General Computer Science
320201 GenCS I & II Lecture Notes

Michael Kohlhase

School of Engineering & Science
Jacobs University, Bremen Germany
`m.kohlhase@jacobs-university.de`

August 26, 2013

Preface

This Document

This document contains the course notes for the course General Computer Science I & II held at Jacobs University Bremen[1] in the academic years 2003-2013.

Contents: The document mixes the slides presented in class with comments of the instructor to give students a more complete background reference.

Caveat: This document is made available for the students of this course only. It is still a draft and will develop over the course of the current course and in coming academic years.

Licensing: This document is licensed under a Creative Commons license that requires attribution, allows commercial use, and allows derivative works as long as these are licensed under the same license.

Knowledge Representation Experiment: This document is also an experiment in knowledge representation. Under the hood, it uses the sTeX package [Koh08, Koh12], a TeX/LaTeX extension for semantic markup, which allows to export the contents into the eLearning platform PantaRhei.

Comments and extensions are always welcome, please send them to the author.

Other Resources: The course notes are complemented by a selection of problems (with and without solutions) that can be used for self-study. [Koh11a, Koh11b]

Course Concept

Aims: The course 320101/2 “General Computer Science I/II” (GenCS) is a two-semester course that is taught as a mandatory component of the “Computer Science” and “Electrical Engineering & Computer Science” majors (EECS) at Jacobs University. The course aims to give these students a solid (and somewhat theoretically oriented) foundation of the basic concepts and practices of computer science without becoming inaccessible to ambitious students of other majors.

Context: As part of the EECS curriculum GenCS is complemented with a programming lab that teaches the basics of `C` and `C`$^{++}$ from a practical perspective and a “Computer Architecture” course in the first semester. As the programming lab is taught in three five-week blocks over the first semester, we cannot make use of it in GenCS.

In the second year, GenCS, will be followed by a standard “Algorithms & Data structures” course and a “Formal Languages & Logics” course, which it must prepare.

Prerequisites: The student body of Jacobs University is extremely diverse — in 2011, we have students from 110 nations on campus. In particular, GenCS students come from both sides of the “digital divide”: Previous CS exposure ranges “almost computer-illiterate” to “professional `Java` programmer” on the practical level, and from “only calculus” to solid foundations in discrete Mathematics for the theoretical foundations. An important commonality of Jacobs students however is that they are bright, resourceful, and very motivated.

As a consequence, the GenCS course does not make any assumptions about prior knowledge, and introduces all the necessary material, developing it from first principles. To compensate for this, the course progresses very rapidly and leaves much of the actual learning experience to homework problems and student-run tutorials.

Course Contents

Goal: To give students a solid foundation of the basic concepts and practices of Computer Science we try to raise awareness for the three basic concepts of CS: “data/information”, “algorithms/programs” and “machines/computational devices” by studying various instances, exposing more and more characteristics as we go along.

[1] International University Bremen until Fall 2006

Computer Science: In accordance to the goal of teaching students to “think first” and to bring out the Science of CS, the general style of the exposition is rather theoretical; practical aspects are largely relegated to the homework exercises and tutorials. In particular, almost all relevant statements are proven mathematically to expose the underlying structures.

GenCS is not a programming course: even though it covers all three major programming paradigms (imperative, functional, and declarative programming)[1]. The course uses `SML` as its primary programming language as it offers a clean conceptualization of the fundamental concepts of recursion, and types. An added benefit is that `SML` is new to virtually all incoming Jacobs students and helps equalize opportunities. EdN:1

GenCS I (the first semester): is somewhat oriented towards computation and representation. In the first half of the semester the course introduces the dual concepts of induction and recursion, first on unary natural numbers, and then on arbitrary abstract data types, and legitimizes them by the Peano Axioms. The introduction and of the functional core of `SML` contrasts and explains this rather abstract development. To highlight the role of representation, we turn to Boolean expressions, propositional logic, and logical calculi in the second half of the semester. This gives the students a first glimpse at the syntax/semantics distinction at the heart of CS.

GenCS II (the second semester): is more oriented towards exposing students to the realization of computational devices. The main part of the semester is taken up by a “building an abstract computer”, starting from combinational circuits, via a register machine which can be programmed in a simple assembler language, to a stack-based machine with a compiler for a bare-bones functional programming language. In contrast to the “computer architecture” course in the first semester, the GenCS exposition abstracts away from all physical and timing issues and considers circuits as labeled graphs. This reinforces the students’ grasp of the fundamental concepts and highlights complexity issues. The course then progresses to a brief introduction of Turing machines and discusses the fundamental limits of computation at a rather superficial level, which completes an introductory “tour de force” through the landscape of Computer Science. As a contrast to these foundational issues, we then turn practical introduce the architecture of the Internet and the World-Wide Web.

The remaining time, is spent on studying one class algorithms (search algorithms) in more detail and introducing the notition of declarative programming that uses search and logical representation as a model of computation.

Acknowledgments

Materials: Some of the material in this course is based on course notes prepared by Andreas Birk, who held the course 320101/2 “General Computer Science” at IUB in the years 2001-03. Parts of his course and the current course materials were based on the book “Hardware Design” (in German) [KP95]. The section on search algorithms is based on materials obtained from Bernhard Beckert (Uni Koblenz), which in turn are based on Stuart Russell and Peter Norvig’s lecture slides that go with their book “Artificial Intelligence: A Modern Approach” [RN95].

The presentation of the programming language Standard ML, which serves as the primary programming tool of this course is in part based on the course notes of Gert Smolka’s excellent course “Programming” at Saarland University [Smo08].

Contributors: The preparation of the course notes has been greatly helped by Ioan Sucan, who has done much of the initial editing needed for semantic preloading in STEX. Herbert Jaeger, Christoph Lange, and Normen Müller have given advice on the contents.

GenCS Students: The following students have submitted corrections and suggestions to this and earlier versions of the notes: Saksham Raj Gautam, Anton Kirilov, Philipp Meerkamp, Paul Ngana, Darko Pesikan, Stojanco Stamkov, Nikolaus Rath, Evans Bekoe, Marek Laska, Moritz Beber, Andrei Aiordachioaie, Magdalena Golden, Andrei Eugeniu Ioniţă, Semir Elezović, Dimitar Asenov, Alen Stojanov, Felix Schlesinger, Ştefan Anca, Dante Stroe, Irina Calciu, Nemanja

[1] EdNote: termrefs!

Ivanovski, Abdulaziz Kivaza, Anca Dragan, Razvan Turtoi, Catalin Duta, Andrei Dragan, Dimitar Misev, Vladislav Perelman, Milen Paskov, Kestutis Cesnavicius, Mohammad Faisal, Janis Beckert, Karolis Uziela, Josip Djolonga, Flavia Grosan, Aleksandar Siljanovski, Iurie Tap, Barbara Khalibinzwa, Darko Velinov, Anton Lyubomirov Antonov, Christopher Purnell, Maxim Rauwald, Jan Brennstein, Irhad Elezovikj, Naomi Pentrel, Jana Kohlhase, Victoria Beleuta, Dominik Kundel, Daniel Hasegan, Mengyuan Zhang, Georgi Gyurchev, Timo Lücke, Sudhashree Sayenju, Lukas Kohlhase, Dmitrii Cucleschin, Aleksandar Gyorev, Tyler Buchmann, Bidesh Thapaliya, Dan Daniel Erdmann-Pham, Petre Munteanu, Utkrist Adhikari, Kim Philipp Jablonski, Aleksandar Gyorev, Tom Wiesing, Sourabh Lal, Nikhil Shakya, Otar Bichiashvili, Cornel Amariei.

Recorded Syllabus for 2013/14

In this section, we record the progress of the course in the academic year 2013/2014 in the form of a "recorded syllabus", i.e. a syllabus that is created after the fact rather than before.

Recorded Syllabus Fall Semester 2013:

#	date	until	slide	page

Here the syllabus of of the last academic year for reference, the current year should be similar. The slide numbers refer to the course notes of that year, available at `http://kwarc.info/teaching/GenCS2/notes2012-13.pdf`.

Syllabus Fall Semester 2012:

#	date	until	slide	page
1	Sep 3.	through with admin	11	8
2	Sep 4.	at end of motivation	29	19
3	Sep 10.	introduced proofs with Peano axioms	34	26
4	Sep 11.	covered mathtalk	42	31
5	Sep 17.	Sets and Russel's paradox	50	36
6	Sep 18.	functions	59	41
7	Sep 24.	SML pattern matching	??	??
8	Sep 25.	higher-order functions	??	??
9	Oct 1.	datatypes	??	??
10	Oct 2.	abstract procedures & computation	??	??
11	Oct 8.	substitutions and terms	??	??
12	Oct 9.	mutual recursion	??	??
13	Oct 15.	I/O and exceptions	??	??
14	Oct 29.	Sufficient Conditions for Prefix Codes	??	??
15	Oct 30.	UTF Codes	??	??
16	Nov 5.	Boolean Expressions	??	??
17	Nov 6.	Costs of Boolean Expressions	??	??
18	Nov 12.	Elementary Complexity Theory	??	??
19	Nov 13.	Naive Quine McCluskey	??	??
20	Nov 19.	Intro to Propositional Logic	??	??
21	Nov 20.	The miracle of logics	??	??
22	Nov 26.	Hilbert Calculus is sound	??	??
23	Nov 27.	Natural Deduction for Mathtalk	??	??
24	Dec 3.	Tableaux & Resolution	??	??
25	Dec 4.	Intellectual Property, Copyright, & Information Privacy	??	??

Spring Semester 2013:

#	date	until	slide	page
1	Feb 4.	Graph Isomorphism	??	??
2	Feb 5.	Parse Tree	??	??
3	Feb 11.	Universality of NAND and NOR	??	??
4	Feb 12.	Full adder	??	??
5	Feb 18.	Two's complement numbers	??	??
6	Feb 19.	RAM layout	??	??
7	Feb 25.	Basic Assembler	??	??
8	Feb 26.	Virtual machine basics	??	??
9	Mar 4.	Implementing VM arithmetics	??	??
10	Mar 5.	Compiling SW into VM	??	??
11	Mar 11.	Realizing Call Frames	??	??
12	Mar 12.	Compiling μML	??	??
13	Mar 18.	Universal Turing Machine	??	??
14	Mar 19.	Halting Problem	??	??
15	Apr 2.	Internet Protocol Suite	??	??
16	Apr 9.	Application Layer	??	??
17	Apr 15.	WWW Overview	??	??
18	Apr 16.	HTML	??	??
19	Apr 22.	Server Side Scripting	??	??
20	Apr 23.	Web Search: Queries	??	??
21	Apr 29.	public key encryption	??	??
22	Apr 30.	XPath	??	??
23	May 6.	Breadth-first search	??	??
24	May 7.	A^* search	??	??

Contents

Chapter 1

Getting Started with "General Computer Science"

Jacobs University offers a unique CS curriculum to a special student body. Our CS curriculum is optimized to make the students successful computer scientists in only three years (as opposed to most US programs that have four years for this). In particular, we aim to enable students to pass the GRE subject test in their fifth semester, so that they can use it in their graduate school applications.

The Course 320101/2 "General Computer Science I/II" is a one-year introductory course that provides an overview over many of the areas in Computer Science with a focus on the foundational aspects and concepts. The intended audience for this course are students of Computer Science, and motivated students from the Engineering and Science disciplines that want to understand more about the "why" rather than only the "how" of Computer Science, i.e. the "science part".

1.1 Overview over the Course

Plot of "General Computer Science"

- ▷ Today: Motivation, Admin, and find out what you already know
  - ▷ What is Computer Science?
  - ▷ Information, Data, Computation, Machines
  - ▷ a (very) quick walk through the topics
- ▷ Get a feeling for the math involved (⚠ not a programming course!!! ⚠)
  - ▷ learn mathematical language (so we can talk rigorously)
  - ▷ inductively defined sets, functions on them
  - ▷ elementary complexity analysis
- ▷ Various machine models (as models of computation)
  - ▷ (primitive) recursive functions on inductive sets
  - ▷ combinational circuits and computer architecture
  - ▷ Programming Language: Standard ML (great equalizer/thought provoker)
  - ▷ Turing machines and the limits of computability

▷ Fundamental Algorithms and Data structures

Overview: The purpose of this two-semester course is to give you an introduction to what the Science in “Computer Science” might be. We will touch on a lot of subjects, techniques and arguments that are of importance. Most of them, we will not be able to cover in the depth that you will (eventually) need. That will happen in your second year, where you will see most of them again, with much more thorough treatment.

Computer Science: We are using the term “Computer Science” in this course, because it is the traditional anglo-saxon term for our field. It is a bit of a misnomer, as it emphasizes the computer alone as a computational device, which is only one of the aspects of the field. Other names that are becoming increasingly popular are “Information Science”, “Informatics” or “Computing”, which are broader, since they concentrate on the notion of information (irrespective of the machine basis: hardware/software/wetware/alienware/vaporware) or on computation.

Definition 1 What we mean with Computer Science here is perhaps best represented by the following quote:

> The body of knowledge of computing is frequently described as the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, What can be (efficiently) automated? [Den00]

Not a Programming Course: Note “General CS” is not a programming course, but an attempt to give you an idea about the “Science” of computation. Learning how to write correct, efficient, and maintainable, programs is an important part of any education in Computer Science, but we will not focus on that in this course (we have the Labs for that). As a consequence, we will not concentrate on teaching how to program in “General CS” but introduce the `SML` language and assume that you pick it up as we go along (however, the tutorials will be a great help; so go there!).

Standard ML: We will be using Standard ML (`SML`), as the primary vehicle for programming in the course. The primary reason for this is that as a functional programming language, it focuses more on clean concepts like recursion or typing, than on coverage and libraries. This teaches students to “think first” rather than “hack first”, which meshes better with the goal of this course. There have been long discussions about the pros and cons of the choice in general, but it has worked well at Jacobs University (even if students tend to complain about `SML` in the beginning).

A secondary motivation for `SML` is that with a student body as diverse as the GenCS first-years at Jacobs[1] we need a language that equalizes them. `SML` is quite successful in that, so far none of the incoming students had even heard of the language (apart from tall stories by the older students).

Algorithms, Machines, and Data: The discussion in “General CS” will go in circles around the triangle between the three key ingredients of computation.

Algorithms are abstract representations of computation instructions

Data are representations of the objects the computations act on

Machines are representations of the devices the computations run on

The figure below shows that they all depend on each other; in the course of this course we will look at various instantiations of this general picture.

[1]traditionally ranging from students with no prior programming experience to ones with 10 years of semi-pro `Java`

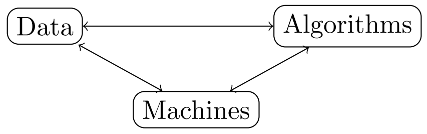


Figure 1.1: The three key ingredients of Computer Science

Representation: One of the primary focal items in "General CS" will be the notion of *representation*. In a nutshell the situation is as follows: we cannot compute with objects of the "real world", but be have to make electronic counterparts that can be manipulated in a computer, which we will call representations. It is essential for a computer scientist to realize that objects and their representations are different, and to be aware of their relation to each other. Otherwise it will be difficult to predict the relevance of the results of computation (manipulating electronic objects in the computer) for the real-world objects. But if cannot do that, computing loses much of its utility.

Of course this may sound a bit esoteric in the beginning, but I will come back to this very often over the course, and in the end you may see the importance as well.

1.2 Administrativa

We will now go through the ground rules for the course. This is a kind of a social contract between the instructor and the students. Both have to keep their side of the deal to make learning and becoming Computer Scientists as efficient and painless as possible.

1.2.1 Grades, Credits, Retaking

Now we come to a topic that is always interesting to the students: the grading scheme. The grading scheme I am using has changed over time, but I am quite happy with it.

Prerequisites, Requirements, Grades

- ▷ Prerequisites: Motivation, Interest, Curiosity, hard work
  - ▷ You can do this course if you want!
- ▷ Grades: (plan your work involvement carefully)

Monday Quizzes	30%
Graded Assignments	20%
Mid-term Exam	20%
Final Exam	30%

  Note that for the grades, the percentages of achieved points are added with the weights above, and only then the resulting percentage is converted to a grade.

- ▷ Monday Quizzes: (Almost) every monday, we will use the first 10 minutes for a brief quiz about the material from the week before (you have to be there)
- ▷ Rationale: I want you to work continuously (maximizes learning)

▷ Requirements for Auditing: You can audit GenCS! (specify in Campus Net)

To earn an audit you have to take the quizzes and do reasonably well(I cannot check that you took part regularly otherwis

My main motivation in this grading scheme is that I want to entice you to learn continuously. You cannot hope to pass the course, if you only learn in the reading week. Let us look at the components of the grade. The first is the exams: We have a mid-term exam relatively early, so that you get feedback about your performance; the need for a final exam is obvious and tradition at Jacobs. Together, the exams make up 50% of your grade, which seems reasonable, so that you cannot completely mess up your grade if you fail one.

In particular, the 50% rule means that if you only come to the exams, you basically have to get perfect scores in order to get an overall passing grade. This is intentional, it is supposed to encourage you to spend time on the other half of the grade. The homework assignments are a central part of the course, you will need to spend considerable time on them. Do not let the 20% part of the grade fool you. If you do not at least attempt to solve all of the assignments, you have practically no chance to pass the course, since you will not get the practice you need to do well in the exams. The value of 20% is attempts to find a good trade-off between discouraging from cheating, and giving enough incentive to do the homework assignments. Finally, the monday quizzes try to ensure that you will show up on time on mondays, and are prepared.

The (relatively severe) rule for auditing is intended to ensure that auditors keep up with the material covered in class. I do not have any other way of ensuring this (at a reasonable cost for me). Many students who think they can audit GenCS find out in the course of the semester that following the course is too much work for them. This is not a problem. An audit that was not awarded does not make any ill effect on your transcript, so feel invited to try.

Advanced Placement

▷ Generally: AP let’s you drop a course, but retain credit for it (sorry no grade!)

- ▷ you register for the course, and take an AP exam
- ▷ ⚠ you will need to have very good results to pass ⚠
- ▷ If you fail, you have to take the course or drop it!

▷ Specifically: AP exams (oral) some time next week (see me for a date)

- ▷ Be prepared to answer elementary questions about: discrete mathematics, terms, substitution, abstract interpretation, computation, recursion, termination, elementary complexity, Standard ML, types, formal languages, Boolean expressions (possible subjects of the exam)

▷ Warning: you should be very sure of yourself to try (genius in C^{++} insufficient)

Although advanced placement is possible, it will be very hard to pass the AP test. Passing an AP does not just mean that you have to have a passing grade, but very good grades in all the topics that we cover. This will be very hard to achieve, even if you have studied a year of Computer Science at another university (different places teach different things in the first year). You can still take the exam, but you should keep in mind that this means considerable work for the instrutor.

1.2.2 Homeworks, Submission, and Cheating

Homework assignments

- ▷ Goal: Reinforce and apply what is taught in class.
- ▷ Homeworks: will be small individual problem/programming/proof assignments(but take time to solve) group submission if and only if explicitly permitted
- ▷ Admin: To keep things running smoothly
  - ▷ Homeworks will be posted on PantaRhei
  - ▷ Homeworks are handed in electronically in `grader` (plain text, Postscript, PDF,. . .)
  - ▷ go to the tutorials, discuss with your TA (they are there for you!)
  - ▷ materials: sometimes posted ahead of time; then read before class, prepare questions, bring printout to class to take notes
- ▷ Homework Discipline:
  - ▷ start early! (many assignments need more than one evening's work)
  - ▷ Don't start by sitting at a blank screen
  - ▷ Humans will be trying to understand the text/code/math when grading it.

Homework assignments are a central part of the course, they allow you to review the concepts covered in class, and practice using them.

Homework Submissions, Grading, Tutorials

- ▷ Submissions: We use Heinrich Stamerjohanns' `grader` system
  - ▷ submit all homework assignments electronically to `https://jgrader.de`
  - ▷ you can login with you Jacobs account (should have one!)
  - ▷ feedback/grades to your submissions
  - ▷ get an overview over how you are doing! (do not leave to midterm)
- ▷ Tutorials: select a tutorial group and actually go to it regularly
  - ▷ to discuss the course topics after class (GenCS needs pre/postparation)
  - ▷ to discuss your homework after submission (to see what was the problem)
  - ▷ to find a study group (probably the most determining factor of success)

The next topic is very important, you should take this very seriously, even if you think that this is just a self-serving regulation made by the faculty.

All societies have their rules, written and unwritten ones, which serve as a social contract among its members, protect their interestes, and optimize the functioning of the society as a whole. This is also true for the community of scientists worldwide. This society is special, since it balances intense cooperation on joint issues with fierce competition. Most of the rules are largely unwritten; you are expected to follow them anyway. The code of academic integrity at Jacobs is an attempt to put some of the aspects into writing.

It is an essential part of your academic education that you learn to behave like academics, i.e. to function as a member of the academic community. Even if you do not want to become a scientist in the end, you should be aware that many of the people you are dealing with have gone through an academic education and expect that you (as a graduate of Jacobs) will behave by these rules.

The Code of Academic Integrity

- ▷ Jacobs has a "Code of Academic Integrity"
  - ▷ this is a document passed by the Jacobs community (our law of the university)
  - ▷ you have signed it last week (we take this seriously)
- ▷ It mandates good behavior and penalizes bad from both faculty and students
  - ▷ honest academic behavior (we don't cheat/falsify)
  - ▷ respect and protect the intellectual property of others (no plagiarism)
  - ▷ treat all Jacobs members equally (no favoritism)
- ▷ this is to protect you and build an atmosphere of mutual respect
  - ▷ academic societies thrive on reputation and respect as primary currency
- ▷ The Reasonable Person Principle (one lubricant of academia)
  - ▷ we treat each other as reasonable persons
  - ▷ the other's requests and needs are reasonable until proven otherwise
  - ▷ but if the other violates our trust, we are deeply disappointed(severe uncompromising consequences)

To understand the rules of academic societies it is central to realize that these communities are driven by economic considerations of their members. However, in academic societies, the primary good that is produced and consumed consists in ideas and knowledge, and the primary currency involved is academic reputation[2]. Even though academic societies may seem as altruistic — scientists share their knowledge freely, even investing time to help their peers understand the concepts more deeply — it is useful to realize that this behavior is just one half of an economic transaction. By publishing their ideas and results, scientists sell their goods for reputation. Of course, this can only work if ideas and facts are attributed to their original creators (who gain reputation by being cited). You will see that scientists can become quite fierce and downright nasty when confronted with behavior that does not respect other's intellectual property.

The Academic Integrity Committee (AIC)

- ▷ Joint Committee by students and faculty (Not at "student honours court")
- ▷ Mandate: to hear and decide on any major or contested allegations, in particular,

[2]Of course, this is a very simplistic attempt to explain academic societies, and there are many other factors at work there. For instance, it is possible to convert reputation into money: if you are a famous scientist, you may get a well-paying job at a good university,...

- ▷ 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

▷ we/you can appeal any academic integrity allegations to the AIC

One special case of academic rules that affects students is the question of cheating, which we will cover next.

Cheating [adapted from CMU:15-211 (P. Lee, 2003)]

▷ There is no need to cheat in this course!! (hard work will do)

▷ cheating prevents you from learning (you are cutting your own flesh)

▷ if you are in trouble, come and talk to me (I am here to help you)

▷ We expect you to know what is useful collaboration and what is 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(even if you change most of the actu
- ▷ more on this as the semester goes on ...

▷ ⚠ There are data mining tools that monitor the originality of text/code. ⚠

▷ Procedure: If we catch you at cheating (correction: if we suspect cheating)

- ▷ we will confront you with the allegation (you can explain yourself)
- ▷ if you admit or are silent, we impose a grade sanction and notify registrar
- ▷ repeat infractions to go the AIC for deliberation (much more serious)

▷ Note: both active (copying from others) and passive cheating (allowing others to copy) are penalized equally

We are fully aware that the border between cheating and useful and legitimate collaboration is difficult to find and will depend on the special case. Therefore it is very difficult to put this into firm rules. We expect you to develop a firm intuition about behavior with integrity over the course of stay at Jacobs.

1.2.3 Resources

Textbooks, Handouts and Information, Forum

▷ No required textbook, but course notes, posted slides

▷ Course notes in PDF will be posted at `http://kwarc.info/teaching/GenCS1.html`

- ▷ Everything will be posted on PantaRhei (Notes+assignments+course forum)
  - ▷ announcements, contact information, course schedule and calendar
  - ▷ discussion among your fellow students(careful, I will occasionally check for academic integrity!)
  - ▷ `http://panta.kwarc.info` (use your jacobs login)
  - ▷ if there are problems send e-mail to `course-gencs-tas@jacobs-university.de`

No Textbook: Due to the special circumstances discussed above, there is no single textbook that covers the course. Instead we have a comprehensive set of course notes (this document). They are provided in two forms: as a large PDF that is posted at the course web page and on the PantaRhei system. The latter is actually the preferred method of interaction with the course materials, since it allows to discuss the material in place, to play with notations, to give feedback, etc. The PDF file is for printing and as a fallback, if the PantaRhei system, which is still under development, develops problems.

Software/Hardware tools

- ▷ You will need computer access for this course(come see me if you do not have a computer of your own)
- ▷ we recommend the use of standard software tools
  - ▷ the `emacs` and `vi` text editor (powerful, flexible, available, free)
  - ▷ `UNIX` (`linux`, `MacOSX`, `cygwin`) (prevalent in CS)
  - ▷ `FireFox` (just a better browser (for Math))
- ▷ learn how to touch-type NOW (reap the benefits earlier, not later)

Touch-typing: You should not underestimate the amount of time you will spend typing during your studies. Even if you consider yourself fluent in two-finger typing, touch-typing will give you a factor two in speed. This ability will save you at least half an hour per day, once you master it. Which can make a crucial difference in your success.

Touch-typing is very easy to learn, if you practice about an hour a day for a week, you will re-gain your two-finger speed and from then on start saving time. There are various free typing tutors on the network. At `http://typingsoft.com/all_typing_tutors.htm` you can find about programs, most for windows, some for linux. I would probably try `Ktouch` or `TuxType`

Darko Pesikan (one of the previous TAs) recommends the `TypingMaster` program. You can download a demo version from `http://www.typingmaster.com/index.asp?go=tutordemo`

You can find more information by googling something like "learn to touch-type". (goto `http://www.google.com` and type these search terms).

Next we come to a special project that is going on in parallel to teaching the course. I am using the coures materials as a research object as well. This gives you an additional resource, but may affect the shape of the coures materials (which now server double purpose). Of course I can use all the help on the research project I can get.

Experiment: E-Learning with OMDoc/PantaRhei

- ▷ My research area: deep representation formats for (mathematical) knowledge
- ▷ Application: E-learning systems (represent knowledge to transport it)
- ▷ Experiment: Start with this course (Drink my own medicine)
  - ▹ Re-Represent the slide materials in OMDoc (Open Math Documents)
  - ▹ Feed it into the PantaRhei system (`http://panta.kwarc.info`)
  - ▹ Try it on you all (to get feedback from you)
- ▷ Tasks (Unfortunately, I cannot pay you for this; maybe later)
  - ▹ help me complete the material on the slides (what is missing/would help?)
  - ▹ I need to remember "what I say", examples on the board. (take notes)
- ▷ Benefits for you (so why should you help?)
  - ▹ you will be mentioned in the acknowledgements (for all that is worth)
  - ▹ you will help build better course materials (think of next-year's freshmen)

JACOBS UNIVERSITY

Chapter 2

Motivation and Introduction

Before we start with the course, we will have a look at what Computer Science is all about. This will guide our intuition in the rest of the course.

Consider the following situation, Jacobs University has decided to build a maze made of high hedges on the the campus green for the students to enjoy. Of course not any maze will do, we want a maze, where every room is reachable (unreachable rooms would waste space) and we want a unique solution to the maze to the maze (this makes it harder to crack).

Acknowledgement: The material in this chapter is adapted from the introduction to a course held by Prof. Peter Lee at Carnegie Mellon university in 2002.

2.1 What is Computer Science?

What is Computer Science about?

- ▷ For instance: Software! (a hardware example would also work)
- ▷ **Example 2** writing a program to generate mazes.
- ▷ We want every maze to be solvable. (should have path from entrance to exit)
- ▷ Also: We want mazes to be fun, i.e.,
  - ▷ We want maze solutions to be unique
  - ▷ We want every "room" to be reachable
- ▷ How should we think about this?

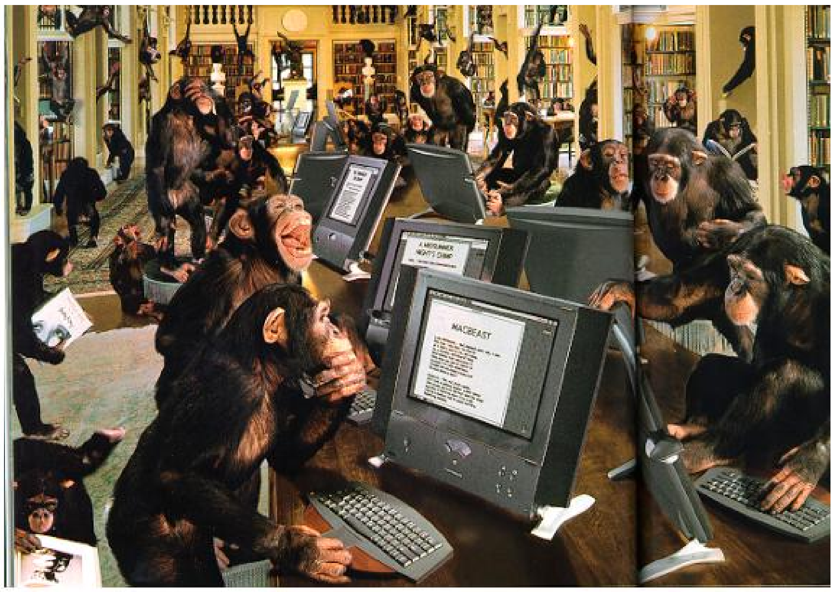
There are of course various ways to build such a a maze; one would be to ask the students from biology to come and plant some hedges, and have them re-plant them until the maze meets our criteria. A better way would be to make a plan first, i.e. to get a large piece of paper, and draw a maze before we plant. A third way is obvious to most students:

An Answer:

Let's hack

However, the result would probably be the following:

2am in the IRC Quiet Study Area



If we just start hacking before we fully understand the problem, chances are very good that we will waste time going down blind alleys, and garden paths, instead of attacking problems. So the main motto of this course is:

no, let's think

- ▷ "*The GIGO Principle: Garbage In, Garbage Out*" (– ca. 1967)
- ▷ "*Applets, Not Craplets*^{tm}" (– ca. 1997)

2.2 Computer Science by Example

Thinking about a problem will involve thinking about the representations we want to use (after all, we want to work on the computer), which computations these representations support, and what constitutes a solutions to the problem.

This will also give us a foundation to talk about the problem with our peers and clients. Enabling students to talk about CS problems like a computer scientist is another important learning goal of this course.

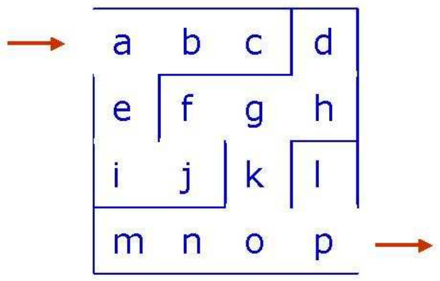
We will now exemplify the process of "thinking about the problem" on our mazes example. It shows that there is quite a lot of work involved, before we write our first line of code. Of course, sometimes, explorative programming sometimes also helps understand the problem , but we would consider this as part of the thinking process.

Thinking about the problem

▷ Idea: Randomly knock out walls until we get a good maze

▷ Think about a grid of rooms separated by walls.

▷ Each room can be given a name.



▷ Mathematical Formulation:

- ▷ a set of rooms: $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
- ▷ Pairs of adjacent rooms that have an open wall between them.

▷ **Example 3** For example, $\langle a, b\rangle$ and $\langle g, k\rangle$ are pairs.

▷ Abstractly speaking, this is a mathematical structure called a graph.

Of course, the "thinking" process always starts with an idea of how to attack the problem. In our case, this is the idea of starting with a grid-like structure and knocking out walls, until we have a maze which meets our requirements.

Note that we have already used our first representation of the problem in the drawing above: we have drawn a picture of a maze, which is of course not the maze itself.

Definition 4 A representation is the realization of real or abstract persons, objects, circumstances, Events, or emotions in concrete symbols or models. This can be by diverse methods, e.g. visual, aural, or written; as three-dimensional model, or even by dance.

Representations will play a large role in the course, we should always be aware, whether we are talking about "the real thing" or a representation of it (chances are that we are doing the latter in computer science). Even though it is important, to be able to always able to distinguish representations from the objects they represent, we will often be sloppy in our language, and rely on the ability of the reader to distinguish the levels.

From the pictorial representation of a maze, the next step is to come up with a mathematical representation; here as sets of rooms (actually room names as representations of rooms in the maze) and room pairs.

Why math?

▷ Q: Why is it useful to formulate the problem so that mazes are room sets/pairs?

▷ A: Data structures are typically defined as mathematical structures.

▷ A: Mathematics can be used to reason about the correctness and efficiency of data structures and algorithms.

▷ A: Mathematical structures make it easier to think — to abstract away from unnecessary details and avoid "hacking".

The advantage of a mathematical representation is that it models the aspects of reality we are interested in in isolation. Mathematical models/representations are very abstract, i.e. they have very few properties: in the first representational step we took we abstracted from the fact that we want to build a maze made of hedges on the campus green. We disregard properties like maze size, which kind of bushes to take, and the fact that we need to water the hedges after we planted them. In the abstraction step from the drawing to the set/pairs representation, we abstracted from further (accidental) properties, e.g. that we have represented a square maze, or that the walls are blue.

As mathematical models have very few properties (this is deliberate, so that we can understand all of them), we can use them as models for many concrete, real-world situations.

Intuitively, there are few objects that have few properties, so we can study them in detail. In our case, the structures we are talking about are well-known mathematical objects, called graphs.

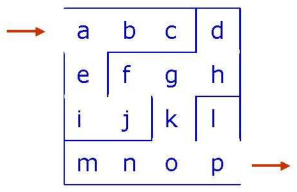
We will study graphs in more detail in this course, and cover them at an informal, intuitive level here to make our points.

Mazes as Graphs

▷ **Definition 5** Informally, a graph consists of a set of nodes and a set of edges.(a good part of CS is about graph algorith

▷ **Definition 6** A maze is a graph with two special nodes.

▷ Interpretation: Each graph node represents a room, and an edge from node x to node y indicates that rooms x and y are adjacent and there is no wall in between them. The first special node is the entry, and the second one the exit of the maze.



Can be represented as

$$\left\langle \left\{ \begin{array}{l} \langle a,e\rangle, \langle e,i\rangle, \langle i,j\rangle, \\ \langle f,j\rangle, \langle f,g\rangle, \langle g,h\rangle, \\ \langle d,h\rangle, \langle g,k\rangle, \langle a,b\rangle \\ \langle m,n\rangle, \langle n,o\rangle, \langle b,c\rangle \\ \langle k,o\rangle, \langle o,p\rangle, \langle l,p\rangle \end{array} \right\}, a, p \right\rangle$$

So now, we have identified the mathematical object, we will use to think about our algorithm, and indeed it is very abstract, which makes it relatively difficult for humans to work with. To get around this difficulty, we will often draw pictures of graphs, and use them instead. But we will always keep in mind that these are not the graphs themselves but pictures of them — arbitrarily adding properties like color and layout that are irrelevant to the actual problem at hand but help the human cognitive apparatus in dealing with the problems. If we do keep this in mind, we can have both, the mathematical rigor and the intuitive ease of argumentation.

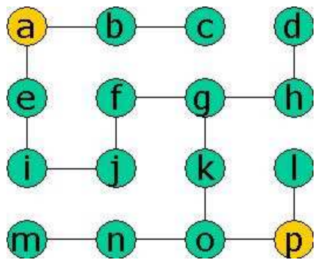
Mazes as Graphs (Visualizing Graphs via Diagrams)

▷ Graphs are very abstract objects, we need a good, intuitive way of thinking about them. We use diagrams, where the nodes are visualized as dots and the edges as lines between them.

Our maze

▷ $\left\langle \left\{ \begin{array}{l} \langle a,e\rangle, \langle e,i\rangle, \langle i,j\rangle, \\ \langle f,j\rangle, \langle f,g\rangle, \langle g,h\rangle, \\ \langle d,h\rangle, \langle g,k\rangle, \langle a,b\rangle \\ \langle m,n\rangle, \langle n,o\rangle, \langle b,c\rangle \\ \langle k,o\rangle, \langle o,p\rangle, \langle l,p\rangle \end{array} \right\}, a, p \right\rangle$

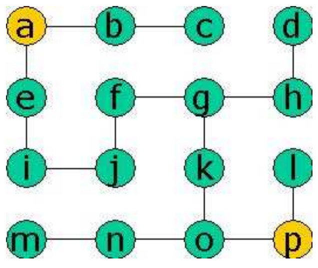
can be visualized as



▷ Note that the diagram is a visualization (a representation intended for humans to process visually) of the graph, and not the graph itself.

Now that we have a mathematical model for mazes, we can look at the subclass of graphs that correspond to the mazes that we are after: unique solutions and all rooms are reachable! We will concentrate on the first requirement now and leave the second one for later.

Unique solutions



▷ Q: What property must the graph have for the maze to have a solution?

▷ A: A path from a to p.

▷ Q: What property must it have for the maze to have a unique solution?

▷ A: The graph must be a tree.

Trees are special graphs, which we will now define.

Mazes as trees

▷ **Definition 7** Informally, a tree is a graph:

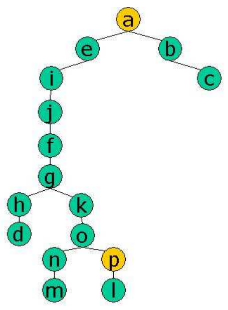
- ▹ with a unique root node, and
- ▹ each node having a unique parent.

▷ **Definition 8** A spanning tree is a tree that includes all of the nodes.

Q: Why is it good to have a spanning tree?

⧐ A: Trees have no cycles! (needed for uniqueness)

▷ A: Every room is reachable from the root!



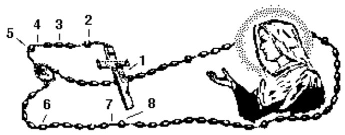
So, we know what we are looking for, we can think about a program that would find spanning trees given a set of nodes in a graph. But since we are still in the process of "thinking about the problems" we do not want to commit to a concrete program, but think about programs in the abstract (this gives us license to abstract away from many concrete details of the program and concentrate on the essentials).

The computer science notion for a program in the abstract is that of an algorithm, which we will now define.

Algorithm

▷ Now that we have a data structure in mind, we can think about the algorithm.

▷ **Definition 9** An algorithm is a series of instructions to control a (computation) process



▷ **Example 10 (Kruskal's algorithm, a graph algorithm for spanning trees)** ▷ Randomly add a pair to the tree if it won't create a cycle. (i.e. tear down a wall)

▷ Repeat until a spanning tree has been created.

Definition 11 An algorithm is a collection of formalized rules that can be understood and executed, and that lead to a particular endpoint or result.

Example 12 An example for an algorithm is a recipe for a cake, another one is a rosary — a kind of chain of beads used by many cultures to remember the sequence of prayers. Both the recipe and rosary represent instructions that specify what has to be done step by step. The instructions in a recipe are usually given in natural language text and are based on elementary forms of manipulations like “scramble an egg” or “heat the oven to 250 degrees Celsius”. In a rosary, the instructions are represented by beads of different forms, which represent different prayers. The physical (circular) form of the chain allows to represent a possibly infinite sequence of prayers.

The name algorithm is derived from the word al-Khwarizmi, the last name of a famous Persian mathematician. Abu Ja'far Mohammed ibn Musa al-Khwarizmi was born around 780 and died around 845. One of his most influential books is “Kitab al-jabr w'al-muqabala” or “Rules of Restoration and Reduction”. It introduced algebra, with the very word being derived from a part of the original title, namely “al-jabr”. His works were translated into Latin in the 12th century, introducing this new science also in the West.

The algorithm in our example sounds rather simple and easy to understand, but the high-level formulation hides the problems, so let us look at the instructions in more detail. The crucial one is the task to check, whether we would be creating cycles.

Of course, we could just add the edge and then check whether the graph is still a tree, but this would be very expensive, since the tree could be very large. A better way is to maintain some information during the execution of the algorithm that we can exploit to predict cyclicity before altering the graph.

Creating a spanning tree

▷ When adding a wall to the tree, how do we detect that it won't create a cycle?

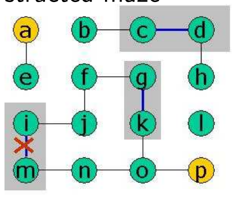
▷ When adding wall $\langle x, y\rangle$, we want to know if there is already a path from x to y in the tree.

▷ In fact, there is a fast algorithm for doing exactly this, called "Union-Find".

Definition 13 (Union Find Algorithm) The Union Find Algorithm successively puts nodes into an equivalence class if there is a path connecting them.

▷ Before adding an edge $\langle x, y\rangle$ to the tree, it makes sure that x and y are not in the same equivalence class.

Example 14 A partially constructed maze



Now that we have made some design decision for solving our maze problem. It is an important part of "thinking about the problem" to determine whether these are good choices. We have argued above, that we should use the Union-Find algorithm rather than a simple "generate-and-test" approach based on the "expense", by which we interpret temporally for the moment. So we ask ourselves

How fast is our Algorithm?

▷ Is this a fast way to generate mazes?

- ▷ How much time will it take to generate a maze?
- ▷ What do we mean by "fast" anyway?

▷ In addition to finding the right algorithms, Computer Science is about analyzing the performance of algorithms.

In order to get a feeling what we mean by "fast algorithm", we to some preliminary computations.

Performance and Scaling

▷ Suppose we have three algorithms to choose from. (which one to select)

▷ Systematic analysis reveals performance characteristics.

▷ For a problem of size n (i.e., detecting cycles out of n nodes) we have

n	$100n$ μs	$7n^2$ μs	2^n μs
1	100 μs	7 μs	2 μs
5	.5 ms	175 μs	32 μs
10	1 ms	.7 ms	1 ms
45	4.5 ms	14 ms	1.1 years
100	...	...	...
1 000	...	...	...
10 000	...	...	...
1 000 000	...	...	...

What?! One year?

- ▷ $2^{10} = 1\,024$ $(1024\ \mu s)$
- ▷ $2^{45} = 35\,184\,372\,088\,832$ $(\cdot 3.5 10^{13}\ \mu s = \cdot 3.5 10^{7}\ s \equiv 1.1$ years)
- ▷ we denote all times that are longer than the age of the universe with –

n	$100n\ \mu s$	$7n^2\ \mu s$	$2^n\ \mu s$
1	$100\ \mu s$	$7\ \mu s$	$2\ \mu s$
5	.5 ms	$175\ \mu s$	$32\ \mu s$
10	1 ms	.7 ms	1 ms
45	4.5 ms	14 ms	1.1 years
100	100 ms	$7\ s$	10^{16} years
1 000	$1\ s$	12 min	–
10 000	$10\ s$	$20\ h$	–
1 000 000	1.6 min	2.5 mo	–

So it does make a difference for larger problems what algorithm we choose. Considerations like the one we have shown above are very important when judging an algorithm. These evaluations go by the name of complexity theory.

2.3 Other Topics in Computer Science

We will now briefly preview other concerns that are important to computer science. These are essential when developing larger software packages. We will not be able to cover them in this course, but leave them to the second year courses, in particular “software engineering”.

The first concern in software engineering is of course whether your program does what it is supposed to do.

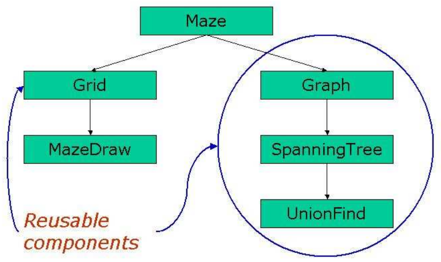
Is it correct?

- ▷ How will we know if we implemented our solution correctly?
  - ▷ What do we mean by “correct”?
  - ▷ Will it generate the right answers?
  - ▷ Will it terminate?
- ▷ Computer Science is about techniques for proving the correctness of programs

Modular design

- ▷ By thinking about the problem, we have strong hints about the structure of our program

- ▷ Grids, Graphs (with edges and nodes), Spanning trees, Union-find.
- ▷ With disciplined programming, we can write our program to reflect this structure.
- ▷ Modular designs are usually easier to get right and easier to understand.



Indeed, modularity is a major concern in the design of software: if we can divide the functionality of the program in to small, self-contained "modules" that provide a well-specified functionality (possibly building on other modules), then we can divide work, and develop and test parts of the program separately, greatly reducing the overall complexity of the development effort.

In particular, if we can identify modules that can be used in multiple situations, these can be published as libraries, and re-used in multiple programs, again reducing complexity.

Modern programming languages support modular design by a variety of measures and structures. The functional programming language SML presented in this course, has a very elaborate module system, which we will not be able to cover in this course. Hover, SML data types allow to define what object-oriented languages use "classes" for: sets of similarly-structured objects that support the same sort of behavior (which can be the pivotal points of modules).

2.4 Summary

The science in CS: not "hacking", but

- ▷ Thinking about problems abstractly.
- ▷ Selecting good structures and obtaining correct and fast algorithms/machines.
- ▷ Implementing programs/machines that are understandable and correct.

In particular, the course "General Computer Science" is not a programming course, it is about being able to think about computational problems and to learn to talk to others about these problems.

Part I

Representation and Computation

Chapter 3

Elementary Discrete Math

We have seen in the last section that we will use mathematical models for objects and data structures throughout Computer Science. As a consequence, we will need to learn some math before we can proceed. But we will study mathematics for another reason: it gives us the opportunity to study rigorous reasoning about abstract objects, which is needed to understand the "science" part of Computer Science.

Note that the mathematics we will be studying in this course is probably different from the mathematics you already know; calculus and linear algebra are relatively useless for modeling computations. We will learn a branch of math. called "discrete mathematics", it forms the foundation of computer science, and we will introduce it with an eye towards computation.

Let's start with the math!

Discrete Math for the moment

▷ Kenneth H. Rosen *Discrete Mathematics and Its Applications*, McGraw-Hill, 1990 [Ros90].

▷ Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, 1998 [LP98].

▷ Paul R. Halmos, *Naive Set Theory*, Springer Verlag, 1974 [Hal74].

The roots of computer science are old, much older than one might expect. The very concept of computation is deeply linked with what makes mankind special. We are the only animal that manipulates abstract concepts and has come up with universal ways to form complex theories and to apply them to our environments. As humans are social animals, we do not only form these theories in our own minds, but we also found ways to communicate them to our fellow humans.

3.1 Mathematical Foundations: Natural Numbers

The most fundamental abstract theory that mankind shares is the use of numbers. This theory of numbers is detached from the real world in the sense that we can apply the use of numbers to arbitrary objects, even unknown ones. Suppose you are stranded on an lonely island where you see a strange kind of fruit for the first time. Nevertheless, you can immediately count these fruits. Also, nothing prevents you from doing arithmetics with some fantasy objects in your mind. The question in the following sections will be: what are the principles that allow us to form and apply numbers in these general ways? To answer this question, we will try to find general ways to specify and manipulate arbitrary objects. Roughly speaking, this is what computation is all about.

Something very basic:

- ▷ Numbers are symbolic representations of numeric quantities.
- ▷ There are many ways to represent numbers (more on this later)
- ▷ let's take the simplest one (about 8,000 to 10,000 years old)

- ▷ we count by making marks on some surface.
- ▷ For instance //// stands for the number four (be it in 4 apples, or 4 worms)
- ▷ Let us look at the way we construct numbers a little more algorithmically,
- ▷ these representations are those that can be created by the following two rules.

  o-**rule** consider ' ' as an empty space.

  s-**rule** given a row of marks or an empty space, make another / mark at the right end of the row.

- ▷ **Example 15** For ////, Apply the *o*-rule once and then the *s*-rule four times.
- ▷ **Definition 16** we call these representations unary naturalnumbers.

In addition to manipulating normal objects directly linked to their daily survival, humans also invented the manipulation of place-holders or symbols. A *symbol* represents an object or a set of objects in an abstract way. The earliest examples for symbols are the cave paintings showing iconic silhouettes of animals like the famous ones of Cro-Magnon. The invention of symbols is not only an artistic, pleasurable "waste of time" for mankind, but it had tremendous consequences. There is archaeological evidence that in ancient times, namely at least some 8000 to 10000 years ago, men started to use tally bones for counting. This means that the symbol "bone" was used to represent numbers. The important aspect is that this bone is a symbol that is completely detached from its original down to earth meaning, most likely of being a tool or a waste product from a meal. Instead it stands for a universal concept that can be applied to arbitrary objects.

Instead of using bones, the slash / is a more convenient symbol, but it is manipulated in the same way as in the most ancient times of mankind. The *o*-rule allows us to start with a blank slate or an empty container like a bowl. The *s*- or successor-rule allows to put an additional bone into a bowl with bones, respectively, to append a slash to a sequence of slashes. For instance ////

stands for the number four — be it 4 apples, or 4 worms. This representation is constructed by applying the o-rule once and then the s-rule four times.

So, we have a basic understanding of natural numbers now, but we will also need to be able to talk about them in a mathematically precise way. Generally, this additional precision will involve defining specialized vocabulary for the concepts and objects we want to talk about, making the assumptions we have about the objects exmplicit, and the use of special modes or argumentation.

We will introduce all of these for the special case of unary natural numbers here, but we will use the concepts and practices throughout the course and assume students will do so as well.

With the notion of a successor from Definition 17 we can formulate a set of assumptions (called axioms) about unary natural numbers. We will want to use these assumptions (statements we believe to be true) to derive other statements, which — as they have been obtained by generally accepted argumentation patterns — we will also believe true. This intuition is put more formally in Definition 28 below, which also supplies us with names for different types of statements.

A little more sophistication (math) please

- **Definition 17** We call a unary natural number the successor (predecessor) of another, if it can be constructing by adding (removing) a slash. (successors are created by the s-rule)
- **Example 18** $///$ is the successor of $//$ and $//$ the predecessor of $///$.
- **Definition 19** The following set of axioms are called the Peano axioms(Giuseppe Peano $*(1858)$, $\dagger(1932)$)
- **Axiom 20 (P1)** “ ” (aka. “zero”) is a unary natural number.
- **Axiom 21 (P2)** Every unary natural number has a successor that is a unary natural number and that is different from it.
- **Axiom 22 (P3)** Zero is not a successor of any unary natural number.
- **Axiom 23 (P4)** Different unary natural numbers have different successors.
- **Axiom 24 (P5: Induction Axiom)** Every unary natural number possesses a property P, if
  - zero has property P and (base condition)
  - the successor of every unary natural number that has property P also possesses property P (step condition)

Question: Why is this a better way of saying things (why so complicated?)

Note that the Peano axioms may not be the first things that come to mind when thinking about characteristic properties of natural numbers. Indeed they have been selected to to be minimal, so that we can get by with as few assumptions as possible; all other statements of properties can be derived from them, so minimality is not a bug, but a feature: the Peano axioms form the foundation, on which all knowledge about unary natural numbers rests.

We now come to the ways we can derive new knowledge from the Peano axioms.

3.2 Reasoning about Natural Numbers

▷ Reasoning about Natural Numbers

- ▷ The Peano axioms can be used to reason about natural numbers.
- ▷ **Definition 25** An axiom is a statement about mathematical objects that we assume to be true.
- ▷ **Definition 26** A theorem is a statement about mathematical objects that we know to be true.
- ▷ We reason about mathematical objects by inferring theorems from axioms or other theorems, e.g.

  1. " " is a unary natural number (axiom P1)
  2. / is a unary natural number (axiom P2 and 1.)
  3. // is a unary natural number (axiom P2 and 2.)
  4. /// is a unary natural number (axiom P2 and 3.)

- ▷ **Definition 27** We call a sequence of inferences a derivation or a proof (of the last statement).

If we want to be more precise about these (important) notions, we can define them as follows:

Definition 28 In general, a axiom or postulate is a starting point in logical reasoning with the aim to prove a mathematical statement or conjecture. A conjecture that is proven is called a theorem. In addition, there are two subtypes of theorems. The lemma is an intermediate theorem that serves as part of a proof of a larger theorem. The corollary is a theorem that follows directly from another theorem. A logical system consists of axioms and rules that allow inference, i.e. that allow to form new formal statements out of already proven ones. So, a proof of a conjecture starts from the axioms that are transformed via the rules of inference until the conjecture is derived.

We will now practice this reasoning on a couple of examples. Note that we also use them to introduce the inference system (see Definition 28) of mathematics via these example proofs.

Here are some theorems you may want to prove for practice. The proofs are relatively simple.

Let's practice derivations and proofs

- ▷ **Example 29** //////////// is a unary natural number
- ▷ **Theorem 30** */// is a different unary natural number than //.*
- ▷ **Theorem 31** *///// is a different unary natural number than //.*
- ▷ **Theorem 32** *There is a unary natural number of which /// is the successor*
- ▷ **Theorem 33** *There are at least 7 unary natural numbers.*
- ▷ **Theorem 34** *Every unary natural number is either zero or the successor of a unary natural number.* (we will come back to this later)

Induction for unary natural numbers

▷ **Theorem 35** *Every unary natural number is either zero or the successor of a unary natural number.*

▷ Proof: We make use of the induction axiom P5:

P.1 We use the property P of "being zero or a successor" and prove the statement by convincing ourselves of the prerequisites of

P.2 ' ' is zero, so ' ' is "zero or a successor".

P.3 Let n be a arbitrary unary natural number that "is zero or a successor"

P.4 Then its successor "is a successor", so the successor of n is "zero or a successor"

P.5 Since we have taken n arbitrary (nothing in our argument depends on the choice) we have shown that for any n, its successor has property P.

P.6 Property P holds for all unary natural numbers by P5, so we have proven the assertion □

We have already seen in the proof above, that it helps to give names to objects: for instance, by using the name n for the number about which we assumed the property P, we could just say that $P(n)$ holds. But there is more we can do.

We can give systematic names to the unary natural numbers. Note that it is often to reference objects in a way that makes their construction overt. the unary natural numbers we can represent as expressions that trace the applications of the o-rule and the s-rules.

This seems awfully clumsy, lets introduce some notation

▷ Idea: we allow ourselves to give names to unary natural numbers(we use $n, m, l, k, n_1, n_2, \ldots$ as names for concrete u

▷ Remember the two rules we had for dealing with unary natural numbers

▷ Idea: represent a number by the trace of the rules we applied to construct it.(e.g. //// is represented as $s(s(s(s(o))))$)

▷ **Definition 36** We introduce some abbreviations

- ▷ we "abbreviate" o and ' ' by the symbol '0' (called "zero")
- ▷ we abbreviate $s(o)$ and / by the symbol '1' (called "one")
- ▷ we abbreviate $s(s(o))$ and // by the symbol '2' (called "two")
- ▷ ...
- ▷ we abbreviate $s(s(s(s(s(s(s(s(s(s(s(s(o))))))))))))$ and //////////// by the symbol '12' (called "twelve")
- ▷ ...

▷ **Definition 37** We denote the set of all unary natural numbers with $\mathbb{N}_1$.(either representation)

This systematic representation of natural numbers by expressions becomes very powerful, if we mix it with the practice of giving names to generic objects. These names are often called variables, when used in expressions.

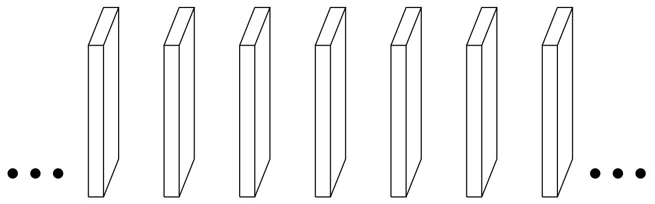
Theorem 35 is a very useful fact to know, it tells us something about the form of unary natural numbers, which lets us streamline induction proofs and bring them more into the form you may know from school: to show that some property P holds for every natural number, we analyze an arbitrary number n by its form in two cases, either it is zero (the base case), or it is a successor of another number (the step case). In the first case we prove the base condition and in the latter, we prove the step condition and use the induction axiom to conclude that all natural numbers have property P. We will show the form of this proof in the domino-induction below.

The Domino Theorem

▷ **Theorem 38** *Let $S_0, S_1, \ldots$ be a linear sequence of dominos, such that for any unary natural number i we know that*

1. *the distance between S_i and $S_{s(i)}$ is smaller than the height of S_i,*
2. *S_i is much higher than wide, so it is unstable, and*
3. *S_i and $S_{s(i)}$ have the same weight.*

If S_0 is pushed towards S_1 so that it falls, then all dominos will fall.



The Domino Induction

▷ Proof: We prove the assertion by induction over i with the property P that "S_i falls in the direction of $S_{s(i)}$".

P.1 We have to consider two cases

P.1.1 base case: i is zero:

P.1.1.1 We have assumed that "S_0 is pushed towards S_1, so that it falls" □

P.1.2 step case: $i = s(j)$ for some unary natural number j:

P.1.2.1 We assume that P holds for S_j, i.e. S_j falls in the direction of $S_{s(j)} = S_i$.

P.1.2.2 But we know that S_j has the same weight as S_i, which is unstable,

P.1.2.3 so S_i falls into the direction opposite to S_j, i.e. towards $S_{s(i)}$ (we have a linear sequence of dominos) □

P.2 We have considered all the cases, so we have proven that P holds for all unary natural numbers i. (by induction)

P.3 Now, the assertion follows trivially, since if "S_i falls in the direction of $S_{s(i)}$", then in particular "S_i falls". □

If we look closely at the proof above, we see another recurring pattern. To get the proof to go through, we had to use a property P that is a little stronger than what we need for the assertion alone. In effect, the additional clause "... in the direction ..." in property P is used to make the step condition go through: we we can use the stronger inductive hypothesis in the proof of step case, which is simpler.

Often the key idea in an induction proof is to find a suitable strengthening of the assertion to get the step case to go through.

3.3 Defining Operations on Natural Numbers

The next thing we want to do is to define operations on unary natural numbers, i.e. ways to do something with numbers. Without really committing what "operations" are, we build on the intuition that they take (unary natural) numbers as input and return numbers. The important thing in this is not what operations are but how we define them.

What can we do with unary natural numbers?

- ▷ So far not much (let's introduce some operations)
- ▷ **Definition 39 (the addition "function")** We "define" the addition operation $\oplus$ procedurally (by an algorithm)
  - ▷ adding zero to a number does not change it.
    written as an equation: $n \oplus o = n$
  - ▷ adding m to the successor of n yields the successor of $m \oplus n$.
    written as an equation: $m \oplus s(n) = s(m \oplus n)$

  Questions: to understand this definition, we have to know
- ▷
  - ▷ Is this "definition" well-formed? (does it characterize a mathematical object?)
  - ▷ May we define "functions" by algorithms? (what is a function anyways?)

So we have defined the addition operation on unary natural numbers by way of two equations. Incidentally these equations can be used for computing sums of numbers by replacing equals by equals; another of the generally accepted manipulation

Definition 40 (Replacement) If we have a representation s of an object and we have an equation $l = r$, then we can obtain an object by replacing an occurrence of the sub-expression l in s by r and have $s = s'$.

In other words if we replace a sub-expression of s with an equal one, nothing changes. This is exactly what we will use the two defining equations from Definition 39 for in the following example

Example 41 (Computing the Sum Two and One) If we start with the expression $s(s(o)) \oplus s(o)$, then we can use the second equation to obtain $s(s(s(o)) \oplus o)$ (replacing equals by equals), and – this time with the first equation $s(s(s(o)))$.

Observe: in the computation in Example 41 at every step there was exactly one of the two equations we could apply. This is a consequence of the fact that in the second argument of the two equations are of the form o and $s(n)$: by Theorem 35 these two cases cover all possible natural numbers and by **P3** (see Axiom 22), the equations are mutually exclusive. As a consequence we do not really have a choice in the computation, so the two equations do form an "algorithm" (to the extend we already understand them), and the operation is indeed well-defined. The form of the arguments in the two equations in Definition 39 is the same as in the induction axiom, therefore we will consider the first equation as the base equation and second one as the step equation.

We can understand the process of computation as a "getting-rid" of operations in the expression. Note that even though the step equation does not really reduce the number of occurrences of the operator (the base equation does), but it reduces the number of constructor in the second argument, essentially preparing the elimination of the operator via the base equation. Note that in any case when we have eliminated the operator, we are left with an expression that is completely made up of constructors; a representation of a unary natural number.

Now we want to see whether we can find out some properties of the addition operation. The method for this is of course stating a conjecture and then proving it.

Addition on unary natural numbers is associative

▷ **Theorem 42** *For all unary natural numbers n, m, and l, we have $n \oplus (m \oplus l) = (n \oplus m) \oplus l$.*

▷ Proof: we prove this by induction on l

P.1 The property of l is that $n \oplus (m \oplus l) = (n \oplus m) \oplus l$ holds.

P.2 We have to consider two cases base case:

P.2.1.1 $n \oplus (m \oplus o) = n \oplus m = (n \oplus m) \oplus o$ □

P.2.2 step case:

P.2.2.1 given arbitrary l, assume $n \oplus (m \oplus l) = (n \oplus m) \oplus l$, show $n \oplus (m \oplus s(l)) = (n \oplus m) \oplus s(l)$.

P.2.2.2 We have $n \oplus (m \oplus s(l)) = n \oplus s(m \oplus l) = s(n \oplus (m \oplus l))$

P.2.2.3 By inductive hypothesis $s((n \oplus m) \oplus l) = (n \oplus m) \oplus s(l)$ □

□

We observe that In the proof above, the induction corresponds to the defining equations of $\oplus$; in particular base equation of $\oplus$ was used in the base case of the induction whereas the step equation of $\oplus$ was used in the step case. Indeed computation (with operations over the unary natural numbers) and induction (over unary natural numbers) are just two sides of the same coin as we will see.

Let us consider a couple more operations on the unary natural numbers to fortify our intutions.

More Operations on Unary Natural Numbers

▷ **Definition 43** The unary multiplication operation can be defined by the equations $n \odot o = o$ and $n \odot s(m) = n \oplus n \odot m$.

▷ **Definition 44** The unary exponentiation operation can be defined by the equations $\exp(n, o) = s(o)$ and $\exp(n, s(m)) = n \odot \exp(n, m)$.

▷ **Definition 45** The unary summation operation can be defined by the equations $\bigoplus_{i=o}^{o} n_i = o$ and $\bigoplus_{i=o}^{s(m)} n_i = n_{s(m)} \oplus \bigoplus_{i=o}^{m} n_i$.

▷ **Definition 46** The unary product operation can be defined by the equations $\bigodot_{i=o}^{o} n_i = s(o)$ and $\bigodot_{i=o}^{s(m)} n_i = n_{s(m)} \odot \bigodot_{i=o}^{m} n_i$.

In Definition 43, we have used the operation ⊕ in the right-hand side of the step-equation. This is perfectly reasonable and only means that we have eliminate more than one operator.

Note that we did not use disambiguating parentheses on the right hand side of the step equation for ⊙. Here $n \oplus n \odot m$ is a unary sum whose second summand is a product. Just as we did there, we will use the usual arithmetic precedences to reduce the notational overload.

The remaining examples are similar in spirit, but a bit more involved, since they nest more operators. Just like we showed associativity for ⊕ in slide 40, we could show properties for these operations, e.g.

$$\bigodot_{i=o}^{n \oplus m} k_i = \bigodot_{i=o}^{n} k_i \oplus \bigodot_{i=o}^{m} k_{(i \oplus n)} \tag{3.1}$$

by induction, with exactly the same observations about the parallelism between computation and induction proofs as ⊕.

Definition 46 gives us the chance to elaborate on the process of definitions some more: When we define new operations such as the product over a sequence of unary natural numbers, we do have freedom of what to do with the corner cases, and for the "empty product" (the base case equation) we could have chosen

1. to leave the product undefined (not nice; we need a base case), or
2. to give it another value, e.g. $s(s(o))$ or o.

But any value but $s(o)$ would violate the generalized distributivity law in equation 3.1 which is exactly what we would expect to see (and which is very useful in calculations). So if we want to have this equation (and I claim that we do) then we have to choose the value $s(o)$.

In summary, even though we have the freedom to define what we want, if we want to define sensible and useful operators our freedom is limited.

3.4 Talking (and writing) about Mathematics

Before we go on, we need to learn how to talk and write about mathematics in a succinct way. This will ease our task of understanding a lot.

Talking about Mathematics (MathTalk)

▷ **Definition 47** Mathematicians use a stylized language that

- ▹ uses formulae to represent mathematical objects, e.g. $\int_0^1 x^{3/2} dx$
- ▹ uses math idioms for special situations (e.g. *iff, hence, let... be..., then...*)
- ▹ classifies statements by role (e.g. Definition, Lemma, Theorem, Proof, Example)

We call this language mathematical vernacular.

▷ **Definition 48** Abbreviations for Mathematical statements in MathTalk

- ▹ ∧ and "∨" are common notations for "and" and "or"
- ▹ "not" is in mathematical statements often denoted with ¬

- ▷ $\forall x.P$ ($\forall x \in S.P$) stands for "condition P holds for all x (in S)"
- ▷ $\exists x.P$ ($\exists x \in S.P$) stands for "there exists an x (in S) such that proposition P holds"
- ▷ $\nexists x.P$ ($\nexists x \in S.P$) stands for "there exists no x (in S) such that proposition P holds"
- ▷ $\exists^1 x.P$ ($\exists^1 x \in S.P$) stands for "there exists one and only one x (in S) such that proposition P holds"
- ▷ "iff" as abbreviation for "if and only if", symbolized by "$\Leftrightarrow$"
- ▷ the symbol "$\Rightarrow$" is used a as shortcut for "implies"

Observation: With these abbreviations we can use formulae for statements.

⊳ **Example 49** $\forall x.\exists y.x = y \Leftrightarrow \neg(x \neq y)$ reads

"For all x, there is a y, such that $x = y$, iff (if and only if) it is not the case that $x \neq y$."

To fortify our intuitions, we look at a more substantial example, which also extends the usage of the expression language for unary natural numbers.

Peano Axioms in Mathtalk

▷ **Example 50** We can write the Peano Axioms in mathtalk: If we write $n \in \mathbb{N}_1$ for *n is a unary natural number*, and $P(n)$ for *n has property P*, then we can write

1. $o \in \mathbb{N}_1$ (zero is a unary natural number)
2. $\forall n \in \mathbb{N}_1.s(n) \in \mathbb{N}_1 \wedge n \neq s(n)$ ($\mathbb{N}_1$closed under successors, distinct)
3. $\neg(\exists n \in \mathbb{N}_1.o = s(n))$ (zero is not a successor)
4. $\forall n \in \mathbb{N}_1.\forall m \in \mathbb{N}_1.n \neq m \Rightarrow s(n) \neq s(m)$ (different successors)
5. $\forall P.(P(o) \wedge (\forall n \in \mathbb{N}_1.P(n) \Rightarrow P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}_1.P(m))$ (induction)

We will use mathematical vernacular throughout the remainder of the notes. The abbreviations will mostly be used in informal communication situations. Many mathematicians consider it bad style to use abbreviations in printed text, but approve of them as parts of formulae (see e.g. Definition 46 for an example).

Mathematics uses a very effective technique for dealing with conceptual complexity. It usually starts out with discussing simple, *basic* objects and their properties. These simple objects can be combined to more complex, *compound* ones. Then it uses a definition to give a compound object a new name, so that it can be used like a basic one. In particular, the newly defined object can be used to form compound objects, leading to more and more complex objects that can be described succinctly. In this way mathematics incrementally extends its vocabulary by add layers and layers of definitions onto very simple and basic beginnings. We will now discuss four definition schemata that will occur over and over in this course.

Definition 51 The simplest form of definition schema is the simple definition. This just introduces a name (the definiendum) for a compound object (the definiens). Note that the name must be new, i.e. may not have been used for anything else, in particular, the definiendum may not occur in the definiens. We use the symbols $:=$ (and the inverse $=:$) to denote simple definitions in formulae.

Example 52 We can give the unary natural number $////$ the name φ. In a formula we write this as $\varphi := ////$ or $//// =: \varphi$.

Definition 53 A somewhat more refined form of definition is used for operators on and relations between objects. In this form, then definiendum is the operator or relation is applied to n distinct variables $v_1, \ldots, v_n$ as arguments, and the definiens is an expression in these variables. When the new operator is applied to arguments $a_1, \ldots, a_n$, then its value is the definiens expression where the v_i are replaced by the a_i. We use the symbol $:=$ for operator definitions and $:\Leftrightarrow$ for pattern definitions.[2] EdN:2

Example 54 The following is a pattern definition for the set intersection operator $\cap$:

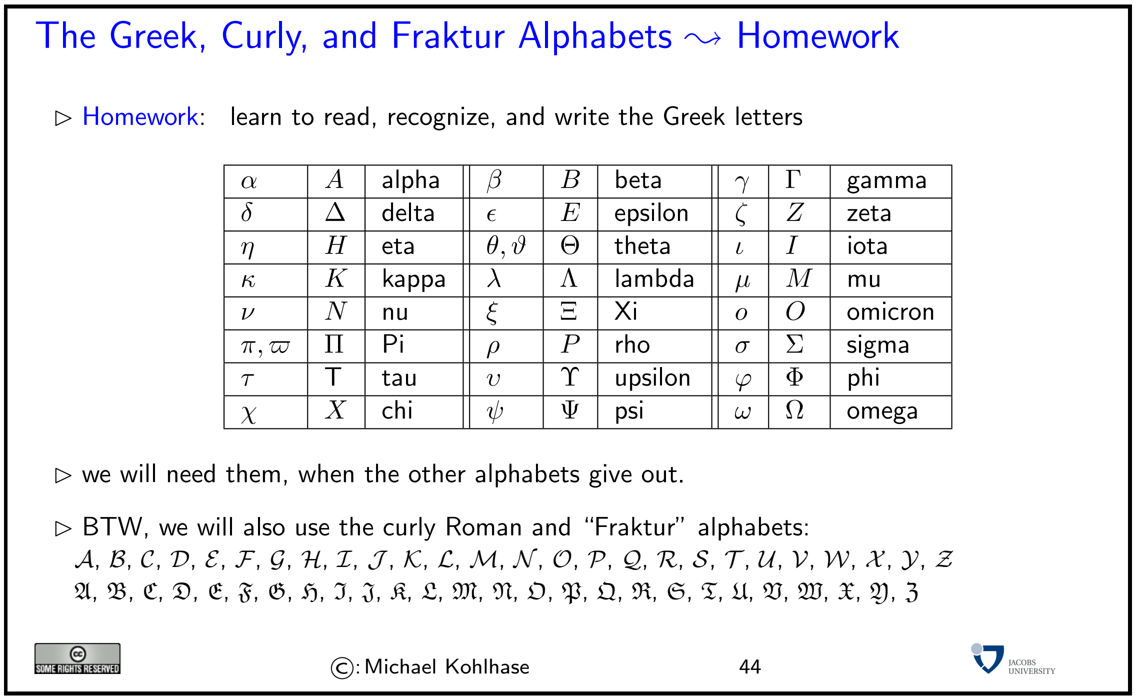
$$A \cap B := \{x \mid x \in A \wedge x \in B\}$$

The pattern variables are A and B, and with this definition we have e.g. $\emptyset \cap \emptyset = \{x \mid x \in \emptyset \wedge x \in \emptyset\}$.

Definition 55 We now come to a very powerful definition schema. An implicit definition (also called definition by description) is a formula $\mathbf{A}$, such that we can prove $\exists^1 n.\mathbf{A}$, where n is a new name.

Example 56 $\forall x.x \notin \emptyset$ is an implicit definition for the empty set $\emptyset$. Indeed we can prove unique existence of $\emptyset$ by just exhibiting $\{\}$ and showing that any other set S with $\forall x.x \notin S$ we have $S \equiv \emptyset$. Indeed$S$ cannot have elements, so it has the same elements ad $\emptyset$, and thus $S \equiv \emptyset$.

To keep mathematical formulae readable (they are bad enough as it is), we like to express mathematical objects in single letters. Moreover, we want to choose these letters to be easy to remember; e.g. by choosing them to remind us of the name of the object or reflect the kind of object (is it a number or a set, ...). Thus the 50 (upper/lowercase) letters supplied by most alphabets are not sufficient for expressing mathematics conveniently. Thus mathematicians and computer scientists use at least two more alphabets.



The Greek, Curly, and Fraktur Alphabets $\rightsquigarrow$ Homework

▷ Homework: learn to read, recognize, and write the 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	Θ	theta	ι	I	iota
κ	K	kappa	λ	Λ	lambda	μ	M	mu
ν	N	nu	ξ	Ξ	Xi	o	O	omicron
π, 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

▷ we will need them, when the other alphabets give out.

▷ BTW, we will also use the curly Roman and "Fraktur" alphabets:
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}, \mathcal{M}, \mathcal{N}, \mathcal{O}, \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$
$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}, \mathfrak{E}, \mathfrak{F}, \mathfrak{G}, \mathfrak{H}, \mathfrak{I}, \mathfrak{J}, \mathfrak{K}, \mathfrak{L}, \mathfrak{M}, \mathfrak{N}, \mathfrak{O}, \mathfrak{P}, \mathfrak{Q}, \mathfrak{R}, \mathfrak{S}, \mathfrak{T}, \mathfrak{U}, \mathfrak{V}, \mathfrak{W}, \mathfrak{X}, \mathfrak{Y}, \mathfrak{Z}$

[2] EdNote: maybe better markup up pattern definitions as binding expressions, where the formal variables are bound.

To be able to *read and understand* math and computer science texts profitably it is only only important to recognize the Greek alphabet, but also to know about the correspondences with the Roman one. For instance, ν corresponds to the n, so we often use ν as names for objects we would otherwise use n for (but cannot).

To be able to *talk about* math and computerscience, we also have to be able to pronounce the Greek letters, otherwise we embarrass ourselves by saying something like "the funny Greek letter that looks a bit like a w".

3.5 Naive Set Theory

We now come to a very important and foundational aspect in Mathematics: Sets. Their importance comes from the fact that all (known) mathematics can be reduced to understanding sets. So it is important to understand them thoroughly before we move on.

But understanding sets is not so trivial as it may seem at first glance. So we will just represent sets by various descriptions. This is called "naive set theory", and indeed we will see that it leads us in trouble, when we try to talk about very large sets.

Understanding Sets

- ▷ Sets are one of the foundations of mathematics,
- ▷ and one of the most difficult concepts to get right axiomatically
- ▷ Early Definition Attempt: A set is "everything that can form a unity in the face of God". (Georg Cantor ($*$(1845), †(1918)))
- ▷ For this course: no definition; just intuition (naive set theory)
- ▷ To understand a set S, we need to determine, what is an element of S and what isn't.
- ▷ We can represent sets by
  - ▷ listing the elements within curly brackets: e.g. $\{a, b, c\}$
  - ▷ describing the elements via a property: $\{x \mid x \text{ has property } P\}$
  - ▷ stating element-hood ($a \in S$) or not ($b \not\in S$).
- ▷ **Axiom 57** Every set we can write down actually exists! (Hidden Assumption)

Warning: Learn to distinguish between objects and their representations!($\{a, b, c\}$ and $\{b, a, a, c\}$ are different representa

Indeed it is very difficult to define something as foundational as a set. We want sets to be collections of objects, and we want to be as unconstrained as possible as to what their elements can be. But what then to say about them? Cantor's intuition is one attempt to do this, but of course this is not how we want to define concepts in math.

a

A b

b

So instead of defining sets, we will directly work with representations of sets. For that we only have to agree on how we can write down sets. Note that with this practice, we introduce a hidden assumption: called set comprehension, i.e. that every set we can write down actually exists. We will see below that we cannot hold this assumption.

Now that we can represent sets, we want to compare them. We can simply define relations between sets using the three set description operations introduced above.

▷ Relations between Sets

- ▷ set equality: $A \equiv B :\Leftrightarrow \forall x.x \in A \Leftrightarrow x \in B$
- ▷ subset: $A \subseteq B :\Leftrightarrow \forall x.x \in A \Rightarrow x \in B$
- ▷ proper subset: $A \subset B :\Leftrightarrow (A \subseteq B) \wedge (A \not\equiv B)$
- ▷ superset: $A \supseteq B :\Leftrightarrow \forall x.x \in B \Rightarrow x \in A$
- ▷ proper superset: $A \supset B :\Leftrightarrow (A \supseteq B) \wedge (A \not\equiv B)$

We want to have some operations on sets that let us construct new sets from existing ones. Again, can define them.

Operations on Sets

- ▷ union: $A \cup B := \{x \,|\, x \in A \vee x \in BQQ\}$
- ▷ union over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcup_{i \in I} S_i := \{x \,|\, \exists i \in I.x \in S_i\}$.
- ▷ intersection: $A \cap B := \{x \,|\, x \in A \wedge x \in B\}$
- ▷ intersection over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcap_{i \in I} S_i := \{x \,|\, \forall i \in I.x \in S_i\}$.
- ▷ set difference: $A \backslash B := \{x \,|\, x \in A \wedge x \notin B\}$
- ▷ the power set: $\mathcal{P}(A) := \{S \,|\, S \subseteq A\}$
- ▷ the empty set: $\forall x.x \notin \emptyset$
- ▷ Cartesian product: $A \times B := \{\langle a, b\rangle \,|\, a \in A \wedge b \in B\}$, call $\langle a, b\rangle$ pair.
- ▷ n-fold Cartesian product: $A_1 \times \cdots \times A_n := \{\langle a_1, \ldots, a_n\rangle \,|\, \forall i.1 \leq i \leq n \Rightarrow a_i \in A_i\}$, call $\langle a_1, \ldots, a_n\rangle$ an n-tuple
- ▷ n-dim Cartesian space: $A^n := \{\langle a_1, \ldots, a_n\rangle \,|\, 1 \leq i \leq n \Rightarrow a_i \in A\}$, call $\langle a_1, \ldots, a_n\rangle$ a vector
- ▷ **Definition 58** We write $\bigcup_{i=1}^n S_i$ for $\bigcup_{i \in \{i \in \mathbb{N} \,|\, 1 \leq i \leq n\}} S_i$ and $\bigcap_{i=1}^n S_i$ for $\bigcap_{i \in \{i \in \mathbb{N} \,|\, 1 \leq i \leq n\}} S_i$.

Finally, we would like to be able to talk about the number of elements in a set. Let us try to define that.

Sizes of Sets

- ▷ We would like to talk about the size of a set. Let us try a definition

- **Definition 59** The size $\#(A)$ of a set A is the number of elements in A.
- **Conjecture 60** *Intuitively we should have the following identities:*
  - $\#(\{a,b,c\})=3$
  - $\#(\mathbb{N})=\infty$ (*infinity*)
  - $\#(A\cup B)\leq\#(A)+\#(B)$ (⚠ *cases with* ∞)
  - $\#(A\cap B)\leq min(\#(A),\#(B))$
  - $\#(A\times B)=\#(A)\cdot\#(B)$
- But how do we prove any of them? (what does "number of elements" mean anyways?)
- Idea: We need a notion of "counting", associating every member of a set with a unary natural number.
- Problem: How do we "associate elements of sets with each other"?(wait for bijective functions)

Once we try to prove the identifies from Conjecture 60 we get into problems. Even though the notion of "counting the elements of a set" is intuitively clear (indeed we have been using that since we were kids), we do not have a mathematical way of talking about associating numbers with objects in a way that avoids double counting and skipping. We will have to postpone the discussion of sizes until we do.

But before we delve in to the notion of relations and functions that we need to associate set members and counting let us now look at large sets, and see where this gets us.

Sets can be Mind-boggling

- sets seem so simple, but are really quite powerful (no restriction on the elements)
- There are very large sets, e.g. "the set $\mathcal{S}$ of all sets"
  - contains the $\emptyset$,
  - for each object O we have $\{O\},\{\{O\}\},\{O,\{O\}\},\ldots\in\mathcal{S}$,
  - contains all unions, intersections, power sets,
  - contains itself: $\mathcal{S}\in\mathcal{S}$ (scary!)
- Let's make $\mathcal{S}$ less scary

A less scary $\mathcal{S}$?

- Idea: how about the "set $\mathcal{S}'$ of all sets that do not contain themselves"
- Question: is $\mathcal{S}'\in\mathcal{S}'$? (were we successful?)
  - suppose it is, then then we must have $\mathcal{S}'\notin\mathcal{S}'$, since we have explicitly taken out the sets that contain themselves

  - ▷ suppose it is not, then have $\mathcal{S}' \in \mathcal{S}'$, since all other sets are elements.

  In either case, we have $\mathcal{S}' \in \mathcal{S}'$ iff $\mathcal{S}' \notin \mathcal{S}'$, which is a contradiction!(Russell's Antinomy [Bertrand Russell '03])

- ▷ Does MathTalk help?: no: $\mathcal{S}' := \{m \mid m \notin m\}$
  - ▷ MathTalk allows statements that lead to contradictions, but are legal wrt. "vocabulary" and "grammar".
- ▷ We have to be more careful when constructing sets! (axiomatic set theory)
- ▷ for now: stay away from large sets. (stay naive)

Even though we have seen that naive set theory is inconsistent, we will use it for this course. But we will take care to stay away from the kind of large sets that we needed to construct the paradox.

3.6 Relations and Functions

Now we will take a closer look at two very fundamental notions in mathematics: functions and relations. Intuitively, functions are mathematical objects that take arguments (as input) and return a result (as output), whereas relations are objects that take arguments and state whether they are related.

We have already encountered functions and relations as set operations — e.g. the elementhood relation $\in$ which relates a set to its elements or the power set function that takes a set and produces another (its power set).

Relations

- ▷ **Definition 61** $R \subseteq A \times B$ is a (binary) relation between A and B.
- ▷ **Definition 62** If $A = B$ then R is called a relation on A.
- ▷ **Definition 63** $R \subseteq A \times B$ is called total iff $\forall x \in A.\exists y \in B.\langle x, y\rangle \in R$.
- ▷ **Definition 64** $R^{-1} := \{\langle y, x\rangle \mid \langle x, y\rangle \in R\}$ is the converse relation of R.
- ▷ Note: $R^{-1} \subseteq B \times A$.
- ▷ The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is defined as $(S \circ R) := \{\langle a, c\rangle \in (A \times C) \mid \exists b \in B.\langle a, b\rangle \in R \wedge \langle b, c\rangle \in$
- ▷ **Example 65** relation $\subseteq$, $=$, has_color
- ▷ Note: we do not really need ternary, quaternary, ... relations
  - ▷ Idea: Consider $A \times B \times C$ as $A \times (B \times C)$ and $\langle a, b, c\rangle$ as $\langle a, \langle b, c\rangle\rangle$
  - ▷ we can (and often will) see $\langle a, b, c\rangle$ as $\langle a, \langle b, c\rangle\rangle$ different representations of the same object.

We will need certain classes of relations in following, so we introduce the necessary abstract properties of relations.

Properties of binary Relations

▷ **Definition 66 (Relation Properties)** A relation $R \subseteq A \times A$ is called

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

▷ **Example 67** The equality relation is an equivalence relation on any set.

▷ **Example 68** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.

These abstract properties allow us to easily define a very important class of relations, the ordering relations.

Strict and Non-Strict Partial Orders

▷ **Definition 69** A relation $R \subseteq A \times A$ is called

- ▷ partial order on A, iff R is reflexive, antisymmetric, and transitive on A.
- ▷ strict partial order on A, iff it is irreflexive and transitive on A.

▷ In contexts, where we have to distinguish between strict and non-strict ordering relations, we often add an adjective like "non-strict" or "weak" or "reflexive" to the term "partial order". We will usually write strict partial orderings with asymmetric symbols like $\prec$, and non-strict ones by adding a line that reminds of equality, e.g. $\preceq$.

▷ **Definition 70 (Linear order)** A partial order is called linear on A, iff all elements in A are comparable, i.e. if $\langle x,y\rangle \in R$ or $\langle y,x\rangle \in R$ for all $x,y \in A$.

▷ **Example 71** The $\leq$ relation is a linear order on $\mathbb{N}$ (all elements are comparable)

▷ **Example 72** The "ancestor-of" relation is a partial order that is not linear.

▷ **Lemma 73** *Strict partial orderings are asymmetric.*

▷ Proof Sketch: By contradiction: If $\langle a,b\rangle \in R$ and $\langle b,a\rangle \in R$, then $\langle a,a\rangle \in R$ by transitivity ↯

▷ **Lemma 74** *If $\preceq$ is a (non-strict) partial order, then $\prec := \{\langle a,b\rangle \,|\, (a \preceq b) \wedge a \neq b\}$ is a strict partial order. Conversely, if $\prec$ is a strict partial order, then $\preceq := \{\langle a,b\rangle \,|\, (a \prec b) \vee a = b\}$ is a non-strict partial order.*

Functions (as special relations)

▷ **Definition 75** $f \subseteq X \times Y$, is called a partial function, iff for all $x \in X$ there is at most one $y \in Y$ with $\langle x, y\rangle \in f$.

- ▹ **Notation 76** $f\colon X \rightharpoonup Y; x \mapsto y$ if $\langle x, y\rangle \in f$ (arrow notation)
- ▹ call X the domain (write $\mathbf{dom}(f)$), and Y the codomain $(\mathbf{codom}(f))$ (come with f)
- ▹ **Notation 77** $f(x) = y$ instead of $\langle x, y\rangle \in f$ (function application)

▷ **Definition 78** We call a partial function $f\colon X \rightharpoonup Y$ undefined at $x \in X$, iff $\langle x, y\rangle \notin f$ for all $y \in Y$. (write $f(x) = \perp$)

▷ **Definition 79** If $f\colon X \rightharpoonup Y$ is a total relation, we call f a total function and write $f\colon X \to Y$. $(\forall x \in X.\exists^1 y \in Y.\langle x, y\rangle \in f)$

- ▹ **Notation 80** $f\colon x \mapsto y$ if $\langle x, y\rangle \in f$ (arrow notation)

⚠: this probably does not conform to your intuition about functions. Do not worry, just think of them as two different things they will come together over time.(In this course we will use "function" as defined here!)

▷ Function Spaces

▷ **Definition 81** Given sets A and B We will call the set $A \to B$ $(A \rightharpoonup B)$ of all (partial) functions from A to B the (partial) function space from A to B.

▷ **Example 82** Let $\mathbb{B} := \{0, 1\}$ be a two-element set, then

$$\mathbb{B} \to \mathbb{B} = \{\{\langle 0,0\rangle, \langle 1,0\rangle\}, \{\langle 0,1\rangle, \langle 1,1\rangle\}, \{\langle 0,1\rangle, \langle 1,0\rangle\}, \{\langle 0,0\rangle, \langle 1,1\rangle\}\}$$

$$\mathbb{B} \rightharpoonup \mathbb{B} = \mathbb{B} \to \mathbb{B} \cup \{\emptyset, \{\langle 0,0\rangle\}, \{\langle 0,1\rangle\}, \{\langle 1,0\rangle\}, \{\langle 1,1\rangle\}\}$$

▷ as we can see, all of these functions are finite (as relations)

Lambda-Notation for Functions

▷ Problem: It is common mathematical practice to write things like $f_a(x) = ax^2 + 3x + 5$, meaning e.g. that we have a collection $\{f_a \mid a \in A\}$ of functions.(is a an argument or jut a "parameter"?)

▷ **Definition 83** To make the role of arguments extremely clear, we write functions in λ-notation. For $f = \{\langle x, E\rangle \mid x \in X\}$, where E is an expression, we write $\lambda x \in X.E$.

▷ **Example 84** The simplest function we always try everything on is the identity function:

$$\begin{aligned} \lambda n \in \mathbb{N}.n &= \{\langle n, n\rangle \mid n \in \mathbb{N}\} = \mathsf{Id}_{\mathbb{N}} \\ &= \{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \ldots\} \end{aligned}$$

▷ **Example 85** We can also to more complex expressions, here we take the square function

$$\begin{aligned}\lambda x \in \mathbb{N}.x^2 &= \{\langle x, x^2\rangle \,|\, x \in \mathbb{N}\}\\ &= \{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,4\rangle, \langle 3,9\rangle, \ldots\}\end{aligned}$$

▷ **Example 86** λ-notation also works for more complicated domains. In this case we have pairs as arguments.

$$\begin{aligned}\lambda\langle x,y\rangle \in \mathbb{N}\times\mathbb{N}.x+y &= \{\langle\langle x,y\rangle, x+y\rangle \,|\, x \in \mathbb{N} \wedge y \in \mathbb{N}\}\\ &= \{\langle\langle 0,0\rangle,0\rangle, \langle\langle 0,1\rangle,1\rangle, \langle\langle 1,0\rangle,1\rangle,\\ &\qquad \langle\langle 1,1\rangle,2\rangle, \langle\langle 0,2\rangle,2\rangle, \langle\langle 2,0\rangle,2\rangle, \ldots\}\end{aligned}$$

EdN:3 [3]

The three properties we define next give us information about whether we can invert functions.

Properties of functions, and their converses

▷ **Definition 87** A function $f\colon S \to T$ is called

- ▹ injective iff $\forall x, y \in S.f(x) = f(y) \Rightarrow x = y$.
- ▹ surjective iff $\forall y \in T.\exists x \in S.f(x) = y$.
- ▹ bijective iff f is injective and surjective.

Note: If f is injective, then the converse relation f^{-1} is a partial function.

⧐ Note: If f is surjective, then the converse f^{-1} is a total relation.

▷ **Definition 88** If f is bijective, call the converse relation f^{-1} the inverse function.

▷ Note: if f is bijective, then the converse relation f^{-1} is a total function.

▷ **Example 89** The function $\nu\colon \mathbb{N}_1 \to \mathbb{N}$ with $\nu(o) = 0$ and $\nu(s(n)) = \nu(n) + 1$ is a bijection between the unary natural numbers and the natural numbers from highschool.

▷ Note: Sets that can be related by a bijection are often considered equivalent, and sometimes confused. We will do so with $\mathbb{N}_1$ and $\mathbb{N}$ in the future

Cardinality of Sets

▷ Now, we can make the notion of the size of a set formal, since we can associate members of sets by bijective functions.

▷ **Definition 90** We say that a set A is finite and has cardinality $\#(A) \in \mathbb{N}$, iff there is a bijective function $f\colon A \to \{n \in \mathbb{N} \,|\, n < \#(A)\}$.

▷ **Definition 91** We say that a set A is countably infinite, iff there is a bijective function

[3] EdNote: define Idon and Bool somewhere else and import it here

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

▷ **Theorem 92** *We have the following identities for finite sets A and B*

- ▹ $\#(\{a,b,c\}) = 3$ *(e.g. choose $f = \{\langle a,0\rangle, \langle b,1\rangle, \langle c,2\rangle\}$)*
- ▹ $\#(A \cup B) \leq \#(A) + \#(B)$
- ▹ $\#(A \cap B) \leq \mathit{min}(\#(A), \#(B))$
- ▹ $\#(A \times B) = \#(A) \cdot \#(B)$

▷ With the definition above, we can prove them (last three $\leadsto$ Homework)

Next we turn to a higher-order function in the wild. The composition function takes two functions as arguments and yields a function as a result.

Operations on Functions

▷ **Definition 93** If $f \in A \to B$ and $g \in B \to C$ are functions, then we call

$$g \circ f: A \to C; x \mapsto g(f(x))$$

the composition of g and f (read g "after" f).

▷ **Definition 94** Let $f \in A \to B$ and $C \subseteq A$, then we call the relation $\{\langle c,b\rangle \,|\, c \in C \wedge \langle c,b\rangle \in f$ the restriction of f to C.

▷ **Definition 95** Let $f: A \to B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call

- ▹ $f(A') := \{b \in B \,|\, \exists a \in A'.\langle a,b\rangle \in f\}$ the image of A' under f,
- ▹ $\mathbf{Im}(f) := f(A)$ the image of f, and
- ▹ $f^{-1}(B') := \{a \in A \,|\, \exists b \in B'.\langle a,b\rangle \in f\}$ the pre-image of B' under f

Bibliography

[Den00] Peter Denning. Computer science: The discipline. In A. Ralston and D. Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 405–419. Nature Publishing Group, 2000.

[Hal74] Paul R. Halmos. *Naive Set Theory*. Springer Verlag, 1974.

[Koh08] Michael Kohlhase. Using LaTeX as a semantic markup format. *Mathematics in Computer Science*, 2(2):279–304, 2008.

[Koh11a] Michael Kohlhase. General Computer Science; Problems for 320101 GenCS I. Online practice problems at `http://kwarc.info/teaching/GenCS1/problems.pdf`, 2011.

[Koh11b] Michael Kohlhase. General Computer Science: Problems for 320201 GenCS II. Online practice problems at `http://kwarc.info/teaching/GenCS2/problems.pdf`, 2011.

[Koh12] Michael Kohlhase. sTeX: Semantic markup in TeX/LaTeX. Technical report, Comprehensive TeX Archive Network (CTAN), 2012.

[KP95] Paul Keller and Wolfgang Paul. *Hardware Design*. Teubner Leibzig, 1995.

[LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1998.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence — A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[Ros90] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 1990.

[Smo08] Gert Smolka. *Programmierung - eine Einführung in die Informatik mit Standard ML*. Oldenbourg, 2008.