, Date and Time

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.

Number Systems in Mathematics

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

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

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

Section 9: Natural Numbers

Numeral Systems for Natural Numbers

- Natural numbers can be represented 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

Natural Numbers Literals

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

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

- 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!

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.

Section 10: Integer Numbers

Integer Numbers

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

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

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

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

- We have a fixed number space with n digits and base b to represent integer numbers, that is, we can distinguish at most b^n different integers.
- Lets represent 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.

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.

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!

Section 11: Rational and Real Numbers

Rational Numbers

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

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

Real Numbers

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

Section 12: Floating Point Numbers

Floating Point Numbers

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

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

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

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.

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.

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.

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.

Section 13: International System of Units

Importance of Units and Unit Prefixes

- Most numbers we encounter in practice have associated units. It is important to be very 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.

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.

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	$\mathrm{kg\,m\,s}^{-1}$	force
watt	W	$\mathrm{kg\,m}^2\,\mathrm{s}^{-3}$	power
volt	V	$\mathrm{kg\,m}^2\,\mathrm{s}^{-3}\,\mathrm{A}^{-1}$	voltage
ohm	Ω	$\mathrm{kg\,m}^2\,\mathrm{s}^{-2}\,\mathrm{A}^{-1}$	resistance
velocity		$\mathrm{m\,s}^{-1}$	speed

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

Section 14: Characters and Strings

Characters and Character Encoding

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

ASCII Characters and Encoding

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

ASCII Characters and Code Points (decimal)

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

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 12.1 (published May 2019) defines 137 929 characters, each identified by an unambiguous name and an integer number called its code point.
- The overall code point space is divided into 17 planes where each plane has $2^{16} = 65536$ code points. The Basic Multilingual Plane (plane 0) contains characters of almost all modern languages, and a large number of symbols.
- Unicode can be implemented using different character encodings. The UTF-32 encoding encodes character code points directly into 32-bit numbers (fixed length encoding). While simple, an ASCII text of size n would become a UTF-32 text of size $4n$.

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 $. ☺

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.

Section 15: Date and Time

System Time and Clocks

- Computer systems usually maintain a notion of *system time*. The term system time indicates that two different systems usually have a different notion of system time.
- System time is measured by a *system clock*, which is typically implemented as a simple count of the number of ticks (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.)

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.

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

Part 4: Boolean Algebra and Logic

Logic can be confusing. . .

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

Section 16: Elementary Boolean Functions

Boolean Variables

- Boolean logic describes objects that can take only one of two values.
- The values may be different voltage levels $\{0, V^+\}$ or special symbols $\{F, T\}$ or simply the digits $\{0, 1\}$.
- In the following, we use the notation $\mathbb{B} = \{0, 1\}$.
- In artificial intelligence, such objects are often called *propositions* and they are either *true* or *false*.
- In mathematics, the objects are called *Boolean variables* and we use the symbols $X_1, X_2, X_3, \ldots$ for them.
- The main purpose of Boolean logic is to describe (or design) interdependencies between Boolean variables.

Interpretation of Boolean Variables

Definition (Boolean variables)

A Boolean variable X_i with $i \geq 1$ is an object that can take on one of the two values 0 or 1. The set of all Boolean variables is $\mathrm{X} = \{X_1, X_2, X_3, \ldots\}$.

Definition (Interpretation)

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

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

Boolean $\wedge$ 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 $\lor$ Function (or)

X	Y	$X \lor Y$
0	0	0
0	1	1
1	0	1
1	1	1

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

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

- 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 $\lor$ operation.

Boolean ¬ Function (not)

X	$\neg X$
0	1
1	0

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

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

- The ¬ operation simply flips a Boolean value.

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

Boolean $\rightarrow$ Function (implies)

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

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

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

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

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

Boolean $\leftrightarrow$ Function (equivalence)

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

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

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

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

Boolean $\dot\vee$ Function (exclusive or)

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

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

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

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

Boolean $\uparrow$ Function (not-and)

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

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

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

- The $\uparrow$ function is also called Sheffer stroke.

- While we use the functions $\wedge$, $\vee$, 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.

Boolean $\downarrow$ Function (not-or)

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

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

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

- The $\downarrow$ function is also called Quine arrow.

- The $\downarrow$ (nor) is like $\uparrow$ (nand) sufficient to derive all elementary Boolean functions.

Section 17: Boolean Functions and Formulas

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.

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.

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 $(\varphi \land \psi)$ is a Boolean formula.

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

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

Semantics of Boolean formulas

Definition (Semantics of Boolean formulas)

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

Basis of the inductive definition:

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

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

Semantics of Boolean formulas

Definition (Semantics of Boolean formulas (cont.))

Induction step, with φ and ψ in Φ(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

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

Section 18: Boolean Algebra Equivalence Laws

Tautology and contradiction

Definition (adapted interpretation)

An interpretation $\mathcal{I} : \mathrm{D} \to \mathbb{B}$ is *adapted* to a Boolean formula φ if all Boolean variables that occur in φ are contained in 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 $\mathcal{I}(\varphi) = 0$.

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.

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.

Boolean Equivalence laws

Proposition (equivalence laws)

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

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

Section 19: 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 $(1 \lor 1) \land (X_1 \lor 0)$
  - $X_1 \land X_2$ (this is a short form of $(X_1 \lor X_1) \land (X_2 \lor X_2)$)
  - $X_1 \lor X_2$ (this is a short form of $(1 \lor 1) \land (X_1 \lor X_2)$)
  - $\neg X_1 \land (X_2 \lor X_3)$ (this is a short form of $(0 \lor \neg X_1) \land (X_2 \lor X_3)$)
  - $(X_1 \lor \neg X_2) \land (\neg X_1 \lor X_2)$
- We typically write the short form, leaving out trivial expansions into full CNF form.

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 $(0 \land 0) \lor (X_1 \land 1)$)
  - $X_1 \land X_2$ (this is a short form of $(0 \land 0) \lor (X_1 \land X_2)$)
  - $X_1 \lor X_2$ (this is a short form of $(X_1 \land X_1) \lor (X_2 \land X_2)$)
  - $(\neg X_1 \land X_2) \lor (\neg X_1 \land X_3)$
  - $(\neg X_1 \land \neg X_2) \lor (X_1 \land X_2)$
- We typically write the short form, leaving out trivial expansions into full DNF form.

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.

Obtaining a DNF from a Truth Table

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

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

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

Obtaining a CNF from a Truth Table

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

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

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

Section 20: Complexity of Boolean Formulas

Cost of Boolean Expressions and Functions

Definition (cost of boolean expression)

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

Definition (cost of boolean function)

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

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

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.

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.

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.

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.

Section 21: Boolean Logic and the Satisfiability Problem

Logic Statements

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

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

and we are interested to find out whether such a formula is true, i.e., whether it is a tautology.

Tautology and Satisfiability

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

  $$\tau = (\varphi_1 \land \ldots \land \varphi_m) \to \psi$$

  is a tautology, we may check whether

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

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

Tautology and Satisfiability

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

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

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

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

Satisfiability Problem

Definition (satisfiability problem)

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

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

Part 5: Computer Architecture

Section 22: Combinational Digital Circuits

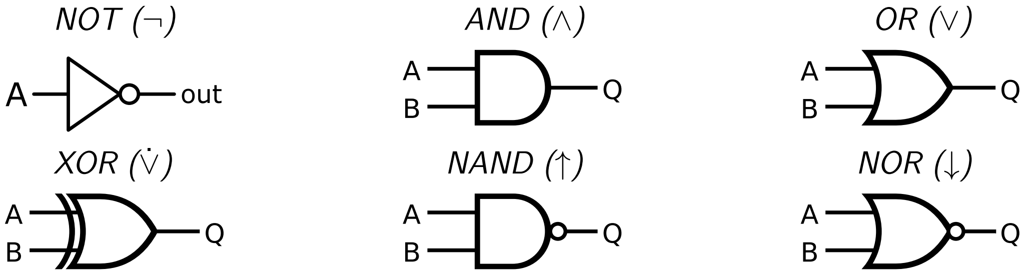
Recall elementary boolean operations and functions

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

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

- 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.

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.

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.

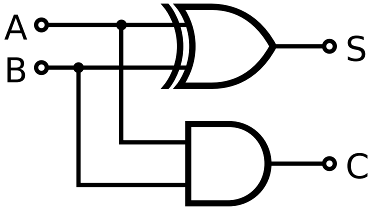
Adding two bits (half adder)

- 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

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

$$C = A \wedge B$$



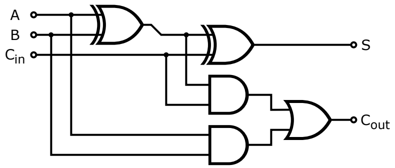
Adding two bits (full adder)

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

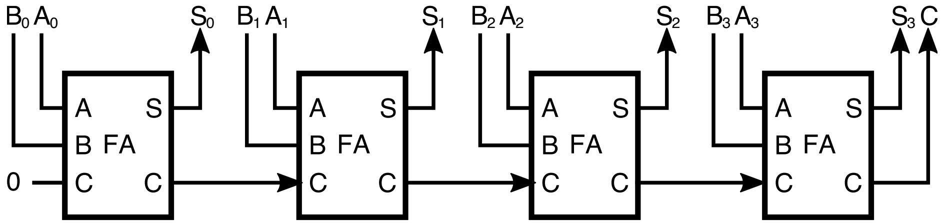
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

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

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



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.

Adding N Bits (carry-lookahead adder)

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

Multiplication of Decimal and Binary Numbers

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

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

Section 23: 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:

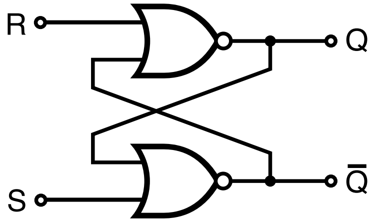
1. It should be able to hold a value.
2. It should be possible to read the value that was saved.
3. It should be possible to change the value that was saved.

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

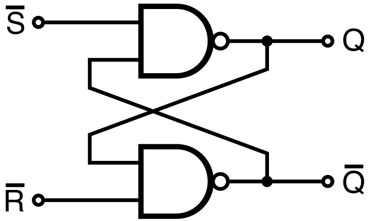
S	R	Q	$\bar{Q}$	Q	$\bar{Q}$
0	0	0	1	0	1
0	0	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1



- 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 $R = 0 \wedge S = 0$.

SR Latch using NAND Gates

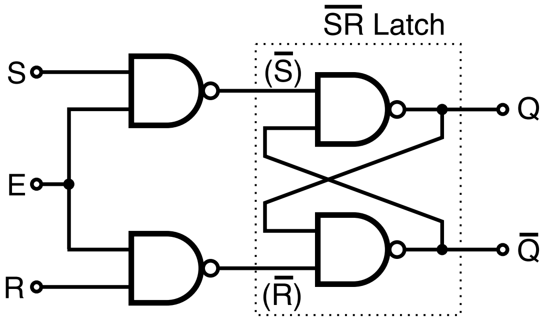
$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	Q	$\bar{Q}$
1	1	0	1	0	1
1	1	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1



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

Gated SR Latch using NAND Gates

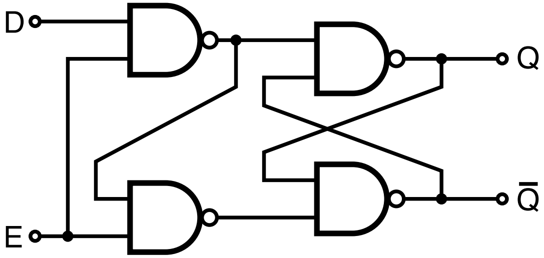
E	S	R	Q	$\bar{Q}$	Q	$\bar{Q}$
0	x	x	0	1	0	1
0	x	x	1	0	1	0
1	1	1	0	1	0	1
1	1	1	1	0	1	0
1	0	1	0	1	1	0
1	0	1	1	0	1	0
1	1	0	0	1	0	1
1	1	0	1	0	0	1



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

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”

Level-triggered versus Edge-triggered

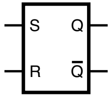
Definition (level-triggered)

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).

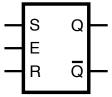
Definition (edge-triggered)

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).

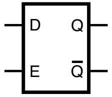
Latch Types and Symbols



SR Latch

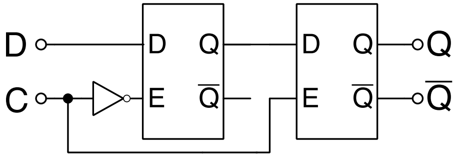


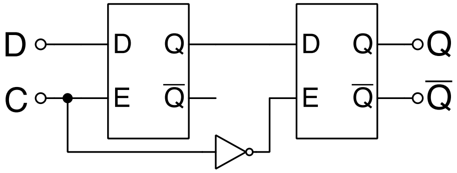
Gated SR Latch



Gated D Latch

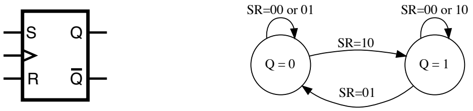
D Flip-Flop (positive and negative edge-triggered)



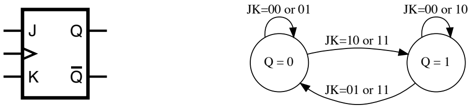


- Edge-triggered D flip-flops propagate changes from the primary to the secondary latch on either the rising edge of a clock or the falling edge of the clock.
- Positive edge-triggered: state changes when the clock becomes high
- Negative edge-triggered: state changes when the clock becomes low

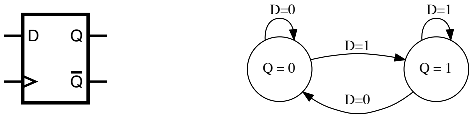
Flip-Flop Types, State Diagrams, and Symbols



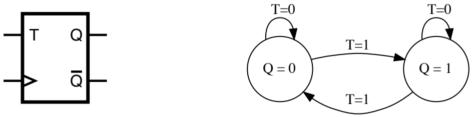
SR Flip-Flop



JK Flip-Flop



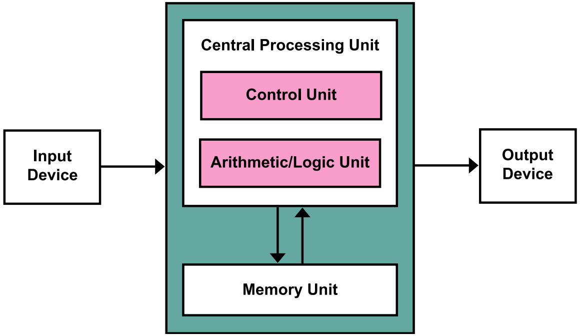
D Flip-Flop



T Flip-Flop

Section 24: Von Neumann Computer Architecture

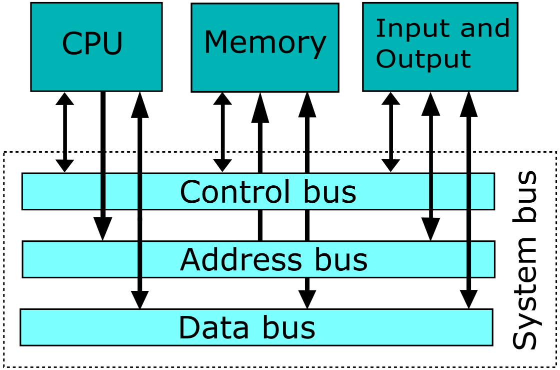
Von Neumann computer architecture



- 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

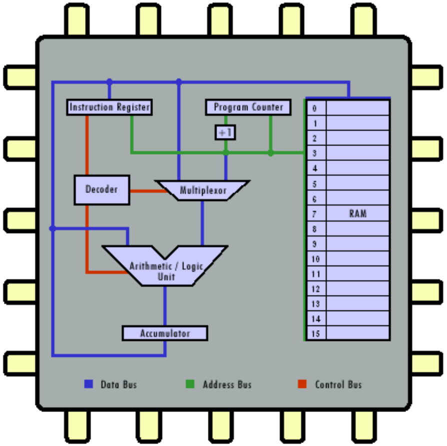
Computer system bus (data, address, and control)



- 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)

Simple Central Processing Unit



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

Instruction cycle (fetch – decode – execute cycle)

```
while True:
    advance_program_counter();
    instruction = fetch();
    decode(instruction);
    execute(instruction);
```

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

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

Simple Machine Language

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

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

Program #1 in our simple machine language

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

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

Program #2 in our simple machine language

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

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

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 for 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 255 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.
- Real 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* is holding a stack frame for every active function call. A *stack frame* provides memory space to hold

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),
- an instruction to call a function (pushing a new stack frame on the call stack),
- an instruction to return from a function (popping a stack frame off the stack).

Part 6: System Software

Section 25: 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.

Simple C program to add two numbers

```
/* C source code

   (C is a compiled procedural programming language) */

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

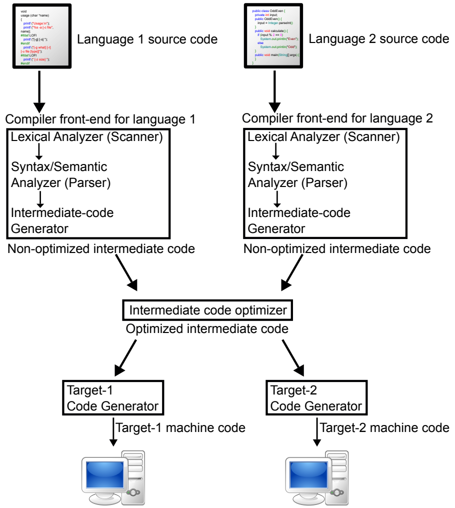
Disassembled machine code (without optimizations)

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

00000000004004ac <main>:
  4004ac:       55                      push   %rbp
  4004ad:       48 89 e5                mov    %rsp,%rbp
  4004b0:       c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
  4004b7:       c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
  4004be:       8b 45 f8                mov    -0x8(%rbp),%eax
  4004c1:       8b 55 fc                mov    -0x4(%rbp),%edx
  4004c4:       01 d0                   add    %edx,%eax
  4004c6:       89 45 f4                mov    %eax,-0xc(%rbp)
  4004c9:       8b 45 f4                mov    -0xc(%rbp),%eax
  4004cc:       5d                      pop    %rbp
  4004cd:       c3                      retq
  4004ce:       90                      nop
  4004cf:       90                      nop
```

Disassembled machine code (with optimizations)

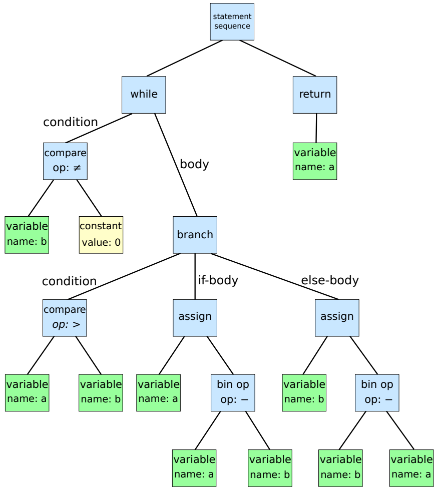
```
# compile with optimization (gcc -O2) and look at the machine code
# gcc (Debian 4.7.2-5) 4.7.2 on Linux

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

Compiler


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

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

Abstract Syntax Tree Example



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

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

Backus-Naur-Form and Formal Languages

The syntax of programming languages is often defined using syntax rules. A common notation for syntax rules is the Backus-Naur-Form (BNF):

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

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

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

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).

Virtual Machines and Emulators

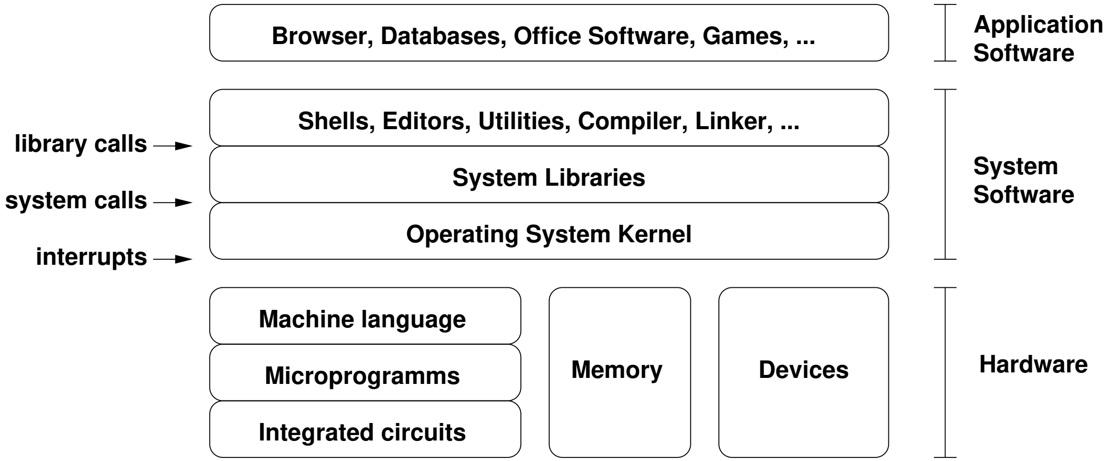
- A virtual machine (VM) is an emulation of a particular computer system. Virtual machines operate based on the computer architecture and functions of a real computer.
- An emulator is hardware or software or both that duplicates (or emulates) the functions of one computer system (the guest) in another computer system (the host), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest).

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

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

Section 26: Operating Systems

Hardware vs. System Software vs. Application Software



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)

OS Abstraction #1: Processes and Process Lifecycle

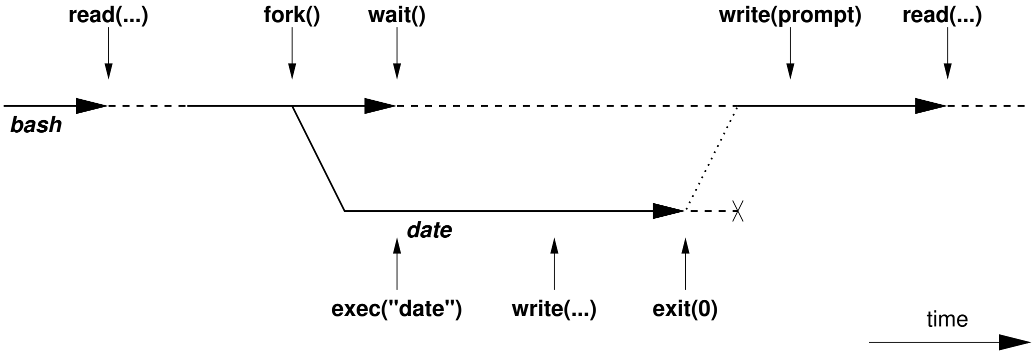
Definition (process)

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

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

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

OS Abstraction #1: Processes and Process Lifecycle



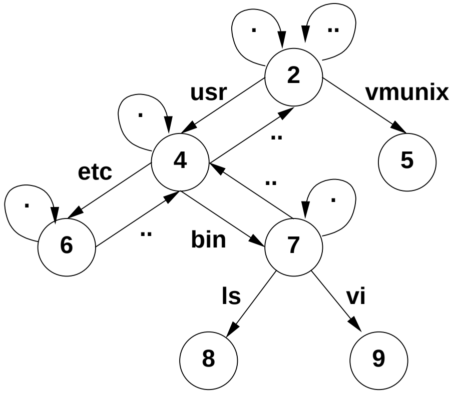
OS Abstraction #1: Processes and Process Lifecycle

```
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 */
    }
}
```

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

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

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

OS Abstraction #3: Inter-process Communication

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

Part 7: Software Correctness

Section 27: 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.

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:

$$\{X = 1\}\ X := X + 1\ \{X = 2\}$$

Partial Correctness and Total Correctness

Definition (partial correctness)

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

Definition (total correctness)

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

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 in the following slides.

- We will use lowercase letters such as x and y to denote auxiliary variables (e.g., to denote values stored in variables).

Hoare Assignments

- Syntax: $V := E$
- Semantics: The state is changed by assigning the value of the term E to the variable V. All variables are assumed to have global scope.
- Example: $X := X + 1$

Hoare Skip Command

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

Hoare Command Sequences

- Syntax: $C_1; \ldots; C_n$
- Semantics: The commands $C_1, \ldots, C_n$ are executed in that order.
- Example: $R := X; X := Y; Y := R$

Hoare Conditionals

- Syntax: $\mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}$
- 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: $\mathit{IF}\ X < Y\ \mathit{THEN}\ M := Y\ \mathit{ELSE}\ M := X\ \mathit{FI}$

Hoare While Loop

- Syntax: $\mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}$
- 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: $\mathit{WHILE}\ \neg(X = 0)\ \mathit{DO}\ X := X - 2\ \mathit{OD}$

Termination can be Tricky

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

- Collatz conjecture: The program will eventually return the number 1, regardless of which positive integer is chosen initially.

Specification can be Tricky

- Specification for the maximum of two variables:

$$\{\mathrm{T}\}\ C\ \{Y = max(X, Y)\}$$

- C could be:

```
IF X > Y THEN Y := X ELSE SKIP FI
```

- But C could also be:

```
IF X > Y THEN X := Y ELSE SKIP FI
```

- And C could also be:

```
Y := X
```

- Use auxiliary variables x and y to associate Q with P:

$$\{X = x \land Y = y\}\ C\ \{Y = max(x, y)\}$$

Section 28: 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.

Precondition Strengthening

- If P implies P' and we have shown $\{P'\}\ C\ \{Q\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

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

- Example: Since $\vdash X = n \rightarrow X + 1 = n + 1$, we can strengthen

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

to

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

Postcondition Weakening

- If Q' implies Q and we have shown $\{P\}\ C\ \{Q'\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

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

- Example: Since $X = n + 1 \rightarrow X > n$, we can weaken

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

to

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

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.

Strongest Postcondition

Definition (stronges postcondition)

Given a program C and a precondition P, the *strongest postcondition* $sp(C, P)$ has the property that $\vdash \{P\}\ C\ \{sp(C, P)\}$ and for any Q with $\vdash \{P\}\ C\ \{Q\}$, we have $\vdash sp(C, P) \rightarrow Q$.

- The "strongest" postcondition Q means that any other valid postcondition is implied by Q (via postcondition weakening).

Assignment Axiom

- Let $P[E/V]$ (P with E for V) denote the result of substituting the term E for all 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:

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

Specification Conjunction and Disjunction

- If we have shown $\{P_1\}\ C\ \{Q_1\}$ and $\{P_2\}\ C\ \{Q_2\}$, then $\{P_1 \land P_2\}\ C\ \{Q_1 \land Q_2\}$ holds as well:

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \land P_2\}\ C\ \{Q_1 \land Q_2\}}$$

- We get a similar rule for disjunctions:

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \lor P_2\}\ C\ \{Q_1 \lor Q_2\}}$$

- These rules allows us to prove $\vdash \{P\}\ C\ \{Q_1 \land Q_2\}$ by proving both $\vdash \{P\}\ C\ \{Q_1\}$ and $\vdash \{P\}\ C\ \{Q_2\}$.

Skip Command Rule

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

$$\frac{}{\vdash \{P\}\ \mathit{SKIP}\ \{P\}}$$

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:

$$\frac{\vdash \{P\}\ C_1\ \{R_1\}, \vdash \{R_1\}\ C_2\ \{R_2\},\ \ldots,\ \vdash \{R_{n-1}\}\ C_n\ \{Q\}}{\vdash \{P\}\ C_1; C_2; \ldots; C_n\ \{Q\}}$$

Conditional Command Rule

- Syntax: *IF S THEN C_1 ELSE C_2 FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Conditional Rule:

$$\frac{\vdash \{P \land S\}\ C_1\ \{Q\}, \quad \vdash \{P \land \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ \{Q\}}$$

While Command Rule

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- While Rule:

$$\frac{\vdash \{P \land S\}\ C\ \{P\}}{\vdash \{P\}\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ \{P \land \neg S\}}$$

P is an invariant of C whenever S holds. Since executing C preserves the truth of P, executing C any numbner of times also preserves the truth of P.

Section 29: 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.

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 $\vdash \{P\}\ C\ \{Q\}$.

Generation of Verification Conditions

- Assignment $\{P\}\ V := E\ \{Q\}$:
  Add verification condition $P \rightarrow Q[E/V]$.
- Conditions $\{P\}\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ \{Q\}$
  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 $\{P\}\ \mathit{WHILE}\ S\ \mathit{DO}\ \{R\}\ C\ \mathit{OD}\ \{Q\}$
  Add verification conditions $P \rightarrow R$ and $R \wedge \neg S \rightarrow Q$
  Add verificiation conditions generated by $\{R \wedge S\}\ C\ \{R\}$

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.

Rules for Total Correctness [1/4]

- Assignment axiom

$$\vdash [P[E/V]]\ V := E\ [P]$$

- Precondition strengthening

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

- Postcondition weakening

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

Rules for Total Correctness [2/4]

- Specification conjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \land P_2]\ C\ [Q_1 \land Q_2]}$$

- Specification disjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \lor P_2]\ C\ [Q_1 \lor Q_2]}$$

- Skip command rule

$$\frac{}{[P]\ SKIP\ [P]}$$

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

$$\frac{\vdash [P \land S \land E = n]\; C\; [P \land (E < n)], \quad \vdash P \land S \to E \geq 0}{\vdash [P]\; \mathit{WHILE}\; S\; \mathit{DO}\; C\; \mathit{OD}\; [P \land \neg S]}$$

  E is an integer-valued expression
  n is an auxiliary variable not occuring in P, C, S, or E

- A prove has to show that a non-negative integer, called a *variant*, decreases on each iteration of the loop command C.

Generation of Termination Verification Conditions

- The rules for the generation of termination verificiation conditions follow directly from the rules for the generation of partial correctness verificiation conditions, except for the while command.
- To handle the while command, we need an additional annotation (in square brackets) that provides the variant expression.
- For while loops of the form $\{P\}$ *WHILE S DO* $\{R\}$ $[E]$ *C OD* $\{Q\}$ add the verification conditions

$$
\begin{gathered}
P \rightarrow R \\
R \land \neg S \rightarrow Q \\
R \land S \rightarrow E \geq 0
\end{gathered}
$$

and add verificiation conditions generated by $\{R \land S \land (E = n)\}\ C\ \{R \land (E < n)\}$

Termination and Correctness

- Partial correctness and termination implies total correctness:

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

- Total correctness implies partial correctness and termination:

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