General Computer Science
320201 GenCS I & II Lecture Notes

Michael Kohlhase

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

May 27, 2012

Chapter 1

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 2003-2012.

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.

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. [Gen11a, Gen11b]

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

[1] International University Bremen until Fall 2006

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

To reach the aim of giving 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.

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. EdNote: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

[1] EdNote: termrefs!

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 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.

Contents

Chapter 2

Representation and Computation

2.1 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".

2.1.1 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`

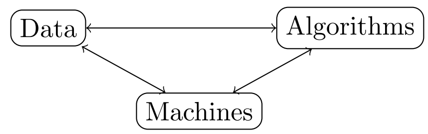


Figure 2.1: The three key ingredients of Computer Science

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.

2.1.2 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.

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 otherwise.)

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.

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.

 4 JACOBS UNIVERSITY

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)

 5 JACOBS UNIVERSITY

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 faculty (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)
  - 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

 6 JACOBS UNIVERSITY

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.

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

[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,. . .

- 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 actual words)
- more on this as the semester goes on ...

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

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.

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` (follow instructions there)
- if there are problems send e-mail to `c.david@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 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://trac.mathweb.org/planetary`)
  - ▹ 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)

 10 JACOBS UNIVERSITY

2.1.3 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).

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?

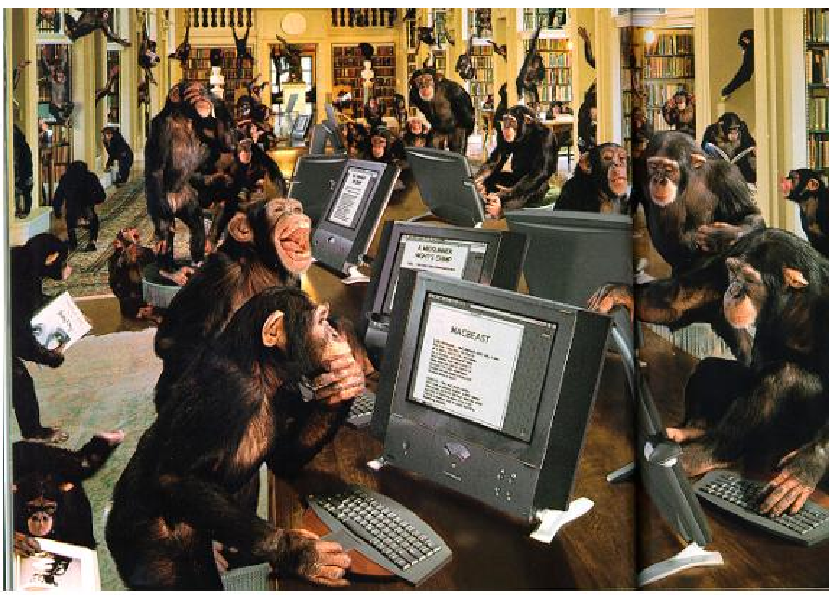
There are of course various ways to build such a a maze; one would be to ask the students from biology to come and plant some hedges, and have them re-plant them until the maze meets our criteria. A better way would be to make a plan first, i.e. to get a large piece of paper, and draw a maze before we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

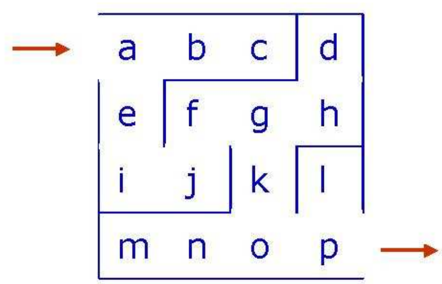
▷ *"Applets, Not Craplets^{tm}"* (– ca. 1997)

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".

 16 JACOBS UNIVERSITY

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.

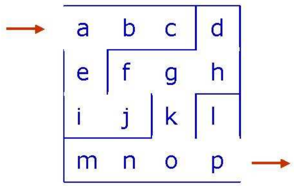
As mathematical models have very few properties (this is deliberate, so that we can understand all of them), we can use them as models for many concrete, real-world situations.

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 algorithms)
- ▷ **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

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

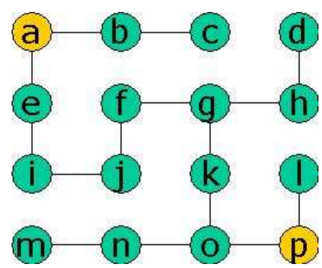
Mazes as Graphs (Visualizing Graphs via Diagrams)

▷ 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

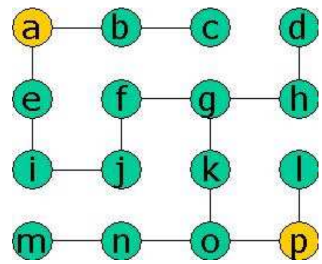


▷ Note that the diagram is a visualization (a representation intended for humans to process visually) of the graph, and not the graph itself.

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.

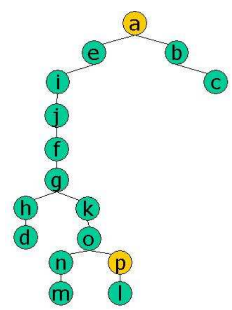
Unique solutions

- ▷ 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.

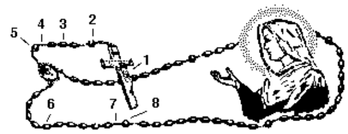
  Q: Why is it good to have a spanning tree?
- ⊳▷ A: Trees have no cycles! (needed for uniqueness)
- ▷ A: Every room is reachable from the root!

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?

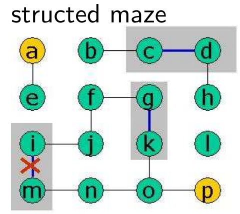
▷ When adding wall $\langle x, y\rangle$, we want to know if there is already a path from x to y in the tree.

▷ 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



 22 JACOBS UNIVERSITY

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.

 23 JACOBS UNIVERSITY

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	...	...	...

 24 JACOBS UNIVERSITY

What?! One year?

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

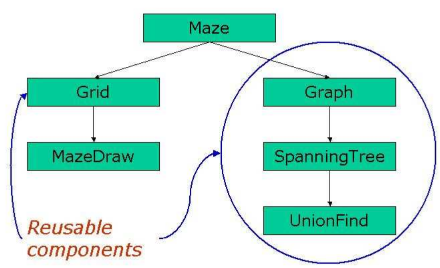
n	$100n$ μs	$7n^2$ μs	2^n μs
1	100 μs	7 μs	2 μs
5	.5 ms	175 μs	32 μs
10	1 ms	.7 ms	1 ms
45	4.5 ms	14 ms	1.1 years
100	100 ms	7 s	10^{16} years
1 000	1 s	12 min	–
10 000	10 s	20 h	–
1 000 000	1.6 min	2.5 mo	–

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.

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".

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.



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

Let us summarize!

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.

 28 JACOBS UNIVERSITY

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.

2.2 Elementary Discrete Math

2.2.1 Mathematical Foundations: Natural Numbers

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.

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 natural numbers.

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.

A little more sophistication (math) please

- ▷ **Definition 17** call /// the successor of // and // the predecessor of /// (successors are created by *s*-rule)
- ▷ **Definition 18** The following set of axioms are called the Peano Axioms (Giuseppe Peano $*(1858)$, $\dagger(1932)$)

▷ **Axiom 19 ($P1$)** " " (aka. "zero") is a unary natural number.

▷ **Axiom 20 ($P2$)** Every unary natural number has a successor that is a unary natural number and that is different from it.

▷ **Axiom 21 ($P3$)** Zero is not a successor of any unary natural number.

▷ **Axiom 22 ($P4$)** Different unary natural numbers have different predecessors.

▷ **Axiom 23 ($P5$: induction)** 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?)

▷ **Definition 24** In general, an 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.

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).

Let's practice derivations and proofs

▷ **Example 28** //////////// is a unary natural number

▷ **Theorem 29** */// is a different unary natural number than //.*

▷ **Theorem 30** *///// is a different unary natural number than //.*

▷ **Theorem 31** *There is a unary natural number of which /// is the successor*

▷ **Theorem 32** *There are at least 7 unary natural numbers.*

▷ **Theorem 33** *Every unary natural number is either zero or the successor of a unary natural number.* (*we will come back to this later*)

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, ... as names for concrete unary natural numbers.)

▷ 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 34** 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 35** We denote the set of all unary natural numbers with $\mathbb{N}_1$. (either representation)

Induction for unary natural numbers

▷ **Theorem 36** *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 □

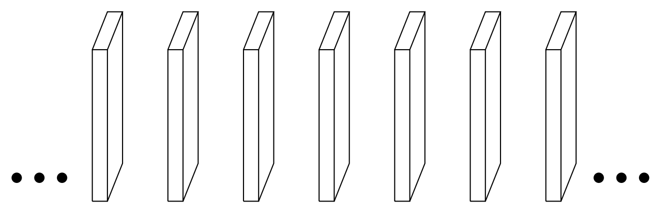
Theorem 36 is a very useful fact to know, it tells us something about the form of unary natural numbers, which lets us streamline induction proofs and bring them more into the form you may know from school: to show that some property P holds for every natural number, we analyze an arbitrary number n 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 37** *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.

What can we do with unary natural numbers?

- So far not much (let's introduce some operations)
- **Definition 38 (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?)

Addition on unary natural numbers is associative

- **Theorem 39** *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 $s((n \oplus m) \oplus l) = (n \oplus m) \oplus s(l)$ □

□

More Operations on Unary Natural Numbers

- **Definition 40** 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 41** 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 42** 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 43** The unary product operation can be defined by the equations $\bigodot_{i=o}^{o} n_i = s(o)$ and $\bigodot_{i=o}^{s(m)} n_i = n_{s(m)} \odot \bigodot_{i=o}^{m} n_i$.

2.2.2 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 44** Mathematicians use a stylized language that

- ▷ uses formulae to represent mathematical objects,[2] e.g. $\int_1^0 x^{\frac{3}{2}}\,dx$
- ▷ 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 45** Abbreviations for Mathematical statements

- ▷ $\wedge$ and "$\vee$" are common notations for "and" and "or"
- ▷ "not" is in mathematical statements often denoted with $\neg$
- ▷ $\forall x.P$ ($\forall x \in S.P$) 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 46** $\forall x.\exists y.x = y \Leftrightarrow \neg(x \neq y)$ reads

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

[b] EdNote: think about how to reactivate this example

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 2.2.3 for an example).

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 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:
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}, \mathcal{M}, \mathcal{N}, \mathcal{O}, \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$
$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}, \mathfrak{E}, \mathfrak{F}, \mathfrak{G}, \mathfrak{H}, \mathfrak{I}, \mathfrak{J}, \mathfrak{K}, \mathfrak{L}, \mathfrak{M}, \mathfrak{N}, \mathfrak{O}, \mathfrak{P}, \mathfrak{Q}, \mathfrak{R}, \mathfrak{S}, \mathfrak{T}, \mathfrak{U}, \mathfrak{V}, \mathfrak{W}, \mathfrak{X}, \mathfrak{Y}, \mathfrak{Z}$

On our way to understanding functions

We need to understand sets first.

2.2.3 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

▷ **Definition 47** 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.

▷ Notations for sets (so we can write them down)

- ▷ listing the elements within curly brackets: e.g. $\{a, b, c\}$
- ▷ to describe the elements by a property: $\{x \mid x \text{ has property } P\}$

- by stating element-hood ($a \in S$) or not ($b \notin S$).

Warning: Learn to distinguish between objects and their representations! ($\{a, b, c\}$ and $\{b, a, a, c\}$ are different representations of the same set)

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

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: $A \cup B := \{x \mid x \in A \vee x \in B\}$
- union over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcup_{i \in I} S_i := \{x \mid \exists i \in I.x \in S_i\}$.
- intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$
- intersection over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcap_{i \in I} S_i := \{x \mid \forall i \in I.x \in S_i\}$.
- set difference: $A \backslash B := \{x \mid x \in A \wedge x \notin B\}$
- the power set: $\mathcal{P}(A) := \{S \mid S \subseteq A\}$
- the empty set: $\forall x.x \notin \emptyset$
- Cartesian product: $A \times B := \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$, call $\langle a, b\rangle$ pair.
- n-fold Cartesian product: $A_1 \times \cdots \times A_n := \{\langle a_1, \ldots, a_n\rangle \mid \forall i.(1 \leq i \leq n) \Rightarrow a_i \in A_i\}$, call $\langle a_1, \ldots, a_n\rangle$ an n-tuple
- n-dim Cartesian space: $A^n := \{\langle a_1, \ldots, a_n\rangle \mid (1 \leq i \leq n) \Rightarrow a_i \in A\}$, call $\langle a_1, \ldots, a_n\rangle$ a vector
- **Definition 48** We write $\bigcup_{i=1}^n S_i$ for $\bigcup_{i \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\}} S_i$ and $\bigcap_{i=1}^n S_i$ for $\bigcap_{i \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\}} S_i$.

These operator definitions give us a chance to reflect on how we do definitions in mathematics.

Definitions in Mathtalk

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 49 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 50 We can give the unary natural number $////$ the name φ. In a formula we write this as $\varphi := ////$ or $//// =: \varphi$.

Definition 51 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.[3] EdNote:3

Example 52 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 53 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 54 $\forall x.x \notin \emptyset$ 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 $S \equiv \emptyset$.

Sizes of Sets

- We would like to talk about the size of a set. Let us try a definition
- **Definition 55** The size $\#(A)$ of a set A is the number of elements in A.
- Intuitively we should have the following identities:
  - $\#(\{a, b, c\}) = 3$
  - $\#(\mathbb{N}) = \infty$ (infinity)
  - $\#(A \cup B) \leq \#(A) + \#(B)$ (cases with ∞)
  - $\#(A \cap B) \leq \mathsf{min}(\#(A), \#(B))$
  - $\#(A \times B) = \#(A) \cdot \#(B)$
- But how do we prove any of them? (what does "number of elements" mean anyways?)

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

▷ 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)

But before we delve in to the notion of relations and functions that we need to associate set members and counding 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 constuct the paradoxon.

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

Relations

- **Definition 56** $R \subseteq A \times B$ is a (binary) relation between A and B.
- **Definition 57** If $A = B$ then R is called a relation on A.
- **Definition 58** $R \subseteq A \times B$ is called total iff $\forall x \in A.\exists y \in B.\langle x, y\rangle \in R$.
- **Definition 59** $R^{-1} := \{\langle y, x\rangle \mid \langle x, y\rangle \in R\}$ 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 $S \circ R := \{\langle a, c\rangle \in (A \times C) \mid \exists b \in B.\langle a, b\rangle \in R \wedge \langle b, c\rangle \in S\}$
- **Example 60** 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.

 50 JACOBS UNIVERSITY

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

Properties of binary Relations

- **Definition 61** A relation $R \subseteq A \times A$ is called
  - reflexive on A, iff $\forall a \in A.\langle a, a\rangle \in R$
  - symmetric on A, iff $\forall a, b \in A.\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \in R$
  - 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
  - partial order on A, iff R is reflexive, antisymmetric, and transitive on A.
  - a linear order on A, iff R is transitive and for all $x, y \in A$ with $x \neq y$ either $\langle x, y\rangle \in R$ or $\langle y, x\rangle \in R$
- **Example 62** The equality relation is an equivalence relation on any set.
- **Example 63** The $\leq$ relation is a linear order on $\mathbb{N}$ (all elements are comparable)

▷ **Example 64** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.
▷ **Example 65** On sets of persons, the "ancestor-of" relation is a partial order that is not linear.

Functions (as special relations)

▷ **Definition 66** $f \subseteq X \times Y$, 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 67** $f \colon X \rightharpoonup Y; x \mapsto y$ 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 68** $f(x) = y$ instead of $\langle x, y\rangle \in f$ (function application)

▷ **Definition 69** 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 70** 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 71** $f \colon x \mapsto y$ 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 72** 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 73** 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 74** 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 75** 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 76** We can also to more complex expressions, here we take the square function

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

▷ **Example 77** λ-notation also works for more complicated domains. In this case we have tuples as arguments.

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

 54 JACOBS UNIVERSITY

[4] EdNote:4

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

Properties of functions, and their converses

▷ **Definition 78** A function $f : S \to T$ 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 79** 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 80** The function $\nu : \mathbb{N}_1 \to \mathbb{N}$ 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

 55 JACOBS UNIVERSITY

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

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

▷ **Definition 82** We say that a set A is countably infinite, iff there is a bijective function $f: A \to \mathbb{N}$.

▷ **Theorem 83** *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 $\rightsquigarrow$ 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 84** 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 85** Let $f \in A \to B$ and $C \subseteq A$, then we call the relation $\{\langle c, b\rangle \mid c \in C \wedge \langle c, b\rangle \in f\}$ the restriction of f to C.

▷ **Definition 86** Let $f: A \to B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call $f(A') := \{b \in B \mid \exists a \in A'.\langle a, b\rangle \in f\}$ the image of A' under f and $f^{-1}(B') := \{a \in A \mid \exists b \in B'.\langle a, b\rangle \in f\}$ the pre-image of B' under f.

2.3 Computing with Functions over Inductively Defined Sets

2.3.1 Standard ML: Functions as First-Class Objects

Enough theory, let us start computing with functions

We will use Standard ML for now

 58

We will use the language SML for the course. This has three reasons

- The mathematical foundations of the computational model of SML is very simple: it consists of functions, which we have already studied. You will be exposed to an imperative programming language (C) in the lab and later in the course.
- We call programming languages where procedures can be fully described in terms of their input/output behavior functional.
- As a functional programming language, SML introduces two very important concepts in a very clean way: typing and recursion.
- Finally, SML has a very useful secondary virtue for a course at Jacobs University, where students come from very different backgrounds: it provides a (relatively) level playing ground, since it is unfamiliar to all students.

Generally, when choosing a programming language for a computer science course, there is the choice between languages that are used in industrial practice (C, C++, Java, FORTRAN, COBOL,...) and languages that introduce the underlying concepts in a clean way. While the first category have the advantage of conveying important practical skills to the students, we will follow the motto "No, let's think" for this course and choose ML for its clarity and rigor. In our experience, if the concepts are clear, adapting the particular syntax of a industrial programming language is not that difficult.

Historical Remark: The name ML comes from the phrase "Meta Language": ML was developed as the scripting language for a tactical theorem prover[3] — a program that can construct mathematical proofs automatically via "tactics" (little proof-constructing programs). The idea behind this is the following: ML has a very powerful type system, which is expressive enough to fully describe proof data structures. Furthermore, the ML compiler type-checks all ML programs and thus guarantees that if an ML expression has the type $A \to B$, then it implements a function from objects of type A to objects of type B. In particular, the theorem prover only admitted tactics, if they were type-checked with type $\mathcal{P} \to \mathcal{P}$, where $\mathcal{P}$ is the type of proof data structures. Thus, using ML as a meta-language guaranteed that theorem prover could only construct valid proofs.

The type system of ML turned out to be so convenient (it catches many programming errors before you even run the program) that ML has long transcended its beginnings as a scripting language for theorem provers, and has developed into a paradigmatic example for functional programming languages.

Standard ML (SML)

- ▷ Why this programming language?
  - ▷ Important programming paradigm (Functional Programming (with static typing))
  - ▷ because all of you are unfamiliar with it (level playing ground)

[3] The "Edinburgh LCF" system

- ▷ clean enough to learn important concepts (e.g. typing and recursion)
- ▷ SML uses functions as a computational model (we already understand them)
- ▷ SML has an interpreted runtime system (inspect program state)

Book: SML for the working programmer by Larry Paulson

▷ Web resources: see the post on the course forum

▷ Homework: install it, and play with it at home!

Disclaimer: We will not give a full introduction to SML in this course, only enough to make the course self-contained. There are good books on ML and various web resources:

- A book by Bob Harper (CMU) `http://www-2.cs.cmu.edu/~rwh/smlbook/`
- The Moscow ML home page, one of the ML's that you can try to install, it also has many interesting links `http://www.dina.dk/~sestoft/mosml.html`
- The home page of SML-NJ (SML of New Jersey), the standard ML `http://www.smlnj.org/` also has a ML interpreter and links Online Books, Tutorials, Links, FAQ, etc. And of course you can download SML from there for Unix as well as for Windows.
- A tutorial from Cornell University. It starts with "Hello world" and covers most of the material we will need for the course. `http://www.cs.cornell.edu/gries/CSCI4900/ML/gimlFolder/manual.html`
- and finally a page on ML by the people who originally invented ML: `http://www.lfcs.inf.ed.ac.uk/software/ML/`

One thing that takes getting used to is that SML is an interpreted language. Instead of transforming the program text into executable code via a process called "compilation" in one go, the SML interpreter provides a run time environment that can execute well-formed program snippets in a dialogue with the user. After each command, the state of the run-time systems can be inspected to judge the effects and test the programs. In our examples we will usually exhibit the input to the interpreter and the system response in a program block of the form

```
- input to the interpreter
system response
```

Programming in SML (Basic Language)

▷ Generally: start the SML interpreter, play with the program state.

▷ **Definition 87 (Predefined objects in SML)** (SML comes with a basic inventory)

- ▷ basic types `int`, `real`, `bool`, `string` , ...
- ▷ basic type constructors `->`, `*`,
- ▷ basic operators numbers, `true`, `false`, `+`, `*`, `-`, `>`, `^`, ... (⚠ overloading)
- ▷ control structures `if` Φ `then` E_1 `else` E_2`;`
- ▷ comments *(*this is a comment *)*

One of the most conspicuous features of SML is the presence of types everywhere.

Definition 88 types are program constructs that classify program objects into categories.

In SML, literally every object has a type, and the first thing the interpreter does is to determine the type of the input and inform the user about it. If we do something simple like typing a number (the input has to be terminated by a semicolon), then we obtain its type:

```
- 2;
val it = 2 : int
```

In other words the SML interpreter has determined that the input is a value, which has type "integer". At the same time it has bound the identifier `it` to the number 2. Generally `it` will always be bound to the value of the last successful input. So we can continue the interpreter session with

```
- it;
val it = 2 : int
- 4.711;
val it = 4.711 : real
- it;
val it = 4.711 : real
```

Programming in SML (Declarations)

- **Definition 89 (Declarations)** allow abbreviations for convenience
  - value declarations `val pi = 3.1415;`
  - type declarations `type twovec = int * int;`
  - function declarations `fun square (x:real) = x*x;` (leave out type, if unambiguous)
- SML functions that have been declared can be applied to arguments of the right type, e.g. `square 4.0`, which evaluates to `4.0 * 4.0` and thus to `16.0`.
- Local declarations: allow abbreviations in their scope (delineated by `in` and `end`)

```
- val test = 4;
val it = 4 : int
- let val test = 7 in test * test end;
val it = 49 :int
- test;
val it = 4 : int
```

  JACOBS UNIVERSITY

While the previous inputs to the interpreters do not change its state, declarations do: they bind identifiers to values. In the first example, the identifier `twovec` to the type `int * int`, i.e. the type of pairs of integers. Functions are declared by the `fun` keyword, which binds the identifier behind it to a function object (which has a type; in our case the function type `real -> real`). Note that in this example we annotated the formal parameter of the function declaration with a type. This is always possible, and in this necessary, since the multiplication operator is overloaded (has multiple types), and we have to give the system a hint, which type of the operator is actually intended.

Programming in SML (Pattern Matching)

- Component Selection: (very convenient)

```
- val unitvector = (1,1);
val unitvector = (1,1) : int * int
- val (x,y) = unitvector
val x = 1 : int
val y = 1 : int
```

▷ **Definition 90** anonymous variables (if we are not interested in one value)

```
- val (x,_) = unitvector;
val x = 1 :int
```

▷ **Example 91** We can define the selector function for pairs in SML as

```
- fun first (p) = let val (x,_) = p in x end;
val first = fn : 'a * 'b -> 'a
```

Note the type: SML supports universal types with type variables `'a`, `'b`,. . . .

⊳⊳ `first` is a function that takes a pair of type `'a*'b` as input and gives an object of type `'a` as output.

 62 JACOBS UNIVERSITY

Another unusual but convenient feature realized in SML is the use of pattern matching. In pattern matching we allow to use variables (previously unused identifiers) in declarations with the understanding that the interpreter will bind them to the (unique) values that make the declaration true. In our example the second input contains the variables `x` and `y`. Since we have bound the identifier `unitvector` to the value `(1,1)`, the only way to stay consistent with the state of the interpreter is to bind both `x` and `y` to the value `1`.

Note that with pattern matching we do not need explicit selector functions, i.e. functions that select components from complex structures that clutter the namespaces of other functional languages. In SML we do not need them, since we can always use pattern matching inside a `let` expression. In fact this is considered better programming style in SML.

What's next?

More SML constructs and general theory of functional programming.

 63 JACOBS UNIVERSITY

One construct that plays a central role in functional programming is the data type of lists. SML has a built-in type constructor for lists. We will use list functions to acquaint ourselves with the essential notion of recursion.

Using SML lists

▷ SML has a built-in "list type" (actually a list type constructor)

▷ given a type `ty`, `list ty` is also a type.

```
- [1,2,3];
val it = [1,2,3] : int list
```

▷ constructors `nil` and `::` (`nil` $\widehat{=}$ empty list, `::` $\widehat{=}$ list constructor "cons")

```
- nil;
val it = [] : 'a list
- 9::nil;
val it = [9] : int list
```

▷ A simple recursive function: creating integer intervals

```
- fun upto (m,n) = if m>n then nil else m::upto(m+1,n);
val upto = fn : int * int -> int list
- upto(2,5);
```

```
val it = [2,3,4,5] : int list
```

Question: What is happening here, we define a function by itself? (circular?)

A constructor is an operator that "constructs" members of an SML data type.

The type of lists has two constructors: `nil` that "constructs" a representation of the empty list, and the "list constructor" `::` (we pronounce this as "cons"), which constructs a new list `h::l` from a list `l` by pre-pending an element `h` (which becomes the new head of the list).

Note that the type of lists already displays the circular behavior we also observe in the function definition above: A list is either empty or the cons of a list. We say that the type of lists is inductive or inductively defined.

In fact, the phenomena of recursion and inductive types are inextricably linked, we will explore this in more detail below.

Defining Functions by Recursion

- ▷ SML allows to call a function already in the function definition.

```
fun upto (m,n) = if m>n then nil else m::upto(m+1,n);
```

- ▷ Evaluation in SML is "call-by-value" i.e. to whenever we encounter a function applied to arguments, we compute the value of the arguments first.
- ▷ So we have the following evaluation sequence:

  `upto(2,4)` $\rightsquigarrow$ `2::upto(3,4)` $\rightsquigarrow$ `2::(3::upto(4,4))` $\rightsquigarrow$ `2::(3::(4::nil))` $=$ `[2,3,4]`

- ▷ **Definition 92** We call an SML function recursive, iff the function is called in the function definition.
- ▷ Note that recursive functions need not terminate, consider the function

```
fun diverges (n) = n + diverges(n+1);
```

  which has the evaluation sequence

  `diverges(1)` $\rightsquigarrow$ `1 + diverges(2)` $\rightsquigarrow$ `1 + (2 + diverges(3))` $\rightsquigarrow \ldots$

Defining Functions by cases

- ▷ Idea: Use the fact that lists are either `nil` or of the form `X::Xs`, where `X` is an element and `Xs` is a list of elements.
- ▷ The body of an SML function can be made of several cases separated by the operator `|`.
- ▷ **Example 93** Flattening lists of lists (using the infix append operator `@`)

```
fun flat [] = [] (* base case *)
   | flat (l::ls) = l @ flat ls; (* step case *)
val flat = fn : 'a list list -> 'a list
- flat [["When","shall"],["we","three"],["meet","again"]]
["When","shall","we","three","meet","again"]
```

Defining functions by cases and recursion is a very important programming mechanism in SML. At the moment we have only seen it for the built-in type of lists. In the future we will see that it can also be used for user-defined data types. We start out with another one of SMLs basic types: strings.

We will now look at the the `string` type of SML and how to deal with it. But before we do, let us recap what strings are. Strings are just sequences of characters.

Therefore, SML just provides an interface to lists for manipulation.

Lists and Strings

- some programming languages provide a type for single characters (strings are lists of characters there)
- in SML, `string` is an atomic type
  - function `explode` converts from `string` to `char list`
  - function `implode` does the reverse

```
- explode "GenCS␣1";
val it = [#"G",#"e",#"n",#"C",#"S",#"␣",#"1"] : char list
- implode it;
val it = "GenCS␣1" : string
```

Exercise: Try to come up with a function that detects palindromes like 'otto' or 'anna', try also (more at [Pal])

- - 'Marge lets Norah see Sharon's telegram', or (up to case, punct and space)
  - 'Ein Neger mit Gazelle zagt im Regen nie' (for German speakers)

The next feature of SML is slightly disconcerting at first, but is an essential trait of functional programming languages: functions are first-class objects. We have already seen that they have types, now, we will see that they can also be passed around as argument and returned as values. For this, we will need a special syntax for functions, not only the `fun` keyword that declares functions.

Higher-Order Functions

- Idea: pass functions as arguments (functions are normal values.)
- **Example 94** Mapping a function over a list

```
- fun f x = x + 1;
- map f [1,2,3,4];
[2,3,4,5] : int list
```

- **Example 95** We can program the map function ourselves!

```
fun mymap (f, nil) = nil
  | mymap (f, h::t) = (f h) :: mymap (f,t);
```

- **Example 96** declaring functions (yes, functions are normal values.)

```
- val identity = fn x => x;
val identity = fn : 'a -> 'a
```

```
- identity(5);
val it = 5 : int
```

▷ **Example 97** returning functions: (again, functions are normal values.)

```
- val constantly = fn k => (fn a => k);
- (constantly 4) 5;
val it = 4 : int
- fun constantly k a = k;
```


One of the neat uses of higher-order function is that it is possible to re-interpret binary functions as unary ones using a technique called "Currying" after the Logician Haskell Brooks Curry ($*$(1900), †(1982)). Of course we can extend this to higher arities as well. So in theory we can consider n-ary functions as syntactic sugar for suitable higher-order functions.

Cartesian and Cascaded Procedures

▷ We have not been able to treat binary, ternary,… procedures directly

▷ Workaround 1: Make use of (Cartesian) products (unary functions on tuples)

▷ **Example 98** $+ \colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ with $+(\langle 3, 2 \rangle)$ instead of $+(3, 2)$

```
fun cartesian_plus (x:int,y:int) = x + y;
cartesian_plus : int * int -> int
```

Workaround 2: Make use of functions as results

▷ **Example 99** $+ \colon \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ with $+(3)(2)$ instead of $+(3, 2)$.

```
fun cascaded_plus (x:int) = (fn y:int => x + y);
cascaded_plus : int -> (int -> int)
```

Note: `cascaded_plus` can be applied to only one argument: `cascaded_plus 1` is the function `(fn y:int => 1 + y)`, which increments its argument.

SML allows both Cartesian- and cascaded functions, since we sometimes want functions to be flexible in function arities to enable reuse, but sometimes we want rigid arities for functions as this helps find programming errors.

Cartesian and Cascaded Procedures (Brackets)

▷ ▷ **Definition 100** Call a procedure Cartesian, iff the argument type is a product type, call it cascaded, iff the result type is a function type.

▷ **Example 101** the following function is both Cartesian and cascading

```
- fun both_plus (x:int,y:int) = fn (z:int) => x + y + z;
val both_plus (int * int) -> (int -> int)
```

Convenient: Bracket elision conventions

▷ ▷ $e_1\ e_2\ e_3 \rightsquigarrow (e_1\ e_2)\ e_3$[5] (procedure application associates to the left)

▷ $\tau_1 \to \tau_2 \to \tau_3 \rightsquigarrow \tau_1 \to (\tau_2 \to \tau_3)$ (function types associate to the right)

▷ SML uses these elision rules

```
- fun both_plus (x:int,y:int) = fn (z:int) => x + y + z;
val both_plus int * int -> int -> int
cascaded_plus 4 5;
```

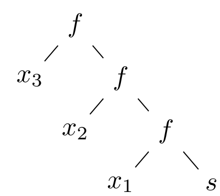
▷ Another simplification (related to those above)

```
- fun cascaded_plus x y = x + y;
val cascaded_plus : int -> int -> int
```


[e] EdNote: Generla Problem: how to mark up SML syntax?

Folding Procedures

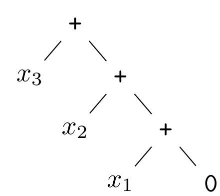
▷ **Definition 102** SML provides the left folding operator to realize a recurrent computation schema



foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl f s [x_1,x_2,x_3] = f(x_3,f(x_2,f(x_1,s)))

We call the procedure f the iterator and s the start value

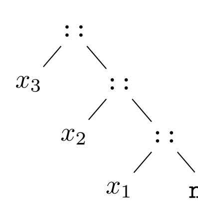
▷ **Example 103** Folding the iterator op+ with start value 0:



foldl op+ 0 [x_1,x_2,x_3] = x_3+(x_2+(x_1+0))

Thus the procedure `fun plus xs = foldl op+ 0 xs` adds the elements of integer lists.

Folding Procedures (continued)



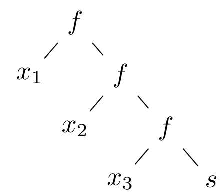
▷ **Example 104 (Reversing Lists)**

foldl op:: nil [x_1,x_2,x_3]
= x_3 :: (x_2 :: (x_1:: nil))

Thus the procedure `fun rev xs = foldl op:: nil xs` reverses a list

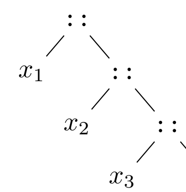
Folding Procedures (`foldr`)

▷ **Definition 105** The right folding operator `foldr` is a variant of `foldl` that processes the list elements in reverse order.




```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```
`foldr` f s $[x_1, x_2, x_3]$ = $f(x_1, f(x_2, f(x_3, s)))$

▷ **Example 106 (Appending Lists)**



`foldr op:: ys` $[x_1, x_2, x_3]$ = x_1 `::` $(x_2$ `::` $(x_3$ `:: ys))`

```
fun append(xs,ys) = foldr op:: ys xs
```


Now that we know some SML

SML is a "functional Programming Language"

What does this all have to do with functions?

Back to Induction, "Peano Axioms" and functions (to keep it simple)

2.3.2 Inductively Defined Sets and Computation

Let us now go back to looking at concrete functions on the unary natural numbers. We want to convince ourselves that addition is a (binary) function. Of course we will do this by constructing a proof that only uses the axioms pertinent to the unary natural numbers: the Peano Axioms.

But before we can prove function-hood of the addition function, we must solve a problem: addition is a binary function (intuitively), but we have only talked about unary functions. We could solve this problem by taking addition to be a cascaded function, but we will take the intuition seriously that it is a Cartesian function and make it a function from $\mathbb{N}_1 \times \mathbb{N}_1$ to $\mathbb{N}_1$.

What about Addition, is that a function?

▷ Problem: Addition takes two arguments (binary function)

▷ One solution: $+ : \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ is unary

▷ $+(\langle n, o\rangle) = n$ (base) and $+(\langle m, s(n)\rangle) = s(+(\langle m, n\rangle))$ (step)

▷ **Theorem 107** $+ \subseteq (\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1$ *is a total function.*

▷ We have to show that for all $\langle n, m\rangle \in (\mathbb{N}_1 \times \mathbb{N}_1)$ there is exactly one $l \in \mathbb{N}_1$ with $\langle\langle n, m\rangle, l\rangle \in +$.

▷ We will use functional notation for simplicity

Addition is a total Function

▷ **Lemma 108** *For all* $\langle n, m\rangle \in (\mathbb{N}_1 \times \mathbb{N}_1)$ *there is exactly one* $l \in \mathbb{N}_1$ *with* $+(\langle n, m\rangle) = l$.

▷ Proof: by induction on m. (what else)

P.1 we have two cases

P.1.1 base case ($m = o$):

P.1.1.1 choose $l := n$, so we have $+(\langle n, o\rangle) = n = l$.

P.1.1.2 For any $l' = +(\langle n, o\rangle)$, we have $l' = n = l$. □

P.1.2 step case ($m = s(k)$):

P.1.2.1 assume that there is a unique $r = +(\langle n, k\rangle)$, choose $l := s(r)$, so we have $+(\langle n, s(k)\rangle) = s(+(\langle n, k\rangle)) = s(r)$.

P.1.2.2 Again, for any $l' = +(\langle n, s(k)\rangle)$ we have $l' = l$. □

□

▷ **Corollary 109** $+ \colon \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ *is a total function.*

The main thing to note in the proof above is that we only needed the Peano Axioms to prove function-hood of addition. We used the induction axiom (P5) to be able to prove something about "all unary natural numbers". This axiom also gave us the two cases to look at. We have used the distinctness axioms (P3 and P4) to see that only one of the defining equations applies, which in the end guaranteed uniqueness of function values.

Reflection: How could we do this?

▷ we have two constructors for $\mathbb{N}_1$: the base element $o \in \mathbb{N}_1$ and the successor function $s \colon \mathbb{N}_1 \to \mathbb{N}_1$

▷ Observation: Defining Equations for $+$: $+(\langle n, o\rangle) = n$ (base) and $+(\langle m, s(n)\rangle) = s(+(\langle m, n\rangle))$ (step)

- ▷ the equations cover all cases: n is arbitrary, $m = o$ and $m = s(k)$ (otherwise we could have not proven existence)
- ▷ but not more (no contradictions)

▷ using the induction axiom in the proof of unique existence.

▷ **Example 110** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$

▷ **Example 111** Defining equations $\mu(l, o) = o$ and $\mu(l, s(r)) = +(\langle \mu(l, r), l\rangle)$

▷ Idea: Are there other sets and operations that we can do this way?

- ▷ the set should be built up by "injective" constructors and have an induction axiom ("abstract data type")
- ▷ the operations should be built up by case-complete equations

The specific characteristic of the situation is that we have an inductively defined set: the unary natural numbers, and defining equations that cover all cases (this is determined by the constructors) and that are non-contradictory. This seems to be the pre-requisites for the proof of functionality we have looked up above.

As we have identified the necessary conditions for proving function-hood, we can now generalize the situation, where we can obtain functions via defining equations: we need inductively defined sets, i.e. sets with Peano-like axioms.

Peano Axioms for Lists $\mathcal{L}[\mathbb{N}]$

▷ Lists of (unary) natural numbers: $[1,2,3]$, $[7,7]$, $[]$, ...

- ▷ nil-rule: start with the empty list $[]$
- ▷ cons-rule: extend the list by adding a number $n \in \mathbb{N}_1$ at the front

▷ two constructors: $\mathsf{nil} \in \mathcal{L}[\mathbb{N}]$ and $\mathsf{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$

▷ **Example 112** e.g. $[3,2,1] \hat{=} \mathsf{cons}(3,\mathsf{cons}(2,\mathsf{cons}(1,\mathsf{nil})))$ and $[] \hat{=} \mathsf{nil}$

▷ **Definition 113** We will call the following set of axioms are called the Peano Axioms for $\mathcal{L}[\mathbb{N}]$ in analogy to the Peano Axioms in Definition 18

▷ **Axiom 114 (LP1)** $\mathsf{nil} \in \mathcal{L}[\mathbb{N}]$ (generation axiom (nil))

▷ **Axiom 115 (LP2)** $\mathsf{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ (generation axiom (cons))

▷ **Axiom 116 (LP3)** nil is not a cons-value

▷ **Axiom 117 (LP4)** cons is injective

▷ **Axiom 118 (LP5)** If the nil possesses property P and (Induction Axiom)

- ▷ for any list l with property P, and for any $n \in \mathbb{N}_1$, the list $\mathsf{cons}(n,l)$ has property P

then every list $l \in \mathcal{L}[\mathbb{N}]$ has property P.

Note: There are actually 10 (Peano) axioms for lists of unary natural numbers the original five for $\mathbb{N}_1$ — they govern the constructors o and s, and the ones we have given for the constructors nil and cons here.

Note that the Pi and the **LPi** are very similar in structure: they say the same things about the constructors.

The first two axioms say that the set in question is generated by applications of the constructors: Any expression made of the constructors represents a member of $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$ respectively.

The next two axioms eliminate any way any such members can be equal. Intuitively they can only be equal, if they are represented by the same expression. Note that we do not need any axioms for the relation between $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$ constructors, since they are different as members of different sets.

Finally, the induction axioms give an upper bound on the size of the generated set. Intuitively the axiom says that any object that is not represented by a constructor expression is not a member of $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$.

Operations on Lists: Append

- ▷ The append function $@ \colon \mathcal{L}[\mathbb{N}] \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ concatenates lists
  Defining equations: $\mathrm{nil}@l = l$ and $\mathrm{cons}(n,l)@r = \mathrm{cons}(n, l@r)$
- ▷ **Example 119** $[3,2,1]@[1,2] = [3,2,1,1,2]$ and $[]@[1,2,3] = [1,2,3] = [1,2,3]@[]$
- ▷ **Lemma 120** *For all $l, r \in \mathcal{L}[\mathbb{N}]$, there is exactly one $s \in \mathcal{L}[\mathbb{N}]$ with $s = l@r$.*
- ▷ Proof: by induction on l. (what does this mean?)

  P.1 we have two cases

  P.1.1 base case: $l = \mathrm{nil}$:

  must have $s = r$.

  P.1.2 step case: $l = \mathrm{cons}(n,k)$ for some list k:

  P.1.2.1 Assume that here is a unique s' with $s' = k@r$,

  P.1.2.2 then $s = \mathrm{cons}(n,k)@r = \mathrm{cons}(n, k@r) = \mathrm{cons}(n, s')$. □

  □
- ▷ **Corollary 121** *Append is a function* *(see, this just worked fine!)*

 79 JACOBS UNIVERSITY

You should have noticed that this proof looks exactly like the one for addition. In fact, wherever we have used an axiom Pi there, we have used an axiom **LPi** here. It seems that we can do anything we could for unary natural numbers for lists now, in particular, programming by recursive equations.

Operations on Lists: more examples

- ▷ **Definition 122** $\lambda(\mathrm{nil}) = o$ and $\lambda(\mathrm{cons}(n,l)) = s(\lambda(l))$
- ▷ **Definition 123** $\rho(\mathrm{nil}) = \mathrm{nil}$ and $\rho(\mathrm{cons}(n,l)) = \rho(l)@\mathrm{cons}(n,\mathrm{nil})$.

 80 JACOBS UNIVERSITY

Now, we have seen that "inductively defined sets" are a basis for computation, we will turn to the programming language see them at work in concrete setting.

2.3.3 Inductively Defined Sets in SML

We are about to introduce one of the most powerful aspects of SML, its ability to define data types. After all, we have claimed that types in SML are first-class objects, so we have to have a means of constructing them.

We have seen above, that the main feature of an inductively defined set is that it has Peano Axioms that enable us to use it for computation. Note that specifying them, we only need to know the constructors (and their types). Therefore the `datatype` constructor in SML only needs to specify this information as well. Moreover, note that if we have a set of constructors of an inductively defined set — e.g. `zero : mynat` and `suc : mynat -> mynat` for the set `mynat`, then their codomain type is always the same: `mynat`. Therefore, we can condense the syntax even further by leaving that implicit.

Data Type Declarations

▷ concrete version of abstract data types in SML

```
- datatype mynat = zero | suc of mynat;
datatype mynat = suc of mynat | zero
```

▷ this gives us constructor functions `zero : mynat` and `suc : mynat -> mynat`.

▷ define functions by (complete) case analysis (abstract procedures)

```
fun num (zero) = 0 | num (suc(n)) = num(n) + 1;
val num = fn : mynat -> int
fun incomplete (zero) = 0;
stdIn:10.1-10.25 Warning: match nonexhaustive
      zero => ...
val incomplete = fn : mynat -> int

fun ic (zero) = 1 | ic(suc(n))=2 | ic(zero)= 3;
stdIn:1.1-2.12 Error: match redundant
     zero => ...
     suc n => ...
     zero => ...
```


So, we can re-define a type of unary natural numbers in SML, which may seem like a somewhat pointless exercise, since we have integers already. Let us see what else we can do.

Data Types Example (Enumeration Type)

▷ a type for weekdays (nullary constructors)

```
datatype day = mon | tue | wed | thu | fri | sat | sun;
```

▷ use as basis for rule-based procedure (first clause takes precedence)

```
- fun weekend sat = true
       | weekend sun = true
       | weekend _ = false
val weekend : day -> bool
```

▷ this give us

```
- weekend sun
true : bool
- map weekend [mon, wed, fri, sat, sun]
[false, false, false, true, true] : bool list
```

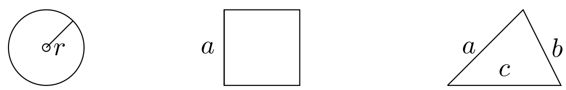
▷ nullary constructors describe values, enumeration types finite sets

Somewhat surprisingly, finite enumeration types that are a separate constructs in most programming languages are a special case of `datatype` declarations in SML. They are modeled by sets of base constructors, without any functional ones, so the base cases form the finite possibilities in this type. Note that if we imagine the Peano Axioms for this set, then they become very simple; in particular, the induction axiom does not have step cases, and just specifies that the property P has to hold on all base cases to hold for all members of the type.

Let us now come to a real-world examples for data types in SML. Say we want to supply a library for talking about mathematical shapes (circles, squares, and triangles for starters), then we can represent them as a data type, where the constructors conform to the three basic shapes they are in. So a circle of radius r would be represented as the constructor term `Circle $r$` (what else).

Data Types Example (Geometric Shapes)

▷ describe three kinds of geometrical forms as mathematical objects



Circle (r) Square (a) Triangle (a, b, c)

Mathematically: $\mathbb{R}^+ \uplus \mathbb{R}^+ \uplus ((\mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+))$

▷ In SML: approximate $\mathbb{R}^+$ by the built-in type `real`.

```
datatype shape =
    Circle of real
  | Square of real
  | Triangle of real * real * real
```

▷ This gives us the constructor functions

```
Circle : real -> shape
Square : real -> shape
Triangle : real * real * real -> shape
```


Some experiments:

```
- Circle 4.0
Circle 4.0 : shape
- Square 3.0
Square 3.0 : shape
- Triangle(4.0, 3.0, 5.0)
Triangle(4.0, 3.0, 5.0) : shape
```

Data Types Example (Areas of Shapes)

▷ a procedure that computes the area of a shape:

```
- fun area (Circle r) = Math.pi*r*r
    | area (Square a) = a*a
    | area (Triangle(a,b,c)) = let val s = (a+b+c)/2.0
                               in Math.sqrt(s*(s-a)*(s-b)*(s-c))
                               end
val area : shape -> real
```

New Construct: Standard structure `Math` (see [SML10])

▷ some experiments

```
- area (Square 3.0)
9.0 : real
- area (Triangle(6.0, 6.0, Math.sqrt 72.0))
18.0 : real
```

The beauty of the representation in user-defined types is that this affords powerful abstractions that allow to structure data (and consequently program functionality). All three kinds of shapes are included in one abstract entity: the type `shape`, which makes programs like the `area` function conceptually simple — it is just a function from type `shape` to type `real`. The complexity — after all, we are employing three different formulae for computing the area of the respective shapes — is hidden in the function body, but is nicely compartmentalized, since the constructor cases in systematically correspond to the three kinds of shapes.

We see that the combination of user-definable types given by constructors, pattern matching, and function definition by (constructor) cases give a very powerful structuring mechanism for heterogeneous data objects. This makes is easy to structure programs by the inherent qualities of the data. A trait that other programming languages seek to achieve by object-oriented techniques.

We will now develop a theory of the expressions we write down in functional programming languages and the way they are used for computation.

2.3.4 A Theory of SML: Abstract Data Types and Term Languages

What's next?

Let us now look at representations
and SML syntax
in the abstract!

 85 JACOBS UNIVERSITY

In this subsection, we will study computation in functional languages in the abstract by building mathematical models for them. We will proceed as we often do in science and modeling: we build a very simple model, and "test-drive" it to see whether it covers the phenomena we want to understand. Following this lead we will start out with a notion of "ground constructor terms" for the representation of data and with a simple notion of abstract procedures that allow computation by replacement of equals. We have chosen this first model intentionally naive, so that it fails to capture the essentials, so we get the chance to refine it to one based on "constructor terms with variables" and finally on "terms", refining the relevant concepts along the way.

This iterative approach intends to raise awareness that in CS theory it is not always the first model that eventually works, and at the same time intends to make the model easier to understand by repetition.

Abstract Data Types and Ground Constructor Terms

Abstract data types are abstract objects that specify inductively defined sets by declaring their constructors.

Abstract Data Types (ADT)

▷ **Definition 124** Let $\mathcal{S}^0 := \{\mathbb{A}_1, \ldots, \mathbb{A}_n\}$ be a finite set of symbols, then we call the set $\mathcal{S}$ the set of sorts over the set $\mathcal{S}^0$, if

- ▷ $\mathcal{S}^0 \subseteq \mathcal{S}$ (base sorts are sorts)
- ▷ If $\mathbb{A}, \mathbb{B} \in \mathcal{S}$, then $(\mathbb{A} \times \mathbb{B}) \in \mathcal{S}$ (product sorts are sorts)
- ▷ If $\mathbb{A}, \mathbb{B} \in \mathcal{S}$, then $(\mathbb{A} \to \mathbb{B}) \in \mathcal{S}$ (function sorts are sorts)

▷ **Definition 125** If c is a symbol and $\mathbb{A} \in \mathcal{S}$, then we call a pair $[c \colon \mathbb{A}]$ a constructor declaration for c over $\mathcal{S}$.

▷ **Definition 126** Let $\mathcal{S}^0$ be a set of symbols and Σ a set of constructor declarations over $\mathcal{S}$, then we call the pair $\langle \mathcal{S}^0, \Sigma \rangle$ an abstract data type

▷ **Example 127** $\langle\{\mathbb{N}\},\{[o\colon\mathbb{N}],[s\colon\mathbb{N}\to\mathbb{N}]\}\rangle$

▷ **Example 128** $\langle\{\mathbb{N},\mathcal{L}(\mathbb{N})\},\{[o\colon\mathbb{N}],[s\colon\mathbb{N}\to\mathbb{N}],[\mathsf{nil}\colon\mathcal{L}(\mathbb{N})],[\mathsf{cons}\colon\mathbb{N}\times\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})]\}\rangle$ In particular, the term $\mathsf{cons}(s(o),\mathsf{cons}(o,\mathsf{nil}))$ represents the list $[1,0]$

▷ **Example 129** $\langle\{\mathcal{S}\},\{[\iota\colon\mathcal{S}],[\to\colon\mathcal{S}\times\mathcal{S}\to\mathcal{S}],[\times\colon\mathcal{S}\times\mathcal{S}\to\mathcal{S}]\}\rangle$

In contrast to SML `datatype` declarations we allow more than one sort to be declared at one time. So abstract data types correspond to a group of `datatype` declarations.

With this definition, we now have a mathematical object for (sequences of) data type declarations in SML. This is not very useful in itself, but serves as a basis for studying what expressions we can write down at any given moment in SML. We will cast this in the notion of constructor terms that we will develop in stages next.

Ground Constructor Terms

▷ **Definition 130** Let $\mathcal{A}:=\langle\mathcal{S}^0,\mathcal{D}\rangle$ be an abstract data type, then we call a representation t a ground constructor term of sort $\mathbb{T}$, iff

- ▷ $\mathbb{T}\in\mathcal{S}^0$ and $[t\colon\mathbb{T}]\in\mathcal{D}$, or
- ▷ $\mathbb{T}=\mathbb{A}\times\mathbb{B}$ and t is of the form $\langle a,b\rangle$, where a and b are ground constructor terms of sorts $\mathbb{A}$ and $\mathbb{B}$, or
- ▷ t is of the form $c(a)$, where a is a ground constructor term of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon\mathbb{A}\to\mathbb{T}]\in\mathcal{D}$.

We denote the set of all ground constructor terms of sort $\mathbb{A}$ with $\mathcal{T}^g_{\mathbb{A}}(\mathcal{A})$ and use $\mathcal{T}^g(\mathcal{A}):=\bigcup_{\mathbb{A}\in\mathcal{S}}\mathcal{T}^g_{\mathbb{A}}(\mathcal{A})$.

▷ **Definition 131** If $t=c(t')$ then we say that the symbol c is the head of t (write $\mathbf{head}(t)$). If $t=a$, then $\mathbf{head}(t)=a$; $\mathbf{head}(\langle t_1,t_2\rangle)$ is undefined.

▷ **Notation 132** We will write $c(a,b)$ instead of $c(\langle a,b\rangle)$ (cf. binary function)

The main purpose of ground constructor terms will be to represent data. In the data type from Example 127 the ground constructor term $s(s(o))$ can be used to represent the unary natural number 2. Similarly, in the abstract data type from Example 128, the term $\mathrm{cons}(s(s(o)),\mathrm{cons}(s(o),\mathrm{nil}))$ represents the list $[2,1]$.

Note: that to be a good data representation format for a set S of objects, ground constructor terms need to

- cover S, i.e. that for every object $s\in S$ there should be a ground constructor term that represents s.
- be unambiguous, i.e. that we can decide equality by just looking at them, i.e. objects $s\in S$ and $t\in S$ are equal, iff their representations are.

But this is just what our Peano Axioms are for, so abstract data types come with specialized Peano axioms, which we can paraphrase as

Peano Axioms for Abstract Data Types

▷ Idea: Sorts represent sets!

▷ **Axiom 133** if t is a ground constructor term of sort $\mathbb{T}$, then $t\in\mathbb{T}$

▷ **Axiom 134** equality on ground constructor terms is trivial

▷ **Axiom 135** only ground constructor terms of sort $\mathbb{T}$ are in $\mathbb{T}$ (induction axioms)

Example 136 (An Abstract Data Type of Truth Values) We want to build an abstract data type for the set $\{T, F\}$ of truth values and various operations on it: We have looked at the abbreviations $\wedge$, $\vee$, $\neg$, $\Rightarrow$ for "and", "or", "not", and "implies". These can be interpreted as functions on truth values: e.g. $\neg(T) = F$, ... We choose the abstract data type $\langle\{\mathbb{B}\}, \{[T\colon \mathbb{B}], [F\colon \mathbb{B}]\}\rangle$, and have the abstract procedures

$$\wedge:\ \langle \wedge::\mathbb{B}\times\mathbb{B}\to\mathbb{B}\,;\,\{\wedge(T,T)\rightsquigarrow T, \wedge(T,F)\rightsquigarrow F, \wedge(F,T)\rightsquigarrow F, \wedge(F,F)\rightsquigarrow F\}\rangle.$$

$$\vee:\ \langle \vee::\mathbb{B}\times\mathbb{B}\to\mathbb{B}\,;\,\{\vee(T,T)\rightsquigarrow T, \vee(T,F)\rightsquigarrow T, \vee(F,T)\rightsquigarrow T, \vee(F,F)\rightsquigarrow F\}\rangle.$$

$$\neg:\ \langle \neg::\mathbb{B}\to\mathbb{B}\,;\,\{\neg(T)\rightsquigarrow F, \neg(F)\rightsquigarrow T\}\rangle,$$

Now that we have established how to represent data, we will develop a theory of programs, which will consist of directed equations in this case. We will do this as theories often are developed; we start off with a very first theory will not meet the expectations, but the test will reveal how we have to extend the theory. We will iterate this procedure of theorizing, testing, and theory adapting as often as is needed to arrive at a successful theory.

A First Abstract Interpreter

Let us now come up with a first formulation of an abstract interpreter, which we will refine later when we understand the issues involved. Since we do not yet, the notions will be a bit vague for the moment, but we will see how they work on the examples.

But how do we compute?

▷ Problem: We can define functions, but how do we compute them?

▷ Intuition: We direct the equations (l2r) and use them as rules.

▷ **Definition 137** Let $\mathcal{A}$ be an abstract data type and $s, t \in \mathcal{T}^g_{\mathbb{T}}(\mathcal{A})$ ground constructor terms over $\mathcal{A}$, then we call a pair $s \rightsquigarrow t$ a rule for f, if $\mathbf{head}(s) = f$.

▷ **Example 138** turn $\lambda(\mathrm{nil}) = o$ and $\lambda(\mathrm{cons}(n, l)) = s(\lambda(l))$ to $\lambda(\mathrm{nil}) \rightsquigarrow o$ and $\lambda(\mathrm{cons}(n, l)) \rightsquigarrow s(\lambda(l))$

▷ **Definition 139** Let $\mathcal{A} := \langle\mathcal{S}^0, \mathcal{D}\rangle$, then call a quadruple $\langle f::\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$ an abstract procedure, iff $\mathcal{R}$ is a set of rules for f. $\mathbb{A}$ is called the argument sort and $\mathbb{R}$ is called the result sort of $\langle f::\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$.

▷ **Definition 140** A computation of an abstract procedure p is a sequence of ground constructor terms $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots$ according to the rules of p. (whatever that means)

▷ **Definition 141** An abstract computation is a computation that we can perform in our heads. (no real world constraints like memory size, time limits)

▷ **Definition 142** An abstract interpreter is an imagined machine that performs (abstract) computations, given abstract procedures.

The central idea here is what we have seen above: we can define functions by equations. But of course when we want to use equations for programming, we will have to take some freedom of applying them, which was useful for proving properties of functions above. Therefore we restrict them to be applied in one direction only to make computation deterministic.

Let us now see how this works in an extended example; we use the abstract data type of lists from Example 128 (only that we abbreviate unary natural numbers).

Example: the functions ρ and @ on lists

- Consider the abstract procedures $\langle\rho::\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{\rho(\mathrm{cons}(n,l))\rightsquigarrow@(\rho(l),\mathrm{cons}(n,\mathrm{nil})),\rho(\mathrm{nil})\rightsquigarrow\mathrm{nil}\}\rangle$ and $\langle@::\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{@(\mathrm{cons}(n,l),r)\rightsquigarrow\mathrm{cons}(n,@(l,r)),@(\mathrm{nil},l)\rightsquigarrow l\}\rangle$
- Then we have the following abstract computation
  - $\rho(\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil})))\rightsquigarrow@(\rho(\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{cons}(n,l))\rightsquigarrow@(\rho(l),\mathrm{cons}(n,\mathrm{nil}))$ with $n=2$ and $l=\mathrm{cons}(1,\mathrm{nil})$)
  - $@(\rho(\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow@(@(\rho(\mathrm{nil}),\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{cons}(n,l))\rightsquigarrow@(\rho(l),\mathrm{cons}(n,\mathrm{nil}))$ with $n=1$ and $l=\mathrm{nil}$)
  - $@(@(\rho(\mathrm{nil}),\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow@(@(\mathrm{nil},\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{nil})\rightsquigarrow\mathrm{nil}$)
  - $@(@(\mathrm{nil},\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow@(\mathrm{cons}(1,\mathrm{nil}),\mathrm{cons}(2,\mathrm{nil}))$ ($@(\mathrm{nil},l)\rightsquigarrow l$ with $l=\mathrm{cons}(1,\mathrm{nil})$)
  - $@(\mathrm{cons}(1,\mathrm{nil}),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow\mathrm{cons}(1,@(\mathrm{nil},\mathrm{cons}(2,\mathrm{nil})))$ ($@(\mathrm{cons}(n,l),r)\rightsquigarrow\mathrm{cons}(n,@(l,r))$ with $n=1$, $l=\mathrm{nil}$, and $r=\mathrm{cons}(2,\mathrm{nil})$)
  - $\mathrm{cons}(1,@(\mathrm{nil},\mathrm{cons}(2,\mathrm{nil})))\rightsquigarrow\mathrm{cons}(1,\mathrm{cons}(2,\mathrm{nil}))$ ($@(\mathrm{nil},l)\rightsquigarrow l$ with $l=\mathrm{cons}(2,\mathrm{nil})$)
  - Aha: ρ terminates on the argument $\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil}))$

Now let's get back to theory: let us see whether we can write down an abstract interpreter for this.

An Abstract Interpreter (preliminary version)

- **Definition 143 (Idea)** Replace equals by equals! (this is licensed by the rules)
  - Input: an abstract procedure $\langle f::\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$ and an argument $a\in\mathcal{T}^g_{\mathbb{A}}(\mathcal{A})$.
  - Output: a result $r\in\mathcal{T}^g_{\mathbb{R}}(\mathcal{A})$.
  - Process:
    - find a part $t:=f(t_1,\ldots t_n)$ in a,
    - find a rule $(l\rightsquigarrow r)\in\mathcal{R}$ and values for the variables in l that make t and l equal.
    - replace t with r' in a, where r' is obtained from r by replacing variables by values.
    - if that is possible call the result a' and repeat the process with a', otherwise stop.
- **Definition 144** We say that an abstract procedure $\langle f::\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$ terminates (on $a\in\mathcal{T}^g_{\mathbb{A}}(\mathcal{A})$), iff the computation (starting with $f(a)$) reaches a state, where no rule applies.
- There are a lot of words here that we do not understand
- let us try to understand them better $\rightsquigarrow$ more theory!

Unfortunately we do not have the means to write down rules: they contain variables, which are not allowed in ground constructor rules. So what do we do in this situation, we just extend the definition of the expressions we are allowed to write down.

Constructor Terms with Variables

- Wait a minute!: what are these rules in abstract procedures?
- Answer: pairs of constructor terms (really constructor terms?)
- Idea: variables stand for arbitrary constructor terms (let's make this formal)
- **Definition 145** Let $\langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type. A (constructor term) variable is a pair of a symbol and a base sort. E.g. $x_{\mathbb{A}}$, $n_{\mathbb{N}_1}$, $x_{\mathbb{C}^3}$,... .
- **Definition 146** We denote the current set of variables of sort $\mathbb{A}$ with $\mathcal{V}_{\mathbb{A}}$, and use $\mathcal{V} := \bigcup_{\mathbb{A}\in\mathcal{S}^0} \mathcal{V}_{\mathbb{A}}$ for the set of all variables.
- Idea: add the following rule to the definition of constructor terms
  - variables of sort $\mathbb{A} \in \mathcal{S}^0$ are constructor terms of sort $\mathbb{A}$.
- **Definition 147** If t is a constructor term, then we denote the set of variables occurring in t with $\mathbf{free}(t)$. If $\mathbf{free}(t) = \emptyset$, then we say t is ground or closed.

 92 JACOBS UNIVERSITY

To have everything at hand, we put the whole definition onto one slide.

Constr. Terms with Variables: The Complete Definition

- **Definition 148** Let $\langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type and $\mathcal{V}$ a set of variables, then we call a representation t a constructor term (with variables from $\mathcal{V}$) of sort $\mathbb{T}$, iff
  - $\mathbb{T} \in \mathcal{S}^0$ and $[t\colon \mathbb{T}] \in \mathcal{D}$, or
  - $t \in \mathcal{V}_{\mathbb{T}}$ is a variable of sort $\mathbb{T} \in \mathcal{S}^0$, or
  - $\mathbb{T} = \mathbb{A} \times \mathbb{B}$ and t is of the form $\langle a, b\rangle$, where a and b are constructor terms with variables of sorts $\mathbb{A}$ and $\mathbb{B}$, or
  - t is of the form $c(a)$, where a is a constructor term with variables of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$.

  We denote the set of all constructor terms of sort $\mathbb{A}$ with $\mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ and use $\mathcal{T}(\mathcal{A}; \mathcal{V}) := \bigcup_{\mathbb{A}\in\mathcal{S}} \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$.

 93 JACOBS UNIVERSITY

Now that we have extended our model of terms with variables, we will need to understand how to use them in computation. The main intuition is that variables stand for arbitrary terms (of the right sort). This intuition is modeled by the action of instantiating variables with terms, which in turn is the operation of applying a "substitution" to a term.

Substitutions

Substitutions are very important objects for modeling the operational meaning of variables: applying a substitution to a term instantiates all the variables with terms in it. Since a substitution only acts on the variables, we simplify its representation, we can view it as a mapping from variables to terms that can be extended to a mapping from terms to terms. The natural way to define substitutions would be to make them partial functions from variables to terms, but the definition below generalizes better to later uses of substitutions, so we present the real thing.

Substitutions

▷ **Definition 149** Let $\mathcal{A}$ be an abstract data type and $\sigma \in \mathcal{V} \to \mathcal{T}(\mathcal{A};\mathcal{V})$, then we call σ a substitution on $\mathcal{A}$, iff $\mathbf{supp}(\sigma) := \{x_{\mathbb{A}} \in \mathcal{V}_{\mathbb{A}} \mid \sigma(x_{\mathbb{A}}) \neq x_{\mathbb{A}}\}$ is finite and $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$. $\mathbf{supp}(\sigma)$ is called the support of σ.

▷ **Notation 150** We denote the substitution σ with $\mathbf{supp}(\sigma) = \{x^i_{\mathbb{A}_i} \mid 1 \leq i \leq n\}$ and $\sigma(x^i_{\mathbb{A}_i}) = t_i$ by $[t_1/x^1_{\mathbb{A}_1}], \ldots, [t_n/x^n_{\mathbb{A}_n}]$.

▷ **Definition 151 (Substitution Application)** Let $\mathcal{A}$ be an abstract data type, σ a substitution on $\mathcal{A}$, and $t \in \mathcal{T}(\mathcal{A};\mathcal{V})$, then then we denote the result of systematically replacing all variables $x_{\mathbb{A}}$ in t by $\sigma(x_{\mathbb{A}})$ by $\sigma(t)$. We call $\sigma(t)$ the application of σ to t.

▷ With this definition we extend a substitution σ from a function $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{A};\mathcal{V})$ to a function $\sigma\colon \mathcal{T}(\mathcal{A};\mathcal{V}) \to \mathcal{T}(\mathcal{A};\mathcal{V})$.

▷ **Definition 152** Let s and t be constructor terms, then we say that s matches t, iff there is a substitution σ, such that $\sigma(s) = t$. σ is called a matcher that instantiates s to t.

▷ **Example 153** $[a/x], [(f(b))/y], [a/z]$ instantiates $g(x, y, h(z))$ to $g(a, f(b), h(a))$. (sorts irrelevant here)

Note that we we have defined constructor terms inductively, we can write down substitution application as a recursive function over the inductively defined set.

Substitution Application (The Recursive Definition)

▷ We give the defining equations for substitution application

- ▹ $[t/x_{\mathbb{A}}](x) = t$
- ▹ $[t/x_{\mathbb{A}}](y) = y$ if $x \neq y$.
- ▹ $[t/x_{\mathbb{A}}](\langle a, b\rangle) = \langle [t/x_{\mathbb{A}}](a), [t/x_{\mathbb{A}}](b)\rangle$
- ▹ $[t/x_{\mathbb{A}}](f(a)) = f([t/x_{\mathbb{A}}](a))$

▷ this definition uses the inductive structure of the terms.

▷ **Definition 154 (Substitution Extension)** Let σ be a substitution, then we denote with $\sigma, [t/x_{\mathbb{A}}]$ the function $\{\langle y_{\mathbb{B}}, t\rangle \in \sigma \mid y_{\mathbb{B}} \neq x_{\mathbb{A}}\} \cup \{\langle x_{\mathbb{A}}, t\rangle\}$. ($\sigma, [t/x_{\mathbb{A}}]$ coincides with σ off $x_{\mathbb{A}}$, and gives the result t there.)

▷ Note: If σ is a substitution, then $\sigma, [t/x_{\mathbb{A}}]$ is also a substitution.

The extension of a substitution is an important operation, which you will run into from time to time. The intuition is that the values right of the comma overwrite the pairs in the substitution on the left, which already has a value for $x_{\mathbb{A}}$, even though the representation of σ may not show it.

Note that the use of the comma notation for substitutions defined in Notation 150 is consistent with substitution extension. We can view a substitution $[a/x], [(f(b))/y]$ as the extension of the empty substitution (the identity function on variables) by $[f(b)/y]$ and then by $[a/x]$. Note furthermore, that substitution extension is not commutative in general.

Now that we understand variable instantiation, we can see what it gives us for the meaning of rules: we get all the ground constructor terms a constructor term with variables stands for by applying all possible substitutions to it. Thus rules represent ground constructor subterm replacement actions in a computations, where we are allowed to replace all ground instances of the left hand side of the rule by the corresponding ground instance of the right hand side.

A Second Abstract Interpreter

Unfortunately, constructor terms are still not enough to write down rules, as rules also contain the symbols from the abstract procedures.

Are Constructor Terms Really Enough for Rules?

- ▷ **Example 155** $\rho(\mathsf{cons}(n,l)) \rightsquigarrow @(\rho(l), \mathsf{cons}(n, \mathsf{nil}))$. ($\rho$ is not a constructor)
- ▷ Idea: need to include defined procedures.
- ▷ **Definition 156** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type with $\mathbb{A} \in \mathcal{S}$, $f \notin \mathcal{D}$ be a symbol, then we call a pair $[f\colon \mathbb{A}]$ a procedure declaration for f over $\mathcal{S}$.

  We call a finite set Σ of procedure declarations a signature over $\mathcal{A}$, if Σ is a partial function. (unique sorts)
- ▷ add the following rules to the definition of constructor terms
  - ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[p\colon \mathbb{T}] \in \Sigma$, or
  - ▷ t is of the form $f(a)$, where a is a term of sort $\mathbb{A}$ and there is a procedure declaration $[f\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.
- ▷ we call the the resulting structures simply "terms" over $\mathcal{A}$, Σ, and $\mathcal{V}$ (the set of variables we use). We denote the set of terms of sort $\mathbb{A}$ with $\mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$.

Again, we combine all of the rules for the inductive construction of the set of terms in one slide for convenience.

Terms: The Complete Definition

- ▷ Idea: treat procedures (from Σ) and constructors (from $\mathcal{D}$) at the same time.
- ▷ **Definition 157** Let $\langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type, and Σ a signature over $\mathcal{A}$, then we call a representation t a term of sort $\mathbb{T}$ (over $\mathcal{A}$ and Σ), iff
  - ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[t\colon \mathbb{T}] \in \mathcal{D}$ or $[t\colon \mathbb{T}] \in \Sigma$, or
  - ▷ $t \in \mathcal{V}_{\mathbb{T}}$ and $\mathbb{T} \in \mathcal{S}^0$, or
  - ▷ $\mathbb{T} = \mathbb{A} \times \mathbb{B}$ and t is of the form $\langle a, b\rangle$, where a and b are terms of sorts $\mathbb{A}$ and $\mathbb{B}$, or
  - ▷ t is of the form $c(a)$, where a is a term of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$ or a procedure declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.

Subterms

▷ Idea: Well-formed parts of constructor terms are constructor terms again (maybe of a different sort)

▷ **Definition 158** Let $\mathcal{A}$ be an abstract data type and s and b be terms over $\mathcal{A}$, then we say that s is an immediate subterm of t, iff $t = f(s)$ or $t = \langle s, b\rangle$ or $t = \langle b, s\rangle$.

▷ **Definition 159** We say that a s is a subterm of t, iff $s = t$ or there is an immediate subterm t' of t, such that s is a subterm of t'.

▷ **Example 160** $f(a)$ is a subterm of the terms $f(a)$ and $h(g(f(a), f(b)))$, and an immediate subterm of $h(f(a))$.

 98 JACOBS UNIVERSITY

We have to strengthen the restrictions on what we allow as rules, so that matching of rule heads becomes unique (remember that we want to take the choice out of interpretation).

Furthermore, we have to get a grip on the signatures involved with programming. The intuition here is that each abstract procedure introduces a new procedure declaration, which can be used in subsequent abstract procedures. We formalize this notion with the concept of an abstract program, i.e. a *sequence* of abstract procedures over the underlying abstract data type that behave well with respect to the induced signatures.

Abstract Programs

▷ **Definition 161 (Abstract Procedures (final version))** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type, Σ a signature over $\mathcal{A}$, and $f \notin (\mathbf{dom}(\mathcal{D}) \cup \mathbf{dom}(\Sigma))$ a symbol, then we call $l \rightsquigarrow r$ a rule for $[f\colon \mathbb{A} \to \mathbb{B}]$ over Σ, if $l = f(s)$ for some $s \in \mathcal{T}_{\mathbb{A}}(\mathcal{D}; \mathcal{V})$ that has no duplicate variables and $r \in \mathcal{T}_{\mathbb{B}}(\mathcal{D}, \Sigma; \mathcal{V})$.

We call a quadruple $\mathcal{P} := \langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ an abstract procedure over Σ, iff $\mathcal{R}$ is a set of rules for $[f\colon \mathbb{A} \to \mathbb{R}] \in \Sigma$. We say that $\mathcal{P}$ induces the procedure declaration $[f\colon \mathbb{A} \to \mathbb{R}]$.

▷ **Definition 162 (Abstract Programs)** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type, and $\mathcal{P} := \mathcal{P}_1, \ldots, \mathcal{P}_n$ a sequence of abstract procedures, then we call $\mathcal{P}$ an abstract Program with signature Σ over $\mathcal{A}$, if the $\mathcal{P}_i$ induce (the procedure declarations) in Σ and

- ▹ $n = 0$ and $\Sigma = \emptyset$ or
- ▹ $\mathcal{P} = \mathcal{P}', \mathcal{P}_n$ and $\Sigma = \Sigma', [f\colon \mathbb{A}]$, where
  - ▹ $\mathcal{P}'$ is an abstract program over Σ'
  - ▹ and $\mathcal{P}_n$ is an abstract procedure over Σ' that induces the procedure declaration $[f\colon \mathbb{A}]$.

 99 JACOBS UNIVERSITY

Now, we have all the prerequisites for the full definition of an abstract interpreter.

An Abstract Interpreter (second version)

▷ **Definition 163 (Abstract Interpreter (second try))** Let $a_0 := a$ repeat the following as long as possible:

- ▹ choose $(l \rightsquigarrow r) \in \mathcal{R}$, a subterm s of a_i and matcher σ, such that $\sigma(l) = s$.
- ▹ let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

▷ **Definition 164** We say that an abstract procedure $\mathcal{P} := \langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ terminates (on $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$), iff the computation (starting with a) reaches a state, where no rule applies. Then a_n is the result of $\mathcal{P}$ on a

Question: Do abstract procedures always terminate?

⊳ Question: Is the result a_n always a constructor term?

Evaluation Order and Termination

To answer the questions remaining from the second abstract interpreter we will first have to think some more about the choice in this abstract interpreter: a fact we will use, but not prove here is we can make matchers unique once a subterm is chosen. Therefore the choice of subterm is all that we need wo worry about. And indeed the choice of subterm does matter as we will see.

Evaluation Order in SML

- Remember in the definition of our abstract interpreter:
  - choose a subterm s of a_i, a rule $(l \rightsquigarrow r) \in \mathcal{R}$, and a matcher σ, such that $\sigma(l) = s$.
  - let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

  Once we have chosen s, the choice of rule and matcher become unique (under reasonable side-conditions we cannot express yet)

- **Example 165** sometimes there we can choose more than one s and rule.

```
fun problem n = problem(n)+2;
datatype mybool = true | false;
fun myif(true,a,_) = a | myif(false,_,b) = b;
myif(true,3,problem(1));
```

- SML is a call-by-value language (values of arguments are computed first)

As we have seen in the example, we have to make up a policy for choosing subterms in evaluation to fully specify the behavior of our abstract interpreter. We will make the choice that corresponds to the one made in SML, since it was our initial goal to model this language.

An abstract call-by-value Interpreter

- **Definition 166 (Call-by-Value Interpreter (final))** We can now define a abstract call-by-value interpreter by the following process:
  - Let s be the leftmost (of the) minimal subterms s of a_i, such that there is a rule $l \rightsquigarrow r \in \mathcal{R}$ and a substitution σ, such that $\sigma(l) = s$.
  - let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

  Note: By this paragraph, this is a deterministic process, which can be implemented, once we understand matching fully (not covered in GenCS)

The name "call-by-value" comes from the fact that data representations as ground constructor terms are sometimes also called "values" and the act of computing a result for an (abstract) procedure applied to a bunch of argument is sometimes referred to as "calling an (abstract) procedure". So we can understand the "call-by-value" policy as restricting computation to the case where all of the arguments are already values (i.e. fully computed to ground terms).

Other programming languages chose another evaluation policy called "call-by-reference", which can be characterized by always choosing the outermost subterm that matches a rule. The most notable one is the Haskell language [Hut07, OSG08]. These programming languages are sometimes "lazy languages", since they are uniquely suited for dealing with objects that are potentially infinite in some form. In our example above, we can see the function `problem` as something that computes positive infinity. A lazy programming language would not be bothered by this and return the value `3`.

▷ **Example 167** A lazy language language can even quite comfortably compute with possibly infinite objects, lazily driving the computation forward as far as needed. Consider for instance the following program:

```
myif(problem(1) > 999,"yes","no");
```

In a "call-by-reference" policy we would try to compute the outermost subterm (the whole expression in this case) by matching the `myif` rules. But they only match if there is a `true` or `false` as the first argument, which is not the case. The same is true with the rules for `>`, which we assume to deal lazily with arithmetical simplification, so that it can find out that $x + 1000 > 999$. So the outermost subterm that matches is `problem(1)`, which we can evaluate 500 times to obtain `true`. Then and only then, the outermost subterm that matches a rule becomes the `myif` subterm and we can evaluate the whole expression to `true`.

Let us now turn to the question of termination of abstract procedures in general. Termination is a very difficult problem as Example 168 shows. In fact all cases that have been tried $\tau(n)$ diverges into the sequence $4, 2, 1, 4, 2, 1, \ldots$, and even though there is a huge literature in mathematics about this problem, a proof that τ diverges on all arguments is still missing.

Another clue to the difficulty of the termination problem is (as we will see) that there cannot be a a program that reliably tells of any program whether it will terminate.

But even though the problem is difficult in full generality, we can indeed make some progress on this. The main idea is to concentrate on the recursive calls in abstract procedures, i.e. the arguments of the defined function in the right hand side of rules. We will see that the recursion relation tells us a lot about the abstract procedure.

Analyzing Termination of Abstract Procedures

▷ **Example 168** $\tau\colon \mathbb{N}_1 \to \mathbb{N}_1$, where $\tau(n) \rightsquigarrow 3\tau(n) + 1$ for n odd and $\tau(n) \rightsquigarrow \tau(n)/2$ for n even. (does this procedure terminate?)

▷ **Definition 169** Let $\langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ be an abstract procedure, then we call a pair $\langle a, b\rangle$ a recursion step, iff there is a rule $f(x) \rightsquigarrow y$, and a substitution ρ, such that $\rho(x) = a$ and $\rho(y)$ contains a subterm $f(b)$.

▷ **Example 170** $\langle 4, 3\rangle$ is a recursion step for $\sigma\colon \mathbb{N}_1 \to \mathbb{N}_1$ with $\sigma(o) \rightsquigarrow o$ and $\sigma(s(n)) \rightsquigarrow n + \sigma(n)$

▷ **Definition 171** We call an abstract procedure $\mathcal{P}$ recursive, iff it has a recursion step. We call the set of recursion steps of $\mathcal{P}$ the recursion relation of $\mathcal{P}$.

▷ Idea: analyze the recursion relation for termination.

Now, we will define termination for arbitrary relations and present a theorem (which we do not really have the means to prove in GenCS) that tells us that we can reason about termination of abstract procedures — complex mathematical objects at best — by reasoning about the termination of their recursion relations — simple mathematical objects.

Termination

- **Definition 172** Let $R \subseteq \mathbb{A}^2$ be a binary relation, an infinite chain in R is a sequence $a_1, a_2, \ldots$ in $\mathbb{A}$, such that $\forall n \in \mathbb{N}_1.\langle a_n, a_{n+1}\rangle \in R$.

  We say that R terminates (on $a \in \mathbb{A}$), iff there is no infinite chain in R (that begins with a). We say that $\mathcal{P}$ diverges (on $a \in \mathbb{A}$), iff it does not terminate on a.

- **Theorem 173** *Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ be an abstract procedure and $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$, then $\mathcal{P}$ terminates on a, iff the recursion relation of $\mathcal{P}$ does.*

- **Definition 174** Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ be an abstract procedure, then we call the function $\{\langle a, b\rangle \mid a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$ and $\mathcal{P}$ terminates for a with $b\}$ in $\mathbb{A} \rightharpoonup \mathbb{B}$ the result function of $\mathcal{P}$.

- **Theorem 175** *Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{B}\,;\,\mathcal{D}\rangle$ be a terminating abstract procedure, then its result function satisfies the equations in $\mathcal{D}$.*

We should read Theorem 175 as the final clue that abstract procedures really do encode functions (under reasonable conditions like termination). This legitimizes the whole theory we have developed in this section.

Abstract vs. Concrete Procedures vs. Functions

- An abstract procedure $\mathcal{P}$ can be realized as concrete procedure $\mathcal{P}'$ in a programming language
- Correctness assumptions (this is the best we can hope for)
  - If the $\mathcal{P}'$ terminates on a, then the $\mathcal{P}$ terminates and yields the same result on a.
  - If the $\mathcal{P}$ diverges, then the $\mathcal{P}'$ diverges or is aborted (e.g. memory exhaustion or buffer overflow)
- Procedures are not mathematical functions (differing identity conditions)
  - compare $\sigma\colon \mathbb{N}_1 \to \mathbb{N}_1$ with $\sigma(o) \rightsquigarrow o$, $\sigma(s(n)) \rightsquigarrow n + \sigma(n)$ with $\sigma'\colon \mathbb{N}_1 \to \mathbb{N}_1$ with $\sigma'(o) \rightsquigarrow 0$, $\sigma'(s(n)) \rightsquigarrow ns(n)/2$
  - these have the same result function, but σ is recursive while σ' is not!
  - Two functions are equal, iff they are equal as sets, iff they give the same results on all arguments

2.3.5 More SML: Recursion in the Real World

We will now look at some concrete SML functions in more detail. The problem we will consider is that of computing the n^{th} Fibonacci number. In the famous Fibonacci sequence, the n^{th} element is obtained by adding the two immediately preceding ones.

This makes the function extremely simple and straightforward to write down in SML. If we look at the recursion relation of this procedure, then we see that it can be visualized a tree, as each natural number has two successors (as the the function `fib` has two recursive calls in the step case).

Consider the Fibonacci numbers

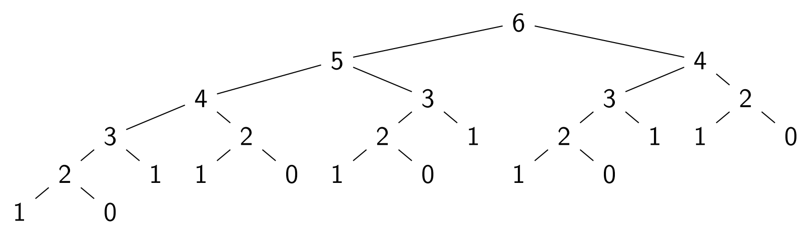
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

- ▷ generally: $f_{n+1} := f_n + f_{n-1}$ plus start conditions

▷ easy to program in SML:

```
fun fib (0) = 0 |fib (1) = 1 | fib (n:int) = fib (n-1) + fib(n-2);
```

▷ Let us look at the recursion relation: $\{\langle n, n-1\rangle, \langle n, n-2\rangle \mid n \in \mathbb{N}\}$ (it is a tree!)



Another thing we see by looking at the recursion relation is that the value `fib(k)` is computed $n-k+1$ times while computing `fib(k)`. All in all the number of recursive calls will be exponential in n, in other words, we can only compute a very limited initial portion of the Fibonacci sequence (the first 41 numbers) before we run out of time.

The main problem in this is that we need to know the last *two* Fibonacci numbers to compute the next one. Since we cannot "remember" any values in functional programming we take advantage of the fact that functions can return pairs of numbers as values: We define an auxiliary function `fob` (for lack of a better name) does all the work (recursively), and define the function `fib(n)` as the first element of the pair `fob(n)`.

The function `fob(n)` itself is a simple recursive procedure with one! recursive call that returns the last two values. Therefore, we use a `let` expression, where we place the recursive call in the declaration part, so that we can bind the local variables `a` and `b` to the last two Fibonacci numbers. That makes the return value very simple, it is the pair `(b,a+b)`.

A better Fibonacci Function

▷ Idea: Do not re-compute the values again and again!

- ▷ keep them around so that we can re-use them. (e.g. let `fib` compute the two last two numbers)

```
fun fob 0 = (0,1)
  | fob 1 = (1,1)
  | fob (n:int) =
    let
       val (a:int, b:int) = fob(n-1)
    in
        (b,a+b)
    end;
fun fib (n) = let val (b:int,_) = fob(n) in b end;
```

▷ Works in linear time! (unfortunately, we cannot see it, because SML Int are too small)

If we run this function, we see that it is indeed much faster than the last implementation. Unfortunately, we can still only compute the first 44 Fibonacci numbers, as they grow too fast, and we reach the maximal integer in SML.

Fortunately, we are not stuck with the built-in integers in SML; we can make use of more sophisticated implementations of integers. In this particular example, we will use the module

IntInf (infinite precision integers) from the SML standard library (a library of modules that comes with the SML distributions). The IntInf module provides a type IntINF.int and a set of infinite precision integer functions.

A better, larger Fibonacci Function

▷ Idea: Use a type with more Integers (Fortunately, there is IntInf)

```
use "/usr/share/smlnj/src/smlnj-lib/Util/int-inf.sml";

val zero = IntInf.fromInt 0;
val one = IntInf.fromInt 1;

fun bigfob (0) = (zero,one)
  | bigfob (1) = (one,one)
  | bigfob (n:int) = let val (a, b) = bigfob(n-1) in (b,IntInf.+(a,b)) end;

fun bigfib (n) = let val (a, _) = bigfob(n) in IntInf.toString(a) end;
```

   JACOBS UNIVERSITY

We have seen that functions are just objects as any others in SML, only that they have functional type. If we add the ability to have more than one declaration at at time, we can combine function declarations for mutually recursive function definitions. In a mutually recursive definition we define n functions *at the same time*; as an effect we can use all of these functions in recursive calls. In our example below, we will define the predicates even and odd in a mutual recursion.

Mutual Recursion

▷ generally, we can make more than one declaration at one time, e.g.

```
- val pi = 3.14 and e = 2.71;
val pi = 3.14
val e = 2.71
```

▷ this is useful mainly for function declarations, consider for instance:

```
fun even (zero) = true
  | even (suc(n)) = odd (n)
and odd (zero) = false
  | odd(suc(n)) = even (n)
```

trace: even(4), odd(3), even(2), odd(1), even(0), true.

   JACOBS UNIVERSITY

This mutually recursive definition is somewhat like the children's riddle, where we define the "left hand" as that hand where the thumb is on the right side and the "right hand" as that where the thumb is on the right hand. This is also a perfectly good mutual recursion, only — in contrast to the even/odd example above — the base cases are missing.

2.3.6 Even more SML: Exceptions and State in SML

Programming with Effects

▷ ▷ Until now, our procedures have been characterized entirely by their values on their arguments (as a mathematical function behaves)

▷ This is not enough, therefore SML also considers effects, e.g. for

- *input/output*: the interesting bit about a `print` statement is the effect
- *mutation*: allocation and modification of storage during evaluation
- *communication*: data may be sent and received over channels
- *exceptions*: abort evaluation by signaling an exceptional condition

Idea: An effect is any action resulting from an evaluation that is not returning a value (formal definition difficult)

- Documentation: should always address arguments, values, and effects!

Raising Exceptions

- Idea: Exceptions are generalized error codes
- **Example 176** predefined exceptions (exceptions have names)

```
- 3 div 0;
uncaught exception divide by zero
raised at: <file stdIn>
- fib(100);
uncaught exception overflow
raised at: <file stdIn>
```

- **Example 177** user-defined exceptions (exceptions are first-class objects)

```
- exception Empty;
exception Empty
- Empty;
val it = Empty : exn
```

- **Example 178** exception constructors (exceptions are just like any other value)

```
- exception SysError of int;
exception SysError of int;
- SysError
val it = fn : int -> exn
```


Programming with Exceptions

- **Example 179** A factorial function that checks for non-negative arguments(just to be safe)

```
exception Factorial;
- fun safe_factorial n =
      if n < 0 then raise Factorial
      else if n = 0 then 1
      else n * safe_factorial (n-1)
val safe_factorial = fn : int -> int
- safe_factorial(~1);
uncaught exception Factorial
raised at: stdIn:28.31-28.40
```

unfortunately, this program checks the argument in every recursive call

Programming with Exceptions (next attempt)

▷ Idea: make use of local function definitions that do the real work

```
- local
      fun fact 0 = 1 | fact n = n * fact (n-1)
  in
      fun safe_factorial n =
       if n >= 0 then fact n else raise Factorial
  end
val safe_factorial = fn : int -> int
- safe_factorial(~1);
uncaught exception Factorial
raised at: stdIn:28.31-28.40
```

this function only checks once, and the local function makes good use of pattern matching ($\leadsto$ standard programming pattern)

 113 JACOBS UNIVERSITY

Handling Exceptions

▷ **Definition 180 (Idea)** Exceptions can be raised (through the evaluation pattern) and handled somewhere above (throw and catch)

▷ Consequence: Exceptions are a general mechanism for non-local transfers of control.

▷ **Definition 181 (SML Construct)** exception handler: `exp handle rules`

▷ **Example 182** Handling the `Factorial` expression

```
fun factorial_driver () =
    let val input = read_integer ()
        val result = toString (safe_factorial input)
    in
        print result
    end
    handle Factorial => print "Out␣of␣range."
         | NaN => print "Not␣a␣Number!"
```

▷ For more information on SML: RTFM (read the fine manuals)

 114 JACOBS UNIVERSITY

Input and Output in SML

▷ Input and Output is handled via "streams" (think of infinite strings)

▷ there are two predefined streams `TextIO.stdIn` and `TextIO.stdOut` ($\hat{=}$ keyboard input and screen)

▷ Input: via {`TextIO.inputLine : TextIO.instream -> string`

```
- TextIO.inputLine(TextIO.stdIn);
  sdflkjsdlfkj
val it = "sdflkjsdlfkj" : string
```

▷ **Example 183** the `read_integer` function (just to be complete)

```
exception NaN; (* Not a Number *)
fun read_integer () =
    let
        val in = TextIO.inputLine(TextIO.stdIn);
    in
        if is_integer(in) then to_int(in) else raise NaN
    end;
```


2.4 Encoding Programs as Strings

With the abstract data types we looked at last, we studied term structures, i.e. complex mathematical objects that were built up from constructors, variables and parameters. The motivation for this is that we wanted to understand SML programs. And indeed we have seen that there is a close connection between SML programs on the one side and abstract data types and procedures on the other side. However, this analysis only holds on a very high level, SML programs are not terms per se, but sequences of characters we type to the keyboard or load from files. We only interpret them to be terms in the analysis of programs.

To drive our understanding of programs further, we will first have to understand more about sequences of characters (strings) and the interpretation process that derives structured mathematical objects (like terms) from them. Of course, not every sequence of characters will be interpretable, so we will need a notion of (legal) well-formed sequence.

2.4.1 Formal Languages

We will now formally define the concept of strings and (building on that) formal languages.

The Mathematics of Strings

- ▷ **Definition 184** An alphabet A is a finite set; we call each element $a \in A$ a character, and an n-tuple of $s \in A^n$ a string (of length n over A).
- ▷ **Definition 185** Note that $A^0 = \{\langle\rangle\}$, where $\langle\rangle$ is the (unique) 0-tuple. With the definition above we consider $\langle\rangle$ as the string of length 0 and call it the empty string and denote it with ϵ
- ▷ Note: Sets $\neq$ Strings, e.g. $\{1, 2, 3\} = \{3, 2, 1\}$, but $\langle 1, 2, 3\rangle \neq \langle 3, 2, 1\rangle$.
- ▷ **Notation 186** We will often write a string $\langle c_1, \ldots, c_n\rangle$ as "$c_1 \ldots c_n$", for instance "$\mathtt{a}, \mathtt{b}, \mathtt{c}$" for $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}\rangle$
- ▷ **Example 187** Take $A = \{\mathtt{h}, \mathtt{1}, /\}$ as an alphabet. Each of the symbols h, 1, and / is a character. The vector $\langle /, /, \mathtt{1}, \mathtt{h}, \mathtt{1}\rangle$ is a string of length 5 over A.
- ▷ **Definition 188 (String Length)** Given a string s we denote its length with $|s|$.
- ▷ **Definition 189** The concatenation $\mathrm{conc}(s, t)$ of two strings $s = \langle s_1, ..., s_n\rangle \in A^n$ and $t = \langle t_1, ..., t_m\rangle \in A^m$ is defined as $\langle s_1, ..., s_n, t_1, ..., t_m\rangle \in A^{n+m}$.

  We will often write $\mathrm{conc}(s, t)$ as $s + t$ or simply st (e.g. $\mathrm{conc}(\text{"}\mathtt{t,e,x,t}\text{"}, \text{"}\mathtt{b,o,o,k}\text{"}) = \text{"}\mathtt{t,e,x,t}\text{"} + \text{"}\mathtt{b,o,o,k}\text{"} = \text{"}\mathtt{t,e,x,t,b,o,o,k}\text{"}$)

We have multiple notations for concatenation, since it is such a basic operation, which is used so often that we will need very short notations for it, trusting that the reader can disambiguate based on the context.

Now that we have defined the concept of a string as a sequence of characters, we can go on to give ourselves a way to distinguish between good strings (e.g. programs in a given programming language) and bad strings (e.g. such with syntax errors). The way to do this by the concept of a formal language, which we are about to define.

Formal Languages

- ▷ **Definition 190** Let A be an alphabet, then we define the sets $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ of nonempty strings and $A^* := A^+ \cup \{\epsilon\}$ of strings.

▷ **Example 191** If $A = \{a, b, c\}$, then $A^* = \{\epsilon, a, b, c, aa, ab, ac, ba, \ldots, aaa, \ldots\}$.

▷ **Definition 192** A set $L \subseteq A^*$ is called a formal language in A.

▷ **Definition 193** We use $c^{[n]}$ for the string that consists of n times c.

▷ **Example 194** $\#^{[5]} = \langle\#, \#, \#, \#, \#\rangle$

▷ **Example 195** The set $M = \{ba^{[n]} \mid n \in \mathbb{N}\}$ of strings that start with character b followed by an arbitrary numbers of a's is a formal language in $A = \{a, b\}$.

▷ **Definition 196** The concatenation $\mathrm{conc}(L_1, L_2)$ of two languages L_1 and L_2 over the same alphabet is defined as $\mathrm{conc}(L_1, L_2) := \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$.

 117 JACOBS UNIVERSITY

There is a common misconception that a formal language is something that is difficult to understand as a concept. This is not true, the only thing a formal language does is separate the "good" from the bad strings. Thus we simply model a formal language as a set of stings: the "good" strings are members, and the "bad" ones are not.

Of course this definition only shifts complexity to the way we construct specific formal languages (where it actually belongs), and we have learned two (simple) ways of constructing them by repetition of characters, and by concatenation of existing languages.

Substrings and Prefixes of Strings

▷ **Definition 197** Let A be an alphabet, then we say that a string $s \in A^*$ is a substring of a string $t \in A^*$ (written $s \subseteq t$), iff there are strings $v, w \in A^*$, such that $t = vsw$.

▷ **Example 198** $\mathrm{conc}(/, \mathtt{1}, \mathtt{h})$ is a substring of $\mathrm{conc}(/, /, \mathtt{1}, \mathtt{h}, \mathtt{1})$, whereas $\mathrm{conc}(/, \mathtt{1}, \mathtt{1})$ is not.

▷ **Definition 199** A string p is a called a prefix of s (write $p \trianglelefteq s$), iff there is a string t, such that $s = \mathrm{conc}(p, t)$. p is a proper prefix of s (write $p \triangleleft s$), iff $t \neq \epsilon$.

▷ **Example 200** $text$ is a prefix of $textbook = \mathrm{conc}(text, book)$.

▷ Note: A string is never a proper prefix of itself.

 118 JACOBS UNIVERSITY

We will now define an ordering relation for formal languages. The nice thing is that we can induce an ordering on strings from an ordering on characters, so we only have to specify that (which is simple for finite alphabets).

Lexical Order

▷ **Definition 201** Let A be an alphabet and $<_A$ a partial order on A, then we define a relation $<_{\mathsf{lex}}$ on A^* by

$$s <_{\mathsf{lex}} t :\Leftrightarrow s \triangleleft t \vee (\exists u, v, w \in A^*.\exists a, b \in A.s = wau \wedge t = wbv \wedge (a <_A b))$$

for $s, t \in A^*$. We call $<_{\mathsf{lex}}$ the lexical order induced by $<_A$ on A^*.

▷ **Theorem 202** *$<_{\mathsf{lex}}$ is a partial order. If $<_A$ is defined as total order, then $<_{\mathsf{lex}}$ is total.*

▷ **Example 203** Roman alphabet with $\mathtt{a}{<}\mathtt{b}{<}\mathtt{c}\cdots{<}\mathtt{z}$ $\leadsto$ telephone book order $((computer <_{\mathsf{lex}} text), (text <_{\mathsf{lex}} textbook))$

 119 JACOBS UNIVERSITY

Even though the definition of the lexical ordering is relatively involved, we know it very well, it is the ordering we know from the telephone books.

The next task for understanding programs as mathematical objects is to understand the process of using strings to encode objects. The simplest encodings or "codes" are mappings from strings to strings. We will now study their properties.

2.4.2 Elementary Codes

The most characterizing property for a code is that if we encode something with this code, then we want to be able to decode it again: We model a code as a function (every character should have a unique encoding), which has a partial inverse (so we can decode). We have seen above, that this is is the case, iff the function is injective; so we take this as the defining characteristic of a code.

Character Codes

- ▷ **Definition 204** Let A and B be alphabets, then we call an injective function $c\colon A \to B^+$ a character code. A string $c(w) \in \{c(a) \mid a \in A\} := B^+$ is called a codeword.
- ▷ **Definition 205** A code is a called binary iff $B = \{0, 1\}$.
- ▷ **Example 206** Let $A = \{a, b, c\}$ and $B = \{0, 1\}$, then $c\colon A \to B^+$ with $c(a) = 0011$, $c(b) = 1101$, $c(c) = 0110$ c is a binary character code and the strings 0011, 1101, and 0110 are the codewords of c.
- ▷ **Definition 207** The extension of a code (on characters) $c\colon A \to B^+$ to a function $c'\colon A^* \to B^*$ is defined as $c'(\langle a_1, \ldots, a_n\rangle = \langle c(a_1), \ldots, c(a_n)\rangle)$.
- ▷ **Example 208** The extension c' of c from the above example on the string "b, b, a, b, c"

$$c'(\text{"b, b, a, b, c"}) = \underbrace{1101}_{c(b)}, \underbrace{1101}_{c(b)}, \underbrace{0011}_{c(a)}, \underbrace{1101}_{c(b)}, \underbrace{0110}_{c(c)}$$

 120 JACOBS UNIVERSITY

Morse Code

- ▷ In the early days of telecommunication the "Morse Code" was used to transmit texts, using long and short pulses of electricity.
- ▷ **Definition 209 (Morse Code)** The following table gives the Morse code for the text characters:

A	.-	B	-...	C	-.-.	D	-..	E	.
F	..-.	G	–.	H	....	I	..	J	.—
K	-.-	L	.-..	M	–	N	-.	O	—
P	.–.	Q	–.-	R	.-.	S	...	T	-
U	..-	V	...-	W	.–	X	-..-	Y	-.–
Z	–..								
1	.—-	2	..—	3	...–	4	....-	5	.....
6	-....	7	–...	8	—..	9	—-.	0	—

Furthermore, the Morse code uses .–.–.– for full stop (sentence termination), ––..–– for comma, and ..––.. for question mark.

▷ **Example 210** The Morse Code in the table above induces a character code $\mu\colon \mathcal{R} \to \{.,-\}$.

Codes on Strings

▷ **Definition 211** A function $c'\colon A^* \to B^*$ is called a code on strings or short string code if c' is an injective function.

▷ **Theorem 212** (⚠2) *There are character codes whose extensions are not string codes.*

▷ Proof: we give an example

P.1 Let $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $B = \{0, 1\}$, $c(\mathtt{a}) = 0$, $c(\mathtt{b}) = 1$, and $c(\mathtt{c}) = 01$.

P.2 The function c is injective, hence it is a character code.

P.3 But its extension c' is not injective as $c'(\mathtt{ab}) = 01 = c'(\mathtt{c})$. □

Question: When is the extension of a character code a string code?(so we can encode strings)

▷ **Definition 213** A (character) code $c\colon A \to B^+$ is a prefix code iff none of the codewords is a proper prefix to an other codeword, i.e.,

$$\forall x, y \in A. x \neq y \Rightarrow (c(x) \not\lhd c(y) \wedge c(y) \not\lhd c(x))$$

We will answer the question above by proving one of the central results of elementary coding theory: *prefix codes induce string codes*. This plays back the infinite task of checking that a string code is injective to a finite task (checking whether a character code is a prefix code).

Prefix Codes induce Codes on Strings

▷ **Theorem 214** *The extension $c'\colon A^* \to B^*$ of a prefix code $c\colon A \to B^+$ is a string code.*

▷ Proof: We will prove this theorem via induction over the string length n

P.1 We show that c' is injective (decodable) on strings of length $n \in \mathbb{N}$.

P.1.1 $n = 0$ (base case):

If $|s| = 0$ then $c'(\epsilon) = \epsilon$, hence c' is injective.

P.1.2 $n = 1$ (another):

If $|s| = 1$ then $c' = c$ thus injective, as c is char. code.

P.1.3 Induction step (n to $n + 1$):

P.1.3.1 Let $a = a_0, \ldots, a_n$, And we only know $c'(a) = c(a_0), \ldots, c(a_n)$.

P.1.3.2 It is easy to find $c(a_0)$ in $c'(a)$: It is the prefix of $c'(a)$ that is in $c(A)$. This is uniquely determined, since c is a prefix code. If there were two distinct ones, one would have to be a prefix of the other, which contradicts our assumption that c is a prefix code.

P.1.3.3 If we remove $c(a_0)$ from $c(a)$, we only have to decode $c(a_1), \ldots, c(a_n)$, which we can do by inductive hypothesis. □

P.2 Thus we have considered all the cases, and proven the assertion. □

Now, checking whether a code is a prefix code can be a tedious undertaking: the naive algorithm for this needs to check all pairs of codewords. Therefore we will look at a couple of properties of character codes that will ensure a prefix code and thus decodeability.

Sufficient Conditions for Prefix Codes

▷ **Theorem 215** *If c is a code with $|c(a)| = k$ for all $a \in A$ for some $k \in \mathbb{N}$, then c is prefix code.*

▷ Proof: by contradiction.

P.1 If c is not at prefix code, then there are $a, b \in A$ with $c(a) \triangleleft c(b)$.

P.2 clearly $|c(a)| < |c(b)|$, which contradicts our assumption. □

▷ **Theorem 216** *Let $c \colon A \to B^+$ be a code and $* \notin B$ be a character, then there is a prefix code $c^* \colon A \to (B \cup \{*\})^+$, such that $c(a) \triangleleft c^*(a)$, for all $a \in A$.*

▷ Proof: Let $c^*(a) := c(a) + "*"$ for all $a \in A$.

P.1 Obviously, $c(a) \triangleleft c^*(a)$.

P.2 If c^* is not a prefix code, then there are $a, b \in A$ with $c^*(a) \triangleleft c^*(b)$.

P.3 So, $c^*(b)$ contains the character $*$ not only at the end but also somewhere in the middle.

P.4 This contradicts our construction $c^*(b) = c(b) + "*"$, where $c(b) \in B^+$ □

2.4.3 Character Codes in the Real World

We will now turn to a class of codes that are extremely important in information technology: character encodings. The idea here is that for IT systems we need to encode characters from our alphabets as bit strings (sequences of binary digits 0 and 1) for representation in computers. Indeed the Morse code we have seen above can be seen as a very simple example of a character encoding that is geared towards the manual transmission of natural languages over telegraph lines. For the encoding of written texts we need more extensive codes that can e.g. distinguish upper and lowercase letters.

The `ASCII` code we will introduce here is one of the first standardized and widely used character encodings for a complete alphabet. It is still widely used today. The code tries to strike a balance between a being able to encode a large set of characters and the representational capabiligies in the time of punch cards (cardboard cards that represented sequences of binary numbers by rectangular arrays of dots).[6] EdNote:6

The ASCII Character Code

▷ **Definition 217** The American Standard Code for Information Interchange (`ASCII`) code assigns characters to numbers 0-127

Code	⋯0	⋯1	⋯2	⋯3	⋯4	⋯5	⋯6	⋯7	⋯8	⋯9	⋯A	⋯B	⋯C	⋯D	⋯E	⋯F
0⋯	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1⋯	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2⋯	␣	!	”	#	$	%	&	’	(	)	*	+	,	−	.	/
3⋯	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4⋯	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5⋯	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6⋯	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7⋯	p	q	r	s	t	u	v	w	x	y	z	{	$\vert$	}	~	DEL

[6] EdNote: is the 7-bit grouping really motivated by the cognitive limit?

The first 32 characters are control characters for ASCII devices like printers

- Motivated by punchcards: The character 0 (binary 000000) carries no information NUL, (used as dividers)

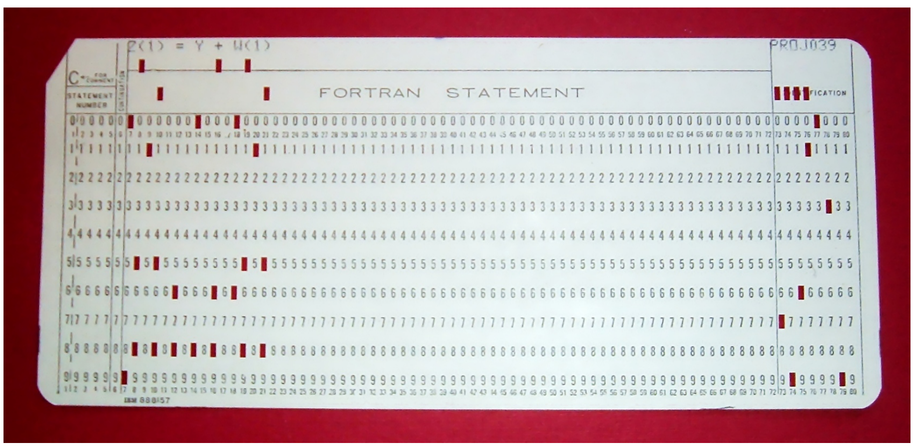
  Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)

- The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)

A Punchcard

- A punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions.

- **Example 218** This punch card encoded the `Fortran` statement `Z(1) = Y + W(1)`



The `ASCII` code as above has a variety of problems, for instance that the control characters are mostly no longer in use, the code is lacking many characters of languages other than the English language it was developed for, and finally, it only uses seven bits, where a byte (eight bits) is the preferred unit in information technology. Therefore there have been a whole zoo of extensions, which — due to the fact that there were so many of them — never quite solved the encoding problem.

Problems with `ASCII` encoding

- Problem: Many of the control characters are obsolete by now (e.g. `NUL`,`BEL`, or `DEL`)

- Problem: Many European characters are not represented (e.g. è,ñ,ü,ß,...)

- European ASCII Variants: Exchange less-used characters for national ones

- **Example 219 (German `ASCII`)** remap e.g. [↦ Ä,] ↦ Ü in German `ASCII` (“`Apple ][`” comes out as “`Apple ÜÄ`”)

▷ **Definition 220 (ISO-Latin (ISO/IEC 8859))** 16 Extensions of ASCII to 8-bit (256 characters) ISO-Latin 1 $\hat{=}$ "Western European", ISO-Latin 6 $\hat{=}$ "Arabic",ISO-Latin 7 $\hat{=}$ "Greek"...

▷ Problem: No cursive Arabic, Asian, African, Old Icelandic Runes, Math,...

▷ Idea: Do something totally different to include all the world's scripts: For a scalable architecture, separate

- ▹ what characters are available from the (character set)
- ▹ bit string-to-character mapping (character encoding)

 127 JACOBS UNIVERSITY

The goal of the UniCode standard is to cover all the worlds scripts (past, present, and future) and provide efficient encodings for them. The only scripts in regular use that are currently excluded are fictional scripts like the elvish scripts from the Lord of the Rings or Klingon scripts from the Star Trek series.

An important idea behind UniCode is to separate concerns between standardizing the character set — i.e. the set of encodable characters and the encoding itself.

Unicode and the Universal Character Set

▷ **Definition 221 (Twin Standards)** A scalable Architecture for representing all the worlds scripts

- ▹ The Universal Character Set defined by the ISO/IEC 10646 International Standard, is a standard set of characters upon which many character encodings are based.
- ▹ The Unicode Standard defines a set of standard character encodings, rules for normalization, decomposition, collation, rendering and bidirectional display order

▷ **Definition 222** Each UCS character is identified by an unambiguous name and an integer number called its code point.

▷ The UCS has 1.1 million code points and nearly 100 000 characters.

▷ **Definition 223** Most (non-Chinese) characters have code points in $[1, 65536]$ (the basic multilingual plane).

▷ **Notation 224** For code points in the Basic Multilingual Plane (BMP), four digits are used, e.g. U+0058 for the character LATIN CAPITAL LETTER X;

 128 JACOBS UNIVERSITY

Note that there is indeed an issue with space-efficient encoding here. UniCode reserves space for 2^{32} (more than a million) characters to be able to handle future scripts. But just simply using 32 bits for every UniCode character would be extremely wasteful: UniCode-encoded versions of `ASCII` files would be four times as large.

Therefore UniCode allows multiple encodings. UTF-32 is a simple 32-bit code that directly uses the code points in binary form. UTF-8 is optimized for western languages and coincides with the `ASCII` where they overlap. As a consequence, `ASCII` encoded texts can be decoded in UTF-8 without changes — but in the UTF-8 encoding, we can also address all other UniCode characters (using multi-byte characters).

Character Encodings in Unicode

▷ **Definition 225** A character encoding is a mapping from bit strings to UCS code points.

- ▷ Idea: Unicode supports multiple encodings (but not character sets) for efficiency
- ▷ **Definition 226 (Unicode Transformation Format)** ▷ UTF-8, 8-bit, variable-width encoding, which maximizes compatibility with ASCII.
  - ▷ UTF-16, 16-bit, variable-width encoding (popular in Asia)
  - ▷ UTF-32, a 32-bit, fixed-width encoding (for safety)
- ▷ **Definition 227** The UTF-8 encoding follows the following encoding scheme

Unicode	Byte1	Byte2	Byte3	Byte4
U+000000 – U+00007F	0xxxxxxx			
U+000080 – U+0007FF	110xxxxx	10xxxxxx		
U+000800 – U+00$FFFF$	1110xxxx	10xxxxxx	10xxxxxx	
U+010000 – U+10$FFFF$	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- ▷ **Example 228** \$ = U+0024 is encoded as 00100100 (1 byte)

  ¢ = U+00A2 is encoded as 11000010,10100010 (two bytes)

  e = U+20AC is encoded as 11100010,10000010,10101100 (three bytes)

 129 JACOBS UNIVERSITY

Note how the fixed bit prefixes in the encoding are engineered to determine which of the four cases apply, so that UTF-8 encoded documents can be safely decoded..

2.4.4 Formal Languages and Meaning

After we have studied the elementary theory of codes for strings, we will come to string representations of structured objects like terms. For these we will need more refined methods.

As we have started out the course with unary natural numbers and added the arithmetical operations to the mix later, we will use unary arithmetics as our running example and study object.

A formal Language for Unary Arithmetics

- ▷ Idea: Start with something very simple: Unary Arithmetics (i.e. $\mathbb{N}$ with addition, multiplication, subtraction, and integer division)
- ▷ E_{un} is based on the alphabet $\Sigma_{\text{un}} := C_{\text{un}} \cup V \cup F^2_{\text{un}} \cup B$, where
  - ▷ $C_{\text{un}} := \{/\}^*$ is a set of constant names,
  - ▷ $V := \{\mathtt{x}\} \times \{1, \ldots, 9\} \times \{0, \ldots, 9\}^*$ is a set of variable names,
  - ▷ $F^2_{\text{un}} := \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{mod}\}$ is a set of (binary) function names, and
  - ▷ $B := \{(,)\} \cup \{,\}$ is a set of structural characters. (“,”,”(“,”)” characters!)
- ▷ define strings in stages: $E_{\text{un}} := \bigcup_{i \in \mathbb{N}} E^i_{\text{un}}$, where
  - ▷ $E^1_{\text{un}} := C_{\text{un}} \cup V$
  - ▷ $E^{i+1}_{\text{un}} := \{a, \mathsf{add}(a,b), \mathsf{sub}(a,b), \mathsf{mul}(a,b), \mathsf{div}(a,b), \mathsf{mod}(a,b) \mid a, b \in E^i_{\text{un}}\}$

  We call a string in E_{un} an expression of unary arithmetics.

 130 JACOBS UNIVERSITY

The first thing we notice is that the alphabet is not just a flat any more, we have characters with different roles in the alphabet. These roles have to do with the symbols used in the complex objects (unary arithmetic expressions) that we want to encode.

The formal language E_{un} is constructed in stages, making explicit use of the respective roles of the characters in the alphabet. Constants and variables form the basic inventory in E_{un}^1, the respective next stage is built up using the function names and the structural characters to encode the applicative structure of the encoded terms.

Note that with this construction $E_{\text{un}}^i \subseteq E_{\text{un}}^{i+1}$.

A formal Language for Unary Arithmetics (Examples)

- **Example 229** add(//////,mul(x1902,///)) $\in E_{\text{un}}$
- Proof: we proceed according to the definition

  P.1 We have ////// $\in C_{\text{un}}$, and x1902 $\in V$, and /// $\in C_{\text{un}}$ by definition

  P.2 Thus ////// $\in E_{\text{un}}^1$, and x1902 $\in E_{\text{un}}^1$ and /// $\in E_{\text{un}}^1$,

  P.3 Hence, ////// $\in E_{\text{un}}^2$ and mul(x1902,///) $\in E_{\text{un}}^2$

  P.4 Thus add(//////,mul(x1902,///)) $\in E_{\text{un}}^3$

  P.5 And finally add(//////,mul(x1902,///)) $\in E_{\text{un}}$ □

- other examples:
  - div(x201,add(////,x12))
  - sub(mul(///,div(x23,///)),///)
- what does it all mean? (nothing, E_{un} is just a set of strings!)

 131 JACOBS UNIVERSITY

To show that a string is an expression s of unary arithmetics, we have to show that it is in the formal language E_{un}. As E_{un} is the union over all the E_{un}^i, the string s must already be a member of a set E_{un}^j for some $j \in \mathbb{N}$. So we reason by the definintion establising set membership.

Of course, computer science has better methods for defining languages than the ones used here (context free grammars), but the simple methods used here will already suffice to make the relevant points for this course.

Syntax and Semantics (a first glimpse)

- **Definition 230** A formal language is also called a syntax, since it only concerns the "form" of strings.
- to give meaning to these strings, we need a semantics, i.e. a way to interpret these.
- Idea (Tarski Semantics): A semantics is a mapping from strings to objects we already know and understand (e.g. arithmetics).
  - e.g. add(//////,mul(x1902,///)) $\mapsto 6 + (x_{1907} \cdot 3)$ (but what does this mean?)
  - looks like we have to give a meaning to the variables as well, e.g. x1902 $\mapsto 3$, then add(//////,mul(x1902,///)) $\mapsto 6 + (3 \cdot 3) = 15$

 132 JACOBS UNIVERSITY

So formal languages do not mean anything by themselves, but a meaning has to be given to them via a mapping. We will explore that idea in more detail in the following.

2.5 Boolean Algebra

We will now look a formal language from a different perspective. We will interpret the language of "Boolean expressions" as formulae of a very simple "logic": A logic is a mathematical construct to study the association of meaning to strings and reasoning processes, i.e. to study how humans[4] derive new information and knowledge from existing one.

2.5.1 Boolean Expressions and their Meaning

In the following we will consider the Boolean Expressions as the language of "Propositional Logic", in many ways the simplest of logics. This means we cannot really express very much of interest, but we can study many things that are common to all logics.

Let us try again (Boolean Expressions)

- ▷ **Definition 231 (Alphabet)** E_{bool} is based on the alphabet $\mathcal{A} := C_{\mathsf{bool}} \cup V \cup F^1_{\mathsf{bool}} \cup F^2_{\mathsf{bool}} \cup B$, where $C_{\mathsf{bool}} = \{0,1\}$, $F^1_{\mathsf{bool}} = \{-\}$ and $F^2_{\mathsf{bool}} = \{+,*\}$. ($V$ and B as in E_{un})
- ▷ **Definition 232 (Formal Language)** $E_{\mathsf{bool}} := \bigcup_{i\in\mathbb{N}} E^i_{\mathsf{bool}}$, where $E^1_{\mathsf{bool}} := C_{\mathsf{bool}} \cup V$ and $E^{i+1}_{\mathsf{bool}} := \{a, (-a), (a+b), (a*b) \mid a, b \in E^i_{\mathsf{bool}}\}$.
- ▷ **Definition 233** Let $a \in E_{\mathsf{bool}}$. The minimal i, such that $a \in E^i_{\mathsf{bool}}$ is called the depth of a.
- ▷ $e_1 := ((-\mathrm{x1})+\mathrm{x3})$ (depth 3)
- ▷ $e_2 := ((-(\mathrm{x1}*\mathrm{x2}))+(\mathrm{x3}*\mathrm{x4}))$ (depth 4)
- ▷ $e_3 := ((\mathrm{x1}+\mathrm{x2})+((-((-\mathrm{x1})*\mathrm{x2}))+(\mathrm{x3}*\mathrm{x4})))$ (depth 6)

 133 JACOBS UNIVERSITY

Boolean Expressions as Structured Objects.

- ▷ Idea: As strings in in E_{bool} are built up via the "union-principle", we can think of them as constructor terms with variables
- ▷ **Definition 234** The abstract data type

$$\mathcal{B} := \langle\{\mathbb{B}\}, \{[1\colon \mathbb{B}], [0\colon \mathbb{B}], [-\colon \mathbb{B} \to \mathbb{B}], [+\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}], [*\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}]\}\rangle$$

- ▷ via the translation
- ▷ **Definition 235** $\sigma\colon E_{\mathsf{bool}} \to \mathcal{T}_{\mathbb{B}}(\mathcal{B}; \mathcal{V})$ defined by

$$\begin{array}{ll} \sigma(1) := 1 & \sigma(0) := 0 \\ \sigma((-A)) := (-\sigma(A)) & \\ \sigma((A*B)) := (\sigma(A)*\sigma(B)) & \sigma((A+B)) := (\sigma(A)+\sigma(B)) \end{array}$$

- ▷ We will use this intuition for our treatment of Boolean expressions and treak the strings and constructor terms synonymouslhy. (σ is a (hidden) isomorphism)

[4] until very recently, humans were thought to be the only systems that could come up with complex argumentations. In the last 50 years this has changed: not only do we attribute more reasoning capabilities to animals, but also, we have developed computer systems that are increasingly capable of reasoning.

▷ **Definition 236** We will write $(-A)$ as $\overline{A}$ and $(A{*}B)$ as $A * B$ (and similarly for $+$). Furthermore we will write variables such as $x71$ as x_{71} and elide brackets for sums and products according to their usual precedences.

▷ **Example 237** $\sigma(((-(\mathrm{x1}{*}\mathrm{x2}))+(\mathrm{x3}{*}\mathrm{x4}))) = \overline{x_1 * x_2} + x_3 * x_4$

▷ ⚠: Do not confuse $+$ and $*$ (Boolean sum and product) with their arithmetic counterparts. (as members of a formal language they have no meaning!)

  JACOBS UNIVERSITY

Now that we have defined the formal language, we turn the process of giving the strings a meaning. We make explicit the idea of providing meaning by specifying a function that assigns objects that we already understand to representations (strings) that do not have a priori meaning.

The first step in assigning meaning is to fix a set of objects what we will assign as meanings: the "universe (of discourse)". To specify the meaning mapping, we try to get away with specifying as little as possible. In our case here, we assign meaning only to the constants and functions and induce the meaning of complex expressions from these. As we have seen before, we also have to assign meaning to variables (which have a different ontological status from constants); we do this by a special meaning function: a variable assignment.

Boolean Expressions: Semantics via Models

▷ **Definition 238** A model $\langle \mathcal{U}, \mathcal{I} \rangle$ for E_{bool} is a set $\mathcal{U}$ of objects (called the universe) together with an interpretation function $\mathcal{I}$ on $\mathcal{A}$ with $\mathcal{I}(C_{\mathsf{bool}}) \subseteq \mathcal{U}$, $\mathcal{I}(F^1_{\mathsf{bool}}) \subseteq \mathcal{F}(\mathcal{U};\mathcal{U})$, and $\mathcal{I}(F^2_{\mathsf{bool}}) \subseteq \mathcal{F}(\mathcal{U}^2;\mathcal{U})$.

▷ **Definition 239** A function $\varphi\colon V \to \mathcal{U}$ is called a variable assignment.

▷ **Definition 240** Given a model $\langle \mathcal{U}, \mathcal{I} \rangle$ and a variable assignment φ, the evaluation function $\mathcal{I}_\varphi\colon E_{\mathsf{bool}} \to \mathcal{U}$ is defined recursively: Let $c \in C_{\mathsf{bool}}$, $a, b \in E_{\mathsf{bool}}$, and $x \in V$, then

- ▷ $\mathcal{I}_\varphi(c) = \mathcal{I}(c)$, for $c \in C_{\mathsf{bool}}$
- ▷ $\mathcal{I}_\varphi(x) = \varphi(x)$, for $x \in V$
- ▷ $\mathcal{I}_\varphi(\overline{a}) = \mathcal{I}(-)(\mathcal{I}_\varphi(a))$
- ▷ $\mathcal{I}_\varphi(a + b) = \mathcal{I}(+)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$ and $\mathcal{I}_\varphi(a * b) = \mathcal{I}(*)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$

▷ $\mathcal{U} = \{\mathsf{T}, \mathsf{F}\}$ with $0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg$.

▷ $\mathcal{U} = E_{\mathsf{un}}$ with $0 \mapsto /, 1 \mapsto //, + \mapsto div, * \mapsto mod, - \mapsto \lambda x.5$.

▷ $\mathcal{U} = \{0, 1\}$ with $0 \mapsto 0, 1 \mapsto 1, + \mapsto \min, * \mapsto \max, - \mapsto \lambda x.1 - x$.

  JACOBS UNIVERSITY

Note that all three models on the bottom of the last slide are essentially different, i.e. there is no way to build an isomorphism between them, i.e. a mapping between the universes, so that all Boolean expressions have corresponding values.

To get a better intuition on how the meaning function works, consider the following example. We see that the value for a large expression is calculated by calculating the values for its sub-expressions and then combining them via the function that is the interpretation of the constructor at the head of the expression.

Evaluating Boolean Expressions

▷ **Example 241** Let $\varphi := [\mathsf{T}/x_1], [\mathsf{F}/x_2], [\mathsf{T}/x_3], [\mathsf{F}/x_4]$, and $\mathcal{I} = \{0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\}$, then

$$
\begin{aligned}
& \mathcal{I}_\varphi((x_1+x_2)+(\overline{\overline{x_1}*x_2}+x_3*x_4)) \\
= \; & \mathcal{I}_\varphi(x_1+x_2) \vee \mathcal{I}_\varphi(\overline{\overline{x_1}*x_2}+x_3*x_4) \\
= \; & \mathcal{I}_\varphi(x_1) \vee \mathcal{I}_\varphi(x_2) \vee \mathcal{I}_\varphi(\overline{\overline{x_1}*x_2}) \vee \mathcal{I}_\varphi(x_3*x_4) \\
= \; & \varphi(x_1) \vee \varphi(x_2) \vee \neg(\mathcal{I}_\varphi(\overline{x_1}*x_2)) \vee \mathcal{I}_\varphi(x_3*x_4) \\
= \; & (\mathsf{T} \vee \mathsf{F}) \vee (\neg(\mathcal{I}_\varphi(\overline{x_1}) \wedge \mathcal{I}_\varphi(x_2)) \vee (\mathcal{I}_\varphi(x_3) \wedge \mathcal{I}_\varphi(x_4))) \\
= \; & \mathsf{T} \vee \neg(\neg(\mathcal{I}_\varphi(x_1)) \wedge \varphi(x_2)) \vee (\varphi(x_3) \wedge \varphi(x_4)) \\
= \; & \mathsf{T} \vee \neg(\neg(\varphi(x_1)) \wedge \mathsf{F}) \vee (\mathsf{T} \wedge \mathsf{F}) \\
= \; & \mathsf{T} \vee \neg(\neg(\mathsf{T}) \wedge \mathsf{F}) \vee \mathsf{F} \\
= \; & \mathsf{T} \vee \neg(\mathsf{F} \wedge \mathsf{F}) \vee \mathsf{F} \\
= \; & \mathsf{T} \vee \neg(\mathsf{F}) \vee \mathsf{F} = \mathsf{T} \vee \mathsf{T} \vee \mathsf{F} = \mathsf{T}
\end{aligned}
$$

What a mess!

A better mouse-trap: Truth Tables

▷ Truth tables to visualize truth functions:

$\overline{\cdot}$	
T	F
F	T

$*$	T	F
T	T	F
F	F	F

$+$	T	F
T	T	T
F	T	F

▷ If we are interested in values for all assignments (e.g. of $x_{123} * x_4 + \overline{x_{123} * x_{72}}$)

assignments			intermediate results			full
x_4	x_{72}	x_{123}	$e_1 := x_{123} * x_{72}$	$e_2 := \overline{e_1}$	$e_3 := x_{123} * x_4$	$e_3 + e_2$
F	F	F	F	T	F	T
F	F	T	F	T	F	T
F	T	F	F	T	F	T
F	T	T	T	F	F	F
T	F	F	F	T	F	T
T	F	T	F	T	T	T
T	T	F	F	T	F	T
T	T	T	T	F	T	T

Boolean Algebra

▷ **Definition 242** A Boolean algebra is E_{bool} together with the models

▷ $\langle \{\mathsf{T}, \mathsf{F}\}, \{0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\} \rangle$.

▷ $\langle \{0, 1\}, \{0 \mapsto 0, 1 \mapsto 1, + \mapsto \mathsf{max}, * \mapsto \mathsf{min}, - \mapsto \lambda x.1 - x\} \rangle$.

▷ BTW, the models are equivalent ($0 \hat{=} \mathsf{F}$, $1 \hat{=} \mathsf{T}$)

▷ **Definition 243** We will use $\mathbb{B}$ for the universe, which can be either $\{0, 1\}$ or $\{\mathsf{T}, \mathsf{F}\}$

▷ **Definition 244** We call two expressions $e_1, e_2 \in E_{\mathsf{bool}}$ equivalent (write $e_1 \equiv e_2$), iff $\mathcal{I}_\varphi(e_1) = \mathcal{I}_\varphi(e_2)$ for all φ.

▷ **Theorem 245** *$e_1 \equiv e_2$, iff $(\overline{e1} + e_2) * (e_1 + \overline{e_2})$ is a theorem of Boolean Algebra.*

As we are mainly interested in the interplay between form and meaning in Boolean Algebra, we will often identify Boolean expressions, if they have the same values in all situations (as specified by the variable assignments). The notion of equivalent formulae formalizes this intuition.

Boolean Equivalences

- Given $a, b, c \in E_{\mathsf{bool}}$, $\circ \in \{+, *\}$, let $\hat{\circ} := \begin{cases} + & \text{if } \circ = * \\ * & \text{else} \end{cases}$
- We have the following equivalences in Boolean Algebra:
  - $a \circ b \equiv b \circ a$ (commutativity)
  - $(a \circ b) \circ c \equiv a \circ (b \circ c)$ (associativity)
  - $a \circ (b \hat{\circ} c) \equiv (a \circ b) \hat{\circ} (a \circ c)$ (distributivity)
  - $a \circ (a \hat{\circ} b) \equiv a$ (covering)
  - $(a \circ b) \hat{\circ} (a \circ \overline{b}) \equiv a$ (combining)
  - $(a \circ b) \hat{\circ} ((\overline{a} \circ c) \hat{\circ} (b \circ c)) \equiv (a \circ b) \hat{\circ} (\overline{a} \circ c)$ (consensus)
  - $\overline{a \circ b} \equiv \overline{a} \hat{\circ} \overline{b}$ (De Morgan)

2.5.2 Boolean Functions

We will now turn to "semantical" counterparts of Boolean expressions: Boolean functions. These are just n-ary functions on the Boolean values.

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

The theory we will develop now will help us understand how to "implement" Boolean functions (as specifications of computer chips), viewing Boolean expressions very abstract representations of configurations of logic gates and wiring. We will study the issues of representing such configurations in more detail later[7] EdNote:7

Boolean Functions

- **Definition 246** A Boolean function is a function from $\mathbb{B}^n$ to $\mathbb{B}$.
- **Definition 247** Boolean functions $f, g \colon \mathbb{B}^n \to \mathbb{B}$ are called equivalent, (write $f \equiv g$), iff $f(c) = g(c)$ for all $c \in \mathbb{B}^n$. (equal as functions)
- Idea: We can turn any Boolean expression into a Boolean function by ordering the variables (use the lexical ordering on $\{X\} \times \{1, \ldots, 9\}^+ \times \{0, \ldots, 9\}^*$)
- **Definition 248** Let $e \in E_{\mathsf{bool}}$ and $\{x_1, \ldots, x_n\}$ the set of variables in e, then call $VL(e) := \langle x_1, \ldots, x_n \rangle$ the variable list of e, iff $(x_i <_{\mathsf{lex}} x_j)$ where $i \leq j$.
- **Definition 249** Let $e \in E_{\mathsf{bool}}$ with $VL(e) = \langle x_1, \ldots, x_n \rangle$, then we call the function

  $$f_e \colon \mathbb{B}^n \to \mathbb{B} \text{ with } f_e \colon c \mapsto \mathcal{I}_{\varphi_c}(e)$$

  the Boolean function induced by e, where $\varphi_{\langle c_1, \ldots, c_n \rangle} \colon x_i \mapsto c_i$.
- **Theorem 250** $e_1 \equiv e_2$, *iff* $f_{e_1} = f_{e_2}$.

[7] EdNote: make a forward reference here.

The definition above shows us that in theory every Boolean Expression induces a Boolean function. The simplest way to compute this is to compute the truth table for the expression and then read off the function from the table.

Boolean Functions and Truth Tables

▷ The truth table of a Boolean function is defined in the obvious way:

x_1	x_2	x_3	$f_{x_1*(\overline{x_2}+x_3)}$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

▷ compute this by assigning values and evaluating

▷ Question: can we also go the other way? (from function to expression?)

▷ Idea: read expression of a special form from truth tables (Boolean Polynomials)

Computing a Boolean expression from a given Boolean function is more interesting — there are many possible candidates to choose from; after all any two equivalent expressions induce the same function. To simplify the problem, we will restrict the space of Boolean expressions that realize a given Boolean function by looking only for expressions of a given form.

Boolean Polynomials

▷ special form Boolean Expressions

- ▷ a literal is a variable or the negation of a variable
- ▷ a monomial or product term is a literal or the product of literals
- ▷ a clause or sum term is a literal or the sum of literals
- ▷ a Boolean polynomial or sum of products is a product term or the sum of product terms
- ▷ a clause set or product of sums is a sum term or the product of sum terms

For literals x_i, write x_i^1, for $\overline{x_i}$ write x_i^0. (⚠ not exponentials, but intended truth values)

▷ **Notation 251** Write $x_i x_j$ instead of $x_i * x_j$. (like in math)

Armed with this normal form, we can now define an way of realizing[8] Boolean functions. EdNote:8

Normal Forms of Boolean Functions

▷ **Definition 252** Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function and $c \in \mathbb{B}^n$, then $M_c := \prod_{j=1}^n x_j^{c_j}$ and $S_c := \sum_{j=1}^n x_j^{1-c_j}$

[8] EdNote: define that formally above

▷ **Definition 253** The disjunctive normal form (DNF) of f is $\sum_{c \in f^{-1}(1)} M_c$ (also called the canonical sum (written as $\mathrm{DNF}(f)$))

▷ **Definition 254** The conjunctive normal form (CNF) of f is $\prod_{c \in f^{-1}(0)} S_c$ (also called the canonical product (written as $\mathrm{CNF}(f)$))

x_1	x_2	x_3	f	monomials	clauses
0	0	0	1	$x_1^0 x_2^0 x_3^0$	
0	0	1	1	$x_1^0 x_2^0 x_3^1$	
0	1	0	0		$x_1^1 + x_2^0 + x_3^1$
0	1	1	0		$x_1^1 + x_2^0 + x_3^0$
1	0	0	1	$x_1^1 x_2^0 x_3^0$	
1	0	1	1	$x_1^1 x_2^0 x_3^1$	
1	1	0	0		$x_1^0 + x_2^0 + x_3^1$
1	1	1	1	$x_1^1 x_2^1 x_3^1$	

▷ DNF of f: $\overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2}\,x_3 + x_1\,\overline{x_2}\,\overline{x_3} + x_1\,\overline{x_2}\,x_3 + x_1\,x_2\,x_3$

▷ CNF of f: $(x_1 + \overline{x_2} + x_3)\,(x_1 + \overline{x_2} + \overline{x_3})\,(\overline{x_1} + \overline{x_2} + x_3)$

  JACOBS UNIVERSITY

In the light of the argument of understanding Boolean expressions as implementations of Boolean functions, the process becomes interesting while realizing specifications of chips. In particular it also becomes interesting, which of the possible Boolean expressions we choose for realizing a given Boolean function. We will analyze the choice in terms of the "cost" of a Boolean expression.

Costs of Boolean Expressions

▷ Idea: Complexity Analysis is about the estimation of resource needs

▷ if we have two expressions for a Boolean function, which one to choose?

▷ Idea: Let us just measure the size of the expression (after all it needs to be written down)

▷ Better Idea: count the number of operators (computation elements)

▷ **Definition 255** The cost $C(e)$ of $e \in E_{\mathrm{bool}}$ is the number of operators in e.

▷ **Example 256** $C(\overline{x_1} + x_3) = 2$, $C(\overline{x_1 * x_2} + x_3 * x_4) = 4$, $C((x_1 + x_2) + (\overline{x_1} * x_2 + x_3 * x_4)) = 7$

▷ **Definition 257** Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, then $C(f) := \min(\{C(e) \mid f = f_e\})$ is the cost of f.

▷ Note: We can find expressions of arbitrarily high cost for a given Boolean function.($e \equiv e * 1$)

▷ but how to find such an e with minimal cost for f?

  JACOBS UNIVERSITY

2.5.3 Complexity Analysis for Boolean Expressions

The Landau Notations (aka. "big-O" Notation)

▷ **Definition 258** Let $f, g\colon \mathbb{N} \to \mathbb{N}$, we say that f is asymptotically bounded by g, written as

$(f \leq_a g)$, iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.

▷ **Definition 259** The three Landau sets $O(g), \Omega(g), \Theta(g)$ are defined as

- $O(g) = \{f \mid \exists k > 0.f \leq_a k \cdot g\}$
- $\Omega(g) = \{f \mid \exists k > 0.f \geq_a k \cdot g\}$
- $\Theta(g) = O(g) \cap \Omega(g)$

Intuition: The Landau sets express the "shape of growth" of the graph of a function.

▷
- If $f \in O(g)$, then f grows at most as fast as g. ("f is in the order of g")
- If $f \in \Omega(g)$, then f grows at least as fast as g. ("f is at least in the order of g")
- If $f \in \Theta(g)$, then f grows as fast as g. ("f is strictly in the order of g")

Commonly used Landau Sets

▷

Landau set	class name	rank	Landau set	class name	rank
$O(1)$	constant	1	$O(n^2)$	quadratic	4
$O(\log_2(n))$	logarithmic	2	$O(n^k)$	polynomial	5
$O(n)$	linear	3	$O(k^n)$	exponential	6

▷ **Theorem 260** *These Ω-classes establish a ranking (increasing rank $\leadsto$ increasing growth)*

$$O(1) \subset O(\log_2(n)) \subset O(n) \subset O(n^2) \subset O(n^{k'}) \subset O(k^n)$$

where $k' > 2$ and $k > 1$. The reverse holds for the Ω-classes

$$\Omega(1) \supset \Omega(\log_2(n)) \supset \Omega(n) \supset \Omega(n^2) \supset \Omega(n^{k'}) \supset \Omega(k^n)$$

▷ Idea: Use O-classes for worst-case complexity analysis and Ω-classes for best-case.

Examples

▷ Idea: the fastest growth function in sum determines the O-class

▷ **Example 261** $(\lambda n.263748) \in O(1)$

▷ **Example 262** $(\lambda n.26n + 372) \in O(n)$

▷ **Example 263** $(\lambda n.7n^2 - 372n + 92) \in O(n^2)$

▷ **Example 264** $(\lambda n.857n^{10} + 7342n^7 + 26n^2 + 902) \in O(n^{10})$

▷ **Example 265** $(\lambda n.3 \cdot 2^n + 72) \in O(2^n)$

▷ **Example 266** $(\lambda n.3 \cdot 2^n + 7342n^7 + 26n^2 + 722) \in O(2^n)$

With the basics of complexity theory well-understood, we can now analyze the cost-complexity of Boolean expressions that realize Boolean functions. We will first derive two upper bounds for the cost of Boolean functions with n variables, and then a lower bound for the cost.

The first result is a very naive counting argument based on the fact that we can always realize a Boolean function via its DNF or CNF. The second result gives us a better complexity with a more involved argument. Another difference between the proofs is that the first one is constructive, i.e. we can read an algorithm that provides Boolean expressions of the complexity claimed by the algorithm for a given Boolean function. The second proof gives us no such algorithm, since it is non-constructive.

An Upper Bound for the Cost of BF with n variables

▷ Idea: Every Boolean function has a DNF and CNF, so we compute its cost.

▷ **Example 267** Let us look at the size of the DNF or CNF for $f \in (\mathbb{B}^3 \to \mathbb{B})$.

x_1	x_2	x_3	f	monomials	clauses
0	0	0	1	$x_1^0\, x_2^0\, x_3^0$	
0	0	1	1	$x_1^0\, x_2^0\, x_3^1$	
0	1	0	0		$x_1^1 + x_2^0 + x_3^1$
0	1	1	0		$x_1^1 + x_2^0 + x_3^0$
1	0	0	1	$x_1^1\, x_2^0\, x_3^0$	
1	0	1	1	$x_1^1\, x_2^0\, x_3^1$	
1	1	0	0		$x_1^0 + x_2^0 + x_3^1$
1	1	1	1	$x_1^1\, x_2^1\, x_3^1$	

▷ **Theorem 268** *Any $f : \mathbb{B}^n \to \mathbb{B}$ is realized by an $e \in E_{bool}$ with $C(e) \in O(n \cdot 2^n)$.*

Proof: by counting (constructive proof (we exhibit a witness))

▷ **P.1** either $e_n :=$ CNF(f) has $\frac{2^n}{2}$ clauses or less or DNF(f) does monomials

take smaller one, multiply/sum the monomials/clauses at cost $2^{n-1} - 1$

there are n literals per clause/monomial e_i, so $C(e_i) \leq 2n - 1$

so $C(e_n) \leq 2^{n-1} - 1 + 2^{n-1} \cdot (2n - 1)$ and thus $C(e_n) \in O(n \cdot 2^n)$ □

For this proof we will introduce the concept of a "realization cost function" $\kappa : \mathbb{N} \to \mathbb{N}$ to save space in the argumentation. The trick in this proof is to make the induction on the arity work by splitting an n-ary Boolean function into two $n-1$-ary functions and estimate their complexity separately. This argument does not give a direct witness in the proof, since to do this we have to decide which of these two split-parts we need to pursue at each level. This yields an algorithm for determining a witness, but not a direct witness itself.

We can do better (if we accept complicated witness)

P.2 P.3 P.4 ▷ **Theorem 269** *Let $\kappa(n) := max(\{C(f) \mid f : \mathbb{B}^n \to \mathbb{B}\})$, then $\kappa \in O(2^n)$.*

▷ Proof: we show that $\kappa(n) \leq 2^n + d$ by induction on n

P.1.1 base case:

We count the operators in all members: $\mathbb{B} \to \mathbb{B} = \{f_1, f_0, f_{x_1}, f_{\overline{x_1}}\}$,

so $\kappa(1) = 1$ and thus $\kappa(1) \leq 2^1 + d$ for $d = 0$.

P.1.2 step case:

P.1.2.1 given $f \in (\mathbb{B}^n \to \mathbb{B})$, then $f(a_1, \ldots, a_n) = 1$, iff either

- ▷ $a_n = 0$ and $f(a_1, \ldots, a_{n-1}, 0) = 1$ or
- ▷ $a_n = 1$ and $f(a_1, \ldots, a_{n-1}, 1) = 1$

P.1.2.2 Let $f_i(a_1, \ldots, a_{n-1}) := f(a_1, \ldots, a_{n-1}, i)$ for $i \in \{0, 1\}$,

P.1.2.3 then there are $e_i \in E_{\text{bool}}$, such that $f_i = f_{e_i}$ and $C(e_i) = 2^{n-1} + d$. (IH)

P.1.2.4 thus $f = f_e$, where $e := \overline{x_n} * e_0 + x_n * e_1$ and $\kappa(n) = 2 \cdot 2^{n-1} + 2d + 4$. □

□

The next proof is quite a lot of work, so we will first sketch the overall structure of the proof, before we look into the details. The main idea is to estimate a cleverly chosen quantity from above and below, to get an inequality between the lower and upper bounds (the quantity itself is irrelevant except to make the proof work).

A Lower Bound for the Cost of BF with n Variables

▷ **Theorem 270** $\kappa \in \Omega(\frac{2^n}{\log_2(n)})$

▷ Proof: Sketch (counting again!)

P.1 the cost of a function is based on the cost of expressions.

P.2 consider the set $\mathcal{E}_n$ of expressions with n variables of cost no more than $\kappa(n)$.

P.3 find an upper and lower bound for $\#(\mathcal{E}_n)$: $(\Phi(n) \leq \#(\mathcal{E}_n) \leq \Psi(\kappa(n)))$

P.4 in particular: $\Phi(n) \leq \Psi(\kappa(n))$

P.5 solving for $\kappa(n)$ yields $\kappa(n) \geq \Xi(n)$ so $\kappa \in \Omega(\frac{2^n}{\log_2(n)})$ □

▷ We will expand P.3 and P.5 in the next slides

A Lower Bound For $\kappa(n)$-Cost Expressions

▷ **Definition 271** $\mathcal{E}_n := \{e \in E_{\text{bool}} \mid e \text{ has } n \text{ variables and } C(e) \leq \kappa(n)\}$

▷ **Lemma 272** $\#(\mathcal{E}_n) \geq \#(\mathbb{B}^n \to \mathbb{B})$

▷ Proof:

P.1 For all $f_n \in \mathbb{B}^n \to \mathbb{B}$ we have $C(f_n) \leq \kappa(n)$

P.2 $C(f_n) = \min(\{C(e) \mid f_e = f_n\})$ choose e_{f_n} with $C(e_{f_n}) = C(f_n)$

P.3 all distinct: if $e_g \equiv e_h$, then $f_{e_g} = f_{e_h}$ and thus $g = h$. □

▷ **Corollary 273** $\#(\mathcal{E}_n) \geq 2^{2^n}$

Proof: consider the n dimensional truth tables

▷ **P.1** 2^n entries that can be either 0 or 1, so 2^{2^n} possibilities

so $\#(\mathbb{B}^n \to \mathbb{B}) = 2^{2^n}$ □

An Upper Bound For $\kappa(n)$-cost Expressions

P.2 ▷ Idea: Estimate the number of E_{bool} strings that can be formed at a given cost by looking at the length and alphabet size.

▷ **Definition 274** Given a cost c let $\Lambda(e)$ be the length of e considering variables as single characters. We define

$$\sigma(c) := \mathsf{max}(\{\Lambda(e) \mid e \in E_{\mathsf{bool}} \wedge (C(e) \leq c)\})$$

▷ **Lemma 275** $\sigma(n) \leq 5n$ *for* $n > 0$.

▷ Proof: by induction on n

P.1.1 base case:

The cost 1 expressions are of the form $(v \circ w)$ and $(-v)$, where v and w are variables. So the length is at most 5.

P.1.2 step case:

$\sigma(n) = \Lambda((e_1 \circ e_2)) = \Lambda(e_1) + \Lambda(e_2) + 3$, where $C(e_1) + C(e_2) \leq n - 1$.

so $\sigma(n) \leq \sigma(i) + \sigma(j) + 3 \leq 5 \cdot C(e_1) + 5 \cdot C(e_2) + 3 \leq 5 \cdot n - 1 + 5 = 5n$ □

▷ **Corollary 276** *max*$(\{\Lambda(e) \mid e \in \mathcal{E}_n\}) \leq 5 \cdot \kappa(n)$

 152 JACOBS UNIVERSITY

An Upper Bound For $\kappa(n)$-cost Expressions

▷ Idea: $e \in \mathcal{E}_n$ has at most n variables by definition.

▷ Let $\mathcal{A}_n := \{x_1, \ldots, x_n, 0, 1, *, +, -, (,)\}$, then $\#(\mathcal{A}_n) = n + 7$

▷ **Corollary 277** $\mathcal{E}_n \subseteq \bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i$ *and* $\#(\mathcal{E}_n) \leq \frac{n+7^{5\kappa(n)+1}-1}{n+7}$

▷ Proof Sketch: Note that the $\mathcal{A}_j$ are disjoint for distinct n, so

$$\#\left(\left(\bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i\right)\right) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} n + 7^i = \frac{n + 7^{5\kappa(n)+1} - 1}{n+7}$$

 153 JACOBS UNIVERSITY

Solving for $\kappa(n)$

▷ $\frac{n+7^{5\kappa(n)+1}-1}{n+7} \geq 2^{2^n}$

▷ $n + 7^{5\kappa(n)+1} \geq 2^{2^n}$ (as $n + 7^{5\kappa(n)+1} \geq \frac{n+7^{5\kappa(n)+1}-1}{n+7}$)

▷ $5\kappa(n) + 1 \cdot \log_2(n+7) \geq 2^n$ (as $\log_a(x) = \log_b(x) \cdot \log_a(b)$)

▷ $5\kappa(n) + 1 \geq \frac{2^n}{\log_2(n+7)}$

▷ $\kappa(n) \geq 1/5 \cdot \frac{2^n}{\log_2(n+7)} - 1$

▷ $\kappa(n) \in \Omega(\frac{2^n}{\log_2(n)})$

 154 JACOBS UNIVERSITY

2.5.4 The Quine-McCluskey Algorithm

After we have studied the worst-case complexity of Boolean expressions that realize given Boolean functions, let us return to the question of computing realizing Boolean expressions in practice. We will again restrict ourselves to the subclass of Boolean polynomials, but this time, we make sure that we find the optimal representatives in this class.

The first step in the endeavor of finding minimal polynomials for a given Boolean function is to optimize monomials for this task. We have two concerns here. We are interested in monomials that contribute to realizing a given Boolean function f (we say they imply f or are implicants), and we are interested in the cheapest among those that do. For the latter we have to look at a way to make monomials cheaper, and come up with the notion of a sub-monomial, i.e. a monomial that only contains a subset of literals (and is thus cheaper.)

Constructing Minimal Polynomials: Prime Implicants

▷ **Definition 278** We will use the following ordering on $\mathbb{B}$: $\mathsf{F} \leq \mathsf{T}$ (remember $0 \leq 1$)

and say that that a monomial M' dominates a monomial M, iff $f_M(c) \leq f_{M'}(c)$ for all $c \in \mathbb{B}^n$. (write $M \leq M'$)

▷ **Definition 279** A monomial M implies a Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}$ (M is an implicant of f; write $M \succ f$), iff $f_M(c) \leq f(c)$ for all $c \in \mathbb{B}^n$.

▷ **Definition 280** Let $M = L_1 \cdots L_n$ and $M' = L'_1 \cdots L'_{n'}$ be monomials, then M' is called a sub-monomial of M (write $M' \subset M$), iff $M' = 1$ or

- ▹ for all $j \leq n'$, there is an $i \leq n$, such that $L'_j = L_i$ and
- ▹ there is an $i \leq n$, such that $L_i \neq L'_j$ for all $j \leq n$

In other words: M is a sub-monomial of M', iff the literals of M are a proper subset of the literals of M'.

 155 JACOBS UNIVERSITY

With these definitions, we can convince ourselves that sub-monomials are dominated by their super-monomials. Intuitively, a monomial is a conjunction of conditions that are needed to make the Boolean function f true; if we have fewer of them, then we cannot approximate the truth-conditions of f sufficiently. So we will look for monomials that approximate f well enough and are shortest with this property: the prime implicants of f.

Constructing Minimal Polynomials: Prime Implicants

▷ ▷ **Lemma 281** *If $M' \subset M$, then M' dominates M.*

▷ Proof:

P.1 Given $c \in \mathbb{B}^n$ with $f_M(c) = \mathsf{T}$, we have, $f_{L_i}(c) = \mathsf{T}$ for all literals in M.

P.2 As M' is a sub-monomial of M, then $f_{L'_j}(c) = \mathsf{T}$ for each literal L'_j of M'.

P.3 Therefore, $f_{M'}(c) = \mathsf{T}$. □

▷ **Definition 282** An implicant M of f is a prime implicant of f iff no sub-monomial of M is an implicant of f.

 156 JACOBS UNIVERSITY

The following Theorem verifies our intuition that prime implicants are good candidates for constructing minimal polynomials for a given Boolean function. The proof is rather simple (if notationally loaded). We just assume the contrary, i.e. that there is a minimal polynomial p that contains a non-prime-implicant monomial M_k, then we can decrease the cost of the of p while still inducing the given function f. So p was not minimal which shows the assertion.

Prime Implicants and Costs

▷ **Theorem 283** *Given a Boolean function $f \neq \lambda x.\mathrm{F}$ and a Boolean polynomial $f_p \equiv f$ with minimal cost, i.e., there is no other polynomial $p' \equiv p$ such that $C(p') < C(p)$. Then, p solely consists of prime implicants of f.*

▷ Proof: The theorem obviously holds for $f = \lambda x.\mathrm{T}$.

P.1 For other f, we have $f \equiv f_p$ where $p := \sum_{i=1}^{n} M_i$ for some $n \geq 1$ monomials M_i.

P.2 Nos, suppose that M_i is not a prime implicant of f, i.e., $M' \succ f$ for some $M' \subset M_k$ with $k < i$.

P.3 Let us substitute M_k by M': $p' := \sum_{i=1}^{k-1} M_i + M' + \sum_{i=k+1}^{n} M_i$

P.4 We have $C(M') < C(M_k)$ and thus $C(p') < C(p)$ (def of sub-monomial)

P.5 Furthermore $M_k \leq M'$ and hence that $p \leq p'$ by Lemma 281.

P.6 In addition, $M' \leq p$ as $M' \succ f$ and $f = p$.

P.7 similarly: $M_i \leq p$ for all M_i. Hence, $p' \leq p$.

P.8 So $p' \equiv p$ and $f_p \equiv f$. Therefore, p is not a minimal polynomial. □

This theorem directly suggests a simple generate-and-test algorithm to construct minimal polynomials. We will however improve on this using an idea by Quine and McCluskey. There are of course better algorithms nowadays, but this one serves as a nice example of how to get from a theoretical insight to a practical algorithm.

The Quine/McCluskey Algorithm (Idea)

▷ Idea: use this theorem to search for minimal-cost polynomials

- ▷ Determine all prime implicants (sub-algorithm QMC_1)
- ▷ choose the minimal subset that covers f (sub-algorithm QMC_2)

▷ Idea: To obtain prime implicants,

- ▷ start with the DNF monomials (they are implicants by construction)
- ▷ find submonomials that are still implicants of f.

▷ Idea: Look at polynomials of the form $p := mx_i + m\overline{x_i}$ (note: $p \equiv m$)

Armed with the knowledge that minimal polynomials must consist entirely of prime implicants, we can build a practical algorithm for computing minimal polynomials: In a first step we compute the set of prime implicants of a given function, and later we see whether we actually need all of them.

For the first step we use an important observation: for a given monomial m, the polynomials $m\,x + m\,\overline{x}$ are equivalent, and in particular, we can obtain an equivalent polynomial by replace the latter (the partners) by the former (the resolvent). That gives the main idea behind the first part of the Quine-McCluskey algorithm. Given a Boolean function f, we start with a polynomial for f:

the disjunctive normal form, and then replace partners by resolvents, until that is impossible.

The algorithm QMC$_1$, for determining Prime Implicants

▷ **Definition 284** Let M be a set of monomials, then

- ▷ $\mathcal{R}(M) := \{m \mid (m\,x) \in M \wedge (m\,\overline{x}) \in M\}$ is called the set of resolvents of M
- ▷ $\widehat{\mathcal{R}}(M) := \{m \in M \mid m$ has a partner in $M\}$ ($n\,\overline{x_i}$ and $n\,x_i$ are partners)

▷ **Definition 285 (Algorithm)** Given $f \colon \mathbb{B}^n \to \mathbb{B}$

- ▷ let $M_0 :=$ DNF(f) and for all $j > 0$ compute (DNF as set of monomials)
  - ▷ $M_j := \mathcal{R}(M_{j-1})$ (resolve to get sub-monomials)
  - ▷ $P_j := M_{j-1} \backslash \widehat{\mathcal{R}}(M_{j-1})$ (get rid of redundant resolution partners)
- ▷ terminate when $M_j = \emptyset$, return $P_{\text{prime}} := \bigcup_{j=1}^{n} P_j$

We will look at a simple example to fortify our intuition.

Example for QMC$_1$

x1	x2	x3	f	monomials
F	F	F	T	$x1^0\,x2^0\,x3^0$
F	F	T	T	$x1^0\,x2^0\,x3^1$
F	T	F	F	
F	T	T	F	
T	F	F	T	$x1^1\,x2^0\,x3^0$
T	F	T	T	$x1^1\,x2^0\,x3^1$
T	T	F	F	
T	T	T	T	$x1^1\,x2^1\,x3^1$

$$P_{prime} = \bigcup_{j=1}^{3} P_j = \{x1\,x3, \overline{x2}\}$$

$$\begin{aligned}
M_0 &= \{\underbrace{\overline{x1}\,\overline{x2}\,\overline{x3}}_{=:\,e_1^0}, \underbrace{\overline{x1}\,\overline{x2}\,x3}_{=:\,e_2^0}, \underbrace{x1\,\overline{x2}\,\overline{x3}}_{=:\,e_3^0}, \underbrace{x1\,\overline{x2}\,x3}_{=:\,e_4^0}, \underbrace{x1\,x2\,x3}_{=:\,e_5^0}\} \\
M_1 &= \{\underbrace{\overline{x1}\,\overline{x2}}_{\substack{\mathcal{R}(e_1^0,e_2^0)\\ =:\,e_1^1}}, \underbrace{\overline{x2}\,\overline{x3}}_{\substack{\mathcal{R}(e_1^0,e_3^0)\\ =:\,e_2^1}}, \underbrace{\overline{x2}\,x3}_{\substack{\mathcal{R}(e_2^0,e_4^0)\\ =:\,e_3^1}}, \underbrace{x1\,\overline{x2}}_{\substack{\mathcal{R}(e_3^0,e_4^0)\\ =:\,e_4^1}}, \underbrace{x1\,x3}_{\substack{\mathcal{R}(e_4^0,e_5^0)\\ =:\,e_5^1}}\} \\
P_1 &= \emptyset \\
M_2 &= \{\underbrace{\overline{x2}}_{\mathcal{R}(e_1^1,e_4^1)}, \underbrace{\overline{x2}}_{\mathcal{R}(e_2^1,e_3^1)}\} \\
P_2 &= \{x1\,x3\} \\
M_3 &= \emptyset \\
P_3 &= \{\overline{x2}\}
\end{aligned}$$

▷ But: even though the minimal polynomial only consists of prime implicants, it need not contain all of them

We now verify that the algorithm really computes what we want: all prime implicants of the Boolean function we have given it. This involves a somewhat technical proof of the assertion below. But we are mainly interested in the direct consequences here.

Properties of QMC$_1$

▷ **Lemma 286** *(proof by simple (mutual) induction)*

1. *all monomials in M_j have exactly $n - j$ literals.*
2. *M_j contains the implicants of f with $n - j$ literals.*
3. *P_j contains the prime implicants of f with $n - j + 1$ for $j > 0$. literals*

▷ **Corollary 287** *QMC_1 terminates after at most n rounds.*

▷ **Corollary 288** *P_{prime} is the set of all prime implicants of f.*

Note that we are not finished with our task yet. We have computed all prime implicants of a given Boolean function, but some of them might be un-necessary in the minimal polynomial. So we have to determine which ones are. We will first look at the simple brute force method of finding the minimal polynomial: we just build all combinations and test whether they induce the right Boolean function. Such algorithms are usually called generate-and-test algorithms.

They are usually simplest, but not the best algorithms for a given computational problem. This is also the case here, so we will present a better algorithm below.

Algorithm QMC_2: Minimize Prime Implicants Polynomial

▷ **Definition 289 (Algorithm)** Generate and test!

- ▹ enumerate $S_p \subseteq P_{prime}$, i.e., all possible combinations of prime implicants of f,
- ▹ form a polynomial e_p as the sum over S_p and test whether $f_{e_p} = f$ and the cost of e_p is minimal

▷ **Example 290** $P_{prime} = \{x1\,x3, \overline{x2}\}$, so $e_p \in \{1, x1\,x3, \overline{x2}, x1\,x3 + \overline{x2}\}$.

▷ Only $f_{x1\,x3+\overline{x2}} \equiv f$, so $x1\,x3 + \overline{x2}$ is the minimal polynomial

▷ Complaint: The set of combinations (power set) grows exponentially

A better Mouse-trap for QMC_2: The Prime Implicant Table

▷ **Definition 291** Let $f : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, then the PIT consists of

- ▹ a left hand column with all prime implicants p_i of f
- ▹ a top row with all vectors $x \in \mathbb{B}^n$ with $f(x) = \mathsf{T}$
- ▹ a central matrix of all $f_{p_i}(x)$

▷ **Example 292**

	FFF	FFT	TFF	TFT	TTT
$x1\,x3$	F	F	F	T	T
$\overline{x2}$	T	T	T	T	F

▷ **Definition 293** A prime implicant p is essential for f iff

- ▹ there is a $c \in \mathbb{B}^n$ such that $f_p(c) = \mathsf{T}$ and
- ▹ $f_q(c) = \mathsf{F}$ for all other prime implicants q.

Note: A prime implicant is essential, iff there is a column in the PIT, where it has a T and all others have F.

Essential Prime Implicants and Minimal Polynomials

▷ ▷ **Theorem 294** *Let $f : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, p an essential prime implicant for f, and p_{min} a minimal polynomial for f, then $p \in p_{min}$.*

▷ Proof: by contradiction: let $p \notin p_{min}$

P.1 We know that $f = f_{p_{min}}$ and $p_{min} = \sum_{j=1}^{n} p_j$ for some $n \in \mathbb{N}$ and prime implicants p_j.

P.2 so for all $c \in \mathbb{B}^n$ with $f(c) = \mathsf{T}$ there is a $j \leq n$ with $f_{p_j}(c) = \mathsf{T}$.

P.3 so p cannot be essential □

  JACOBS UNIVERSITY

Let us now apply the optimized algorithm to a slightly bigger example.

A complex Example for QMC (Function and DNF)

x1	x2	x3	x4	f	monomials
F	F	F	F	T	$x1^0\, x2^0\, x3^0\, x4^0$
F	F	F	T	T	$x1^0\, x2^0\, x3^0\, x4^1$
F	F	T	F	T	$x1^0\, x2^0\, x3^1\, x4^0$
F	F	T	T	F	
F	T	F	F	F	
F	T	F	T	T	$x1^0\, x2^1\, x3^0\, x4^1$
F	T	T	F	F	
F	T	T	T	F	
T	F	F	F	F	
T	F	F	T	F	
T	F	T	F	T	$x1^1\, x2^0\, x3^1\, x4^0$
T	F	T	T	T	$x1^1\, x2^0\, x3^1\, x4^1$
T	T	F	F	F	
T	T	F	T	F	
T	T	T	F	T	$x1^1\, x2^1\, x3^1\, x4^0$
T	T	T	T	T	$x1^1\, x2^1\, x3^1\, x4^1$

  JACOBS UNIVERSITY

A complex Example for QMC (QMC_1)

$$\begin{array}{rcl}
M_0 & = & \{x1^0\, x2^0\, x3^0\, x4^0, x1^0\, x2^0\, x3^0\, x4^1, x1^0\, x2^0\, x3^1\, x4^0, \\
& & x1^0\, x2^1\, x3^0\, x4^1, x1^1\, x2^0\, x3^1\, x4^0, x1^1\, x2^0\, x3^1\, x4^1, \\
& & x1^1\, x2^1\, x3^1\, x4^0, x1^1\, x2^1\, x3^1\, x4^1\} \\
M_1 & = & \{x1^0\, x2^0\, x3^0, x1^0\, x2^0\, x4^0, x1^0\, x3^0\, x4^1, x1^1\, x2^0\, x3^1, \\
& & x1^1\, x2^1\, x3^1, x1^1\, x3^1\, x4^1, x2^0\, x3^1\, x4^0, x1^1\, x3^1\, x4^0\} \\
P_1 & = & \emptyset \\
M_2 & = & \{x1^1\, x3^1\} \\
P_2 & = & \{x1^0\, x2^0\, x3^0, x1^0\, x2^0\, x4^0, x1^0\, x3^0\, x4^1, x2^0\, x3^1\, x4^0\} \\
M_3 & = & \emptyset \\
P_3 & = & \{x1^1\, x3^1\}
\end{array}$$

$$P_{\mathsf{prime}} = \{\overline{x1}\,\overline{x2}\,\overline{x3}, \overline{x1}\,\overline{x2}\,\overline{x4}, \overline{x1}\,\overline{x3}\,x4, \overline{x2}\,x3\,\overline{x4}, x1\,x3\}$$

  JACOBS UNIVERSITY

A better Mouse-trap for QMC_1: optimizing the data structure

▷ Idea: Do the calculations directly on the DNF table

x1	x2	x3	x4	monomials
F	F	F	F	$x1^0\,x2^0\,x3^0\,x4^0$
F	F	F	T	$x1^0\,x2^0\,x3^0\,x4^1$
F	F	T	F	$x1^0\,x2^0\,x3^1\,x4^0$
F	T	F	T	$x1^0\,x2^1\,x3^0\,x4^1$
T	F	T	F	$x1^1\,x2^0\,x3^1\,x4^0$
T	F	T	T	$x1^1\,x2^0\,x3^1\,x4^1$
T	T	T	F	$x1^1\,x2^1\,x3^1\,x4^0$
T	T	T	T	$x1^1\,x2^1\,x3^1\,x4^1$

- ▷ Note: the monomials on the right hand side are only for illustration
- ▷ Idea: do the resolution directly on the left hand side
- ▷ Find rows that differ only by a single entry. (first two rows)
- ▷ resolve: replace them by one, where that entry has an X (canceled literal)
- ▷ **Example 295** ⟨F, F, F, F⟩ and ⟨F, F, F, T⟩ resolve to ⟨F, F, F, X⟩.

 167 JACOBS UNIVERSITY

A better Mouse-trap for QMC_1: optimizing the data structure

- ▷ One step resolution on the table

x1	x2	x3	x4	monomials
F	F	F	F	$x1^0\,x2^0\,x3^0\,x4^0$
F	F	F	T	$x1^0\,x2^0\,x3^0\,x4^1$
F	F	T	F	$x1^0\,x2^0\,x3^1\,x4^0$
F	T	F	T	$x1^0\,x2^1\,x3^0\,x4^1$
T	F	T	F	$x1^1\,x2^0\,x3^1\,x4^0$
T	F	T	T	$x1^1\,x2^0\,x3^1\,x4^1$
T	T	T	F	$x1^1\,x2^1\,x3^1\,x4^0$
T	T	T	T	$x1^1\,x2^1\,x3^1\,x4^1$

$\leadsto$

x1	x2	x3	x4	monomials
F	F	F	X	$x1^0\,x2^0\,x3^0$
F	F	X	F	$x1^0\,x2^0\,x4^0$
F	X	F	T	$x1^0\,x3^0\,x4^1$
T	F	T	X	$x1^1\,x2^0\,x3^1$
T	T	T	X	$x1^1\,x2^1\,x3^1$
T	X	T	T	$x1^1\,x3^1\,x4^1$
X	F	T	F	$x2^0\,x3^1\,x4^0$
T	X	T	F	$x1^1\,x3^1\,x4^0$

- ▷ Repeat the process until no more progress can be made

x1	x2	x3	x4	monomials
F	F	F	X	$x1^0\,x2^0\,x3^0$
F	F	X	F	$x1^0\,x2^0\,x4^0$
F	X	F	T	$x1^0\,x3^0\,x4^1$
T	X	T	X	$x1^1\,x3^1$
X	F	T	F	$x2^0\,x3^1\,x4^0$

- ▷ This table represents the prime implicants of f

 168 JACOBS UNIVERSITY

A complex Example for QMC (QMC_1)

- ▷ The PIT:

	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x1}\,\overline{x2}\,\overline{x3}$	T	T	F	F	F	F	F	F
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$\overline{x2}\,x3\,\overline{x4}$	F	F	T	F	T	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

- ▷ $\overline{x1}\,\overline{x2}\,\overline{x3}$ is not essential, so we are left with

	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$\overline{x2}\,x3\,\overline{x4}$	F	F	T	F	T	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

- ▷ here $\overline{x2}, x3, \overline{x4}$ is not essential, so we are left with

	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

- ▷ all the remaining ones ($\overline{x1}\,\overline{x2}\,\overline{x4}$, $\overline{x1}\,\overline{x3}\,x4$, and $x1\,x3$) are essential

▷ So, the minimal polynomial of f is $\overline{x1}\,\overline{x2}\,\overline{x4} + \overline{x1}\,\overline{x3}\,x4 + x1\,x3$.

⚠ The following section about KV-Maps was only taught until fall 2008, it is included here just for reference ⚠

2.5.5 A simpler Method for finding Minimal Polynomials

Simple Minimization: Karnaugh-Veitch Diagram

▷ The QMC algorithm is simple but tedious (not for the back of an envelope)

▷ KV-maps provide an efficient alternative for up to 6 variables

▷ **Definition 296** A Karnaugh-Veitch map (KV-map) is a rectangular table filled with truth values induced by a Boolean function. Minimal polynomials can be read of KV-maps by systematically grouping equivalent table cells into rectangular areas of size 2^k.

▷ **Example 297 (Common KV-map schemata)**

2 vars

	$\overline{A}$	A
$\overline{B}$		
B		

square
2/4-groups

3 vars

	$\overline{AB}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$				
C				

ring
2/4/8-groups

4 vars

	$\overline{AB}$	$\overline{A}B$	A
$\overline{CD}$	m_0	m_4	m
$\overline{C}D$	m_1	m_5	m
CD	m_3	m_7	m
$C\overline{D}$	m_2	m_6	m

torus
2/4/8/16-grou

▷ Note: Note that the values in are ordered, so that exactly one variable flips sign between adjacent cells (Gray Code)

KV-maps Example: $E(6, 8, 9, 10, 11, 12, 13, 14)$

Example 298

#	A	B	C	D	V
0	F	F	F	F	F
1	F	F	F	T	F
2	F	F	T	F	F
3	F	F	T	T	F
4	F	T	F	F	F
5	F	T	F	T	F
6	F	T	T	F	T
7	F	T	T	T	F
8	T	F	F	F	T
9	T	F	F	T	T
10	T	F	T	F	T
11	T	F	T	T	T
12	T	T	F	F	T
13	T	T	F	T	T
14	T	T	T	F	T
15	T	T	T	T	F

▷ The corresponding KV-map:

	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}\,\overline{D}$	F	F	T	T
$\overline{C}D$	F	F	T	T
CD	F	F	F	T
$C\overline{D}$	F	T	T	T

▷ in the red/brown group

- ▹ A does not change, so include A
- ▹ B changes, so do not include it
- ▹ C does not change, so include $\overline{C}$
- ▹ D changes, so do not include it

So the monomial is $A\,\overline{C}$

▷ in the green/brown group we have $A\,\overline{B}$

▷ in the blue group we have $B\,C\,\overline{D}$

▷ The minimal polynomial for $E(6, 8, 9, 10, 11, 12, 13, 14)$ is $A\,\overline{B} + A\,\overline{C} + B\,C\,\overline{D}$

 171 JACOBS UNIVERSITY

KV-maps Caveats

▷ groups are always rectangular of size 2^k (no crooked shapes!)

▷ a group of size 2^k induces a monomial of size $n - k$ (the bigger the better)

▷ groups can straddle vertical borders for three variables

▷ groups can straddle horizontal and vertical borders for four variables

▷ picture the the n-variable case as a n-dimensional hypercube!

 172 JACOBS UNIVERSITY

2.6 Propositional Logic

2.6.1 Boolean Expressions and Propositional Logic

We will now look at Boolean expressions from a different angle. We use them to give us a very simple model of a representation language for

- knowledge — in our context mathematics, since it is so simple, and
- argumentation — i.e. the process of deriving new knowledge from older knowledge

Still another Notation for Boolean Expressions

▷ Idea: get closer to MathTalk

- ▷ Use $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ directly (after all, we do in MathTalk)
- ▷ construct more complex names (propositions) for variables (Use ground terms of sort $\mathbb{B}$ in an ADT)

▷ **Definition 299** Let $\Sigma = \langle\mathcal{S},\mathcal{D}\rangle$ be an abstract data type, such that $\mathbb{B}\in\mathcal{S}$ and $[\neg\colon\mathbb{B}\to\mathbb{B}],[\vee\colon\mathbb{B}\times\mathbb{B}\to\mathbb{B}]\in\mathcal{D}$, then we call the set $\mathcal{T}^g_{\mathbb{B}}(\Sigma)$ of ground Σ-terms of sort $\mathbb{B}$ a formulation of Propositional Logic.

▷ We will also call this formulation Predicate Logic without Quantifiers and denote it with PLNQ.

▷ **Definition 300** Call terms in $\mathcal{T}^g_{\mathbb{B}}(\Sigma)$ without $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ atoms. (write $\mathcal{A}(\Sigma)$)

▷ Note: Formulae of propositional logic "are" Boolean Expressions

- ▷ replace $\mathbf{A}\Leftrightarrow\mathbf{B}$ by $(\mathbf{A}\Rightarrow\mathbf{B})\wedge(\mathbf{B}\Rightarrow\mathbf{A})$ and $\mathbf{A}\Rightarrow\mathbf{B}$ by $\neg\mathbf{A}\vee\mathbf{B}$...
- ▷ Build print routine $\hat{\cdot}$ with $\widehat{\mathbf{A}\wedge\mathbf{B}}=\widehat{\mathbf{A}}*\widehat{\mathbf{B}}$, and $\widehat{\neg\mathbf{A}}=\overline{\widehat{\mathbf{A}}}$ and that turns atoms into variable names. (variables and atoms are countable)

Conventions for Brackets in Propositional Logic

▷ we leave out outer brackets: $\mathbf{A}\Rightarrow\mathbf{B}$ abbreviates $(\mathbf{A}\Rightarrow\mathbf{B})$.

▷ implications are right associative: $\mathbf{A}^1\Rightarrow\cdots\Rightarrow\mathbf{A}^n\Rightarrow\mathbf{C}$ abbreviates $\mathbf{A}^1\Rightarrow(\cdots\Rightarrow(\cdots\Rightarrow(\mathbf{A}^n\Rightarrow\mathbf{C})))$

▷ a **.** stands for a left bracket whose partner is as far right as is consistent with existing brackets ($\mathbf{A}\Rightarrow.\mathbf{C}\wedge\mathbf{D}=\mathbf{A}\Rightarrow(\mathbf{C}\wedge\mathbf{D})$)

We will now use the distribution of values of a Boolean expression under all (variable) assignments to characterize them semantically. The intuition here is that we want to understand theorems, examples, counterexamples, and inconsistencies in mathematics and everyday reasoning[5].

The idea is to use the formal language of Boolean expressions as a model for mathematical language. Of course, we cannot express all of mathematics as Boolean expressions, but we can at

[5]Here (and elsewhere) we will use mathematics (and the language of mathematics) as a test tube for understanding reasoning, since mathematics has a long history of studying its own reasoning processes and assumptions.

least study the interplay of mathematical statements (which can be true or false) with the copula "and", "or" and "not".

Semantic Properties of Boolean Expressions

- **Definition 301** Let $\mathcal{M} := \langle \mathcal{U}, \mathcal{I} \rangle$ be our model, then we call e
  - true under φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{T}$ (write $\mathcal{M} \models^\varphi e$)
  - false under φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ (write $\mathcal{M} \not\models^\varphi e$)
  - satisfiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{T}$ for some assignment φ
  - valid in $\mathcal{M}$, iff $\mathcal{M} \models^\varphi e$ for all assignments φ (write $\mathcal{M} \models e$)
  - falsifiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ for some assignments φ
  - unsatisfiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ for all assignments φ
- **Example 302** $x \vee x$ is satisfiable and falsifiable.
- **Example 303** $x \vee \neg x$ is valid and $x \wedge \neg x$ is unsatisfiable.
- **Notation 304** (alternative) Write $[\![e]\!]^{\mathcal{M}}_\varphi$ for $\mathcal{I}_\varphi(e)$, if $\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$. (and $[\![e]\!]^{\mathcal{M}}$, if e is ground, and $[\![e]\!]$, if $\mathcal{M}$ is clear)
- **Definition 305 (Entailment)** (aka. logical consequence)

  We say that e entails f ($e \models f$), iff $\mathcal{I}_\varphi(f) = \mathsf{T}$ for all φ with $\mathcal{I}_\varphi(e) = \mathsf{T}$ (i.e. all assignments that make e true also make f true)

Let us now see how these semantic properties model mathematical practice.

In mathematics we are interested in assertions that are true in all circumstances. In our model of mathematics, we use variable assignments to stand for circumstances. So we are interested in Boolean expressions which are true under all variable assignments; we call them valid. We often give examples (or show situations) which make a conjectured assertion false; we call such examples counterexamples, and such assertions "falsifiable". We also often give examples for certain assertions to show that they can indeed be made true (which is not the same as being valid yet); such assertions we call "satisfiable". Finally, if an assertion cannot be made true in any circumstances we call it "unsatisfiable"; such assertions naturally arise in mathematical practice in the form of refutation proofs, where we show that an assertion (usually the negation of the theorem we want to prove) leads to an obviously unsatisfiable conclusion, showing that the negation of the theorem is unsatisfiable, and thus the theorem valid.

Example: Propositional Logic with ADT variables

- Idea: We use propositional logic to express things about the world (PLNQ $\hat{=}$ Predicate Logic without Quantifiers)
- **Example 306** Abstract Data Type: $\langle \{\mathbb{B}, \mathbb{I}\}, \{\ldots, [\mathsf{love}\colon \mathbb{I} \times \mathbb{I} \to \mathbb{B}], [\mathsf{bill}\colon \mathbb{I}], [\mathsf{mary}\colon \mathbb{I}], \ldots\} \rangle$

  ground terms:
  - $g_1 := \mathsf{love}(\mathsf{bill}, \mathsf{mary})$ (how nice)
  - $g_2 := \mathsf{love}(\mathsf{mary}, \mathsf{bill}) \wedge \neg \mathsf{love}(\mathsf{bill}, \mathsf{mary})$ (how sad)
  - $g_3 := \mathsf{love}(\mathsf{bill}, \mathsf{mary}) \wedge \mathsf{love}(\mathsf{mary}, \mathsf{john}) \Rightarrow \mathsf{hate}(\mathsf{bill}, \mathsf{john})$ (how natural)
- Semantics: by mapping into known stuff, (e.g. $\mathbb{I}$ to persons $\mathbb{B}$ to $\{\mathsf{T}, \mathsf{F}\}$)

▷ Idea: Import semantics from Boolean Algebra (atoms "are" variables)

▹ only need variable assignment $\varphi\colon \mathcal{A}(\Sigma) \to \{\mathsf{T}, \mathsf{F}\}$

▷ **Example 307** $\mathcal{I}_\varphi(\mathsf{love}(\mathsf{bill}, \mathsf{mary}) \wedge (\mathsf{love}(\mathsf{mary}, \mathsf{john}) \Rightarrow \mathsf{hate}(\mathsf{bill}, \mathsf{john}))) = \mathsf{T}$ if $\varphi(\mathsf{love}(\mathsf{bill}, \mathsf{mary})) = \mathsf{T}$, $\varphi(\mathsf{love}(\mathsf{mary}, \mathsf{john})) = \mathsf{F}$, and $\varphi(\mathsf{hate}(\mathsf{bill}, \mathsf{john})) = \mathsf{T}$

▷ **Example 308** $g_1 \wedge g_3 \wedge \mathsf{love}(\mathsf{mary}, \mathsf{john}) \models \mathsf{hate}(\mathsf{bill}, \mathsf{john})$

What is Logic?

▷ formal languages, inference and their relation with the world

▹ Formal language $\mathcal{FL}$: set of formulae ($2 + 3/7$, $\forall x.x + y = y + x$)

▹ Formula: sequence/tree of symbols ($x, y, f, g, p, 1, \pi, \in, \neg, \wedge \forall, \exists$)

▹ Models: things we understand (e.g. number theory)

▹ Interpretation: maps formulae into models ($[\![\text{three plus five}]\!] = 8$)

▹ Validity: $\mathcal{M} \models \mathbf{A}$, iff $[\![\mathbf{A}]\!]^{\mathcal{M}} = \mathsf{T}$ (five greater three is valid)

▹ Entailment: $\mathbf{A} \models \mathbf{B}$, iff $\mathcal{M} \models \mathbf{B}$ for all $\mathcal{M} \models \mathbf{A}$. (generalize to $\mathcal{H} \models \mathbf{A}$)

▹ Inference: rules to transform (sets of) formulae ($\mathbf{A}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{B}$)

▷ Syntax: formulae, inference (just a bunch of symbols)

▷ Semantics: models, interpr., validity, entailment (math. structures)

▷ Important Question: relation between syntax and semantics?

So logic is the study of formal representations of objects in the real world, and the formal statements that are true about them. The insistence on a *formal language* for representation is actually something that simplifies life for us. Formal languages are something that is actually easier to understand than e.g. natural languages. For instance it is usually decidable, whether a string is a member of a formal language. For natural language this is much more difficult: there is still no program that can reliably say whether a sentence is a grammatical sentence of the English language.

We have already discussed the meaning mappings (under the monicker "semantics"). Meaning mappings can be used in two ways, they can be used to understand a formal language, when we use a mapping into "something we already understand", or they are the mapping that legitimize a representation in a formal language. We understand a formula (a member of a formal language) $\mathbf{A}$ to be a representation of an object $\mathcal{O}$, iff $[\![\mathbf{A}]\!] = \mathcal{O}$.

However, the game of representation only becomes really interesting, if we can do something with the representations. For this, we give ourselves a set of syntactic rules of how to manipulate the formulae to reach new representations or facts about the world.

Consider, for instance, the case of calculating with numbers, a task that has changed from a difficult job for highly paid specialists in Roman times to a task that is now feasible for young children. What is the cause of this dramatic change? Of course the formalized reasoning procedures for arithmetic that we use nowadays. These *calculi* consist of a set of rules that can be followed purely syntactically, but nevertheless manipulate arithmetic expressions in a correct and fruitful way. An essential prerequisite for syntactic manipulation is that the objects are given in a formal language suitable for the problem. For example, the introduction of the decimal system has been instrumental to the simplification of arithmetic mentioned above. When the arithmetical calculi were sufficiently well-understood and in principle a mechanical procedure, and when the art of clock-making was mature enough to design and build mechanical devices of an appropriate kind, the invention of calculating machines for arithmetic by Wilhelm Schickard (1623), Blaise Pascal (1642), and Gottfried Wilhelm Leibniz (1671) was only a natural consequence.

We will see that it is not only possible to calculate with numbers, but also with representations of statements about the world (propositions). For this, we will use an extremely simple example; a fragment of propositional logic (we restrict ourselves to only one logical connective) and a small calculus that gives us a set of rules how to manipulate formulae.

A simple System: Prop. Logic with Hilbert-Calculus

- Formulae: built from prop. variables: $P, Q, R, \ldots$ and implication: $\Rightarrow$
- Semantics: $\mathcal{I}_\varphi(P) = \varphi(P)$ and $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{T}$, iff $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.
- $\mathbf{K} := P \Rightarrow Q \Rightarrow P$, $\mathbf{S} := (P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R$
- $\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}\mathrm{MP} \qquad \dfrac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\mathrm{Subst}$
- Let us look at a $\mathcal{H}^0$ theorem (with a proof)
- $\mathbf{C} \Rightarrow \mathbf{C}$ (Tertium non datur)
- Proof:

  P.1 $(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ ($\mathbf{S}$ with $[\mathbf{C}/P], [\mathbf{C} \Rightarrow \mathbf{C}/Q], [\mathbf{C}/R]$)

  P.2 $\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}$ ($\mathbf{K}$ with $[\mathbf{C}/P], [\mathbf{C} \Rightarrow \mathbf{C}/Q]$)

  P.3 $(\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (MP on P.1 and P.2)

  P.4 $\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ ($\mathbf{K}$ with $[\mathbf{C}/P], [\mathbf{C}/Q]$)

  P.5 $\mathbf{C} \Rightarrow \mathbf{C}$ (MP on P.3 and P.4)

  P.6 We have shown that $\emptyset \vdash_{\mathcal{H}^0} \mathbf{C} \Rightarrow \mathbf{C}$ (i.e. $\mathbf{C} \Rightarrow \mathbf{C}$ is a theorem) (is is also valid?)

  □

 178

This is indeed a very simple logic, that with all of the parts that are necessary:

- A formal language: expressions built up from variables and implications.
- A semantics: given by the obvious interpretation function
- A calculus: given by the two axioms and the two inference rules.

The calculus gives us a set of rules with which we can derive new formulae from old ones. The axioms are very simple rules, they allow us to derive these two formulae in any situation. The inference rules are slightly more complicated: we read the formulae above the horizontal line as assumptions and the (single) formula below as the conclusion. An inference rule allows us to derive the conclusion, if we have already derived the assumptions.

Now, we can use these inference rules to perform a proof. A proof is a sequence of formulae that can be derived from each other. The representation of the proof in the slide is slightly compactified to fit onto the slide: We will make it more explicit here. We first start out by deriving the formula

$$(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R \tag{2.1}$$

which we can always do, since we have an axiom for this formula, then we apply the rule *subst*, where $\mathbf{A}$ is this result, $\mathbf{B}$ is $\mathbf{C}$, and X is the variable P to obtain

$$(\mathbf{C} \Rightarrow Q \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow Q) \Rightarrow \mathbf{C} \Rightarrow R \tag{2.2}$$

Next we apply the rule *subst* to this where $\mathbf{B}$ is $\mathbf{C} \Rightarrow \mathbf{C}$ and X is the variable Q this time to obtain

$$(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow R \tag{2.3}$$

And again, we apply the rule *subst* this time, $\mathbf{B}$ is $\mathbf{C}$ and X is the variable R yielding the first formula in our proof on the slide. To conserve space, we have combined these three steps into one in the slide. The next steps are done in exactly the same way.

2.6.2 A digression on Names and Logics

The name MP comes from the Latin name "modus ponens" (the "mode of putting" [new facts]), this is one of the classical syllogisms discovered by the ancient Greeks. The name Subst is just short for substitution, since the rule allows to instantiate variables in formulae with arbitrary other formulae.

Digression: To understand the reason for the names of $\mathbf{K}$ and $\mathbf{S}$ we have to understand much more logic. Here is what happens in a nutshell: There is a very tight connection between types of functional languages and propositional logic (google Curry/Howard Isomorphism). The $\mathbf{K}$ and $\mathbf{S}$ axioms are the types of the K and S combinators, which are functions that can make all other functions. In SML, we have already seen the K in Example 97

```
val K = fn x => (fn y => x) : ‘a -> ‘b -> ‘a
```

Note that the type `‘a -> ‘b -> ‘a` looks like (is isomorphic under the Curry/Howard isomorphism) to our axiom $P \Rightarrow Q \Rightarrow P$. Note furthermore that K a function that takes an argument n and returns a constant function (the function that returns n on all arguments). Now the German name for "constant function" is "Konstante Function", so you have letter K in the name. For the $\mathbf{S}$ aiom (which I do not know the naming of) you have

```
val S = fn x => (fn y => (fn z => x z (y z))) : (‘a -> ‘b -> ‘c) - (‘a -> ‘c) -> ‘a -> ‘c
```

Now, you can convince yourself that $SKKx = x = Ix$ (i.e. the function S applied to two copies of K is the identity combinator I). Note that

```
val I = x => x : ‘a -> ‘a
```

where the type of the identity looks like the theorem $\mathbf{C} \Rightarrow \mathbf{C}$ we proved. Moreover, under the Curry/Howard Isomorphism, proofs correspond to functions (axioms to combinators), and SKK is the function that corresponds to the proof we looked at in class.

We will now generalize what we have seen in the example so that we can talk about calculi and proofs in other situations and see what was specific to the example.

2.6.3 Logical Systems and Calculi

Calculi: general

- A calculus is a systems of inference rules: $\frac{\mathbf{A}_1 \quad \cdots}{\mathbf{A}_n} \mathbf{C} R$ and $\frac{}{\mathbf{A}} Ax$

  $\mathbf{A}_1$: assumptions, $\mathbf{C}$: conclusion (axioms have no assumptions)

- A Proof of $\mathbf{A}$ from hypotheses in $\mathcal{H}$ ($\mathcal{H} \vdash \mathbf{A}$) is a tree, such that its
  - nodes contain inference rules
  - leaves contain formulae from $\mathcal{H}$
  - root contains $\mathbf{A}$

$$\dfrac{\dfrac{}{\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}}\mathsf{Ax} \quad \mathbf{A}}{\mathbf{B} \Rightarrow \mathbf{A}} \Rightarrow E$$

▷ **Example 309** $\mathbf{A} \vdash \mathbf{B} \Rightarrow \mathbf{A}$

 179 JACOBS UNIVERSITY

Derivations and Proofs

▷ **Definition 310** A derivation of a formula $\mathbf{C}$ from a set $\mathcal{H}$ of hypotheses (write $\mathcal{H} \vdash \mathbf{C}$) is a sequence $\mathbf{A}_1, \ldots, \mathbf{A}_m$ of formulae, such that

- ▷ $\mathbf{A}_m = \mathbf{C}$ (derivation culminates in $\mathbf{C}$)
- ▷ for all $(1 \leq i \leq m)$, either $\mathbf{A}_i \in \mathcal{H}$ (hypothesis) or there is an inference rule $\dfrac{\mathbf{A}_{l_1} \quad \cdots \quad \mathbf{A}_{l_k}}{\mathbf{A}_i}$, where $l_j < i$ for all $j \leq k$.

$$\dfrac{\dfrac{}{\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}}\mathsf{Ax} \quad \mathbf{A}}{\mathbf{B} \Rightarrow \mathbf{A}} \Rightarrow E$$

▷ **Example 311** In the propositional calculus of natural deduction we have $\mathbf{A} \vdash \mathbf{B} \Rightarrow \mathbf{A}$: the sequence is $\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}, \mathbf{A}, \mathbf{B} \Rightarrow \mathbf{A}$

▷ **Observation 312** *Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then the $\mathcal{C}$ derivation relation defined in Definition 310 is a derivation system in the sense of ??*

▷ **Definition 313** A derivation $\emptyset \vdash_{\mathcal{C}} \mathbf{A}$ is called a proof of $\mathbf{A}$ and if one exists ($\vdash_{\mathcal{C}} \mathbf{A}$) then $\mathbf{A}$ is called a $\mathcal{C}$-theorem.

▷ **Definition 314** an inference rule $\mathcal{I}$ is called admissible in $\mathcal{C}$, if the extension of $\mathcal{C}$ by $\mathcal{I}$ does not yield new theorems.

 180 JACOBS UNIVERSITY

With formula schemata we mean representations of sets of formulae. In our example above, we used uppercase boldface letters as (meta)-variables for formulae. For instance, the the "modus ponens" inference rule stands for[9] EdNote:9

As an axiom does not have assumptions, it can be added to a proof at any time. This is just what we did with the axioms in our example proof.

In general formulae can be used to represent facts about the world as propositions; they have a semantics that is a mapping of formulae into the real world (propositions are mapped to truth values.) We have seen two relations on formulae: the entailment relation and the deduction relation. The first one is defined purely in terms of the semantics, the second one is given by a calculus, i.e. purely syntactically. Is there any relation between these relations?

Ideally, both relations would be the same, then the calculus would allow us to infer all facts that can be represented in the given formal language and that are true in the real world, and only those. In other words, our representation and inference is faithful to the world.

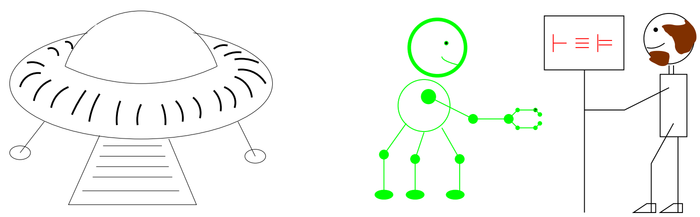
A consequence of this is that we can rely on purely syntactical means to make predictions about the world. Computers rely on formal representations of the world; if we want to solve a problem on our computer, we first represent it in the computer (as data structures, which can be seen as a formal language) and do syntactic manipulations on these structures (a form of calculus). Now, if the provability relation induced by the calculus and the validity relation coincide (this will

[9] EdNote: continue

be quite difficult to establish in general), then the solutions of the program will be correct, and we will find all possible ones.

Properties of Calculi (Theoretical Logic)

- Correctness: (provable implies valid)
  - $\mathcal{H} \vdash \mathbf{B}$ implies $\mathcal{H} \models \mathbf{B}$ (equivalent: $\vdash \mathbf{A}$ implies $\models \mathbf{A}$)
- Completeness: (valid implies provable)
  - $\mathcal{H} \models \mathbf{B}$ implies $\mathcal{H} \vdash \mathbf{B}$ (equivalent: $\models \mathbf{A}$ implies $\vdash \mathbf{A}$)
- Goal: $\vdash \mathbf{A}$ iff $\models \mathbf{A}$ (provability and validity coincide)
  - To TRUTH through PROOF (CALCULEMUS [Leibniz ∼1680])



 181 JACOBS UNIVERSITY

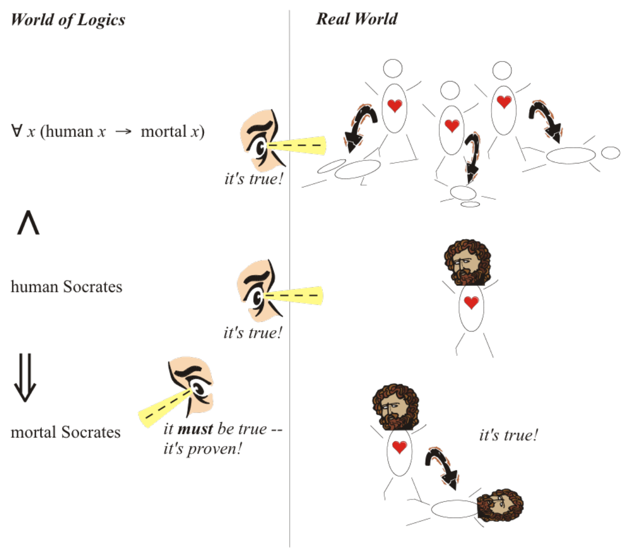
Of course, the logics we have studied so far are very simple, and not able to express interesting facts about the world, but we will study them as a simple example of the fundamental problem of Computer Science: How do the formal representations correlate with the real world.

Within the world of logics, one can derive new propositions (the *conclusions*, here: *Socrates is mortal*) from given ones (the *premises*, here: *Every human is mortal* and *Sokrates is human*). Such derivations are *proofs*.

Logics can describe the internal structure of real-life facts; e.g. individual things, actions, properties. A famous example, which is in fact as old as it appears, is illustrated in the slide below.

If a logic is correct, the conclusions one can prove are true (= hold in the real world) whenever the premises are true. This is a miraculous fact (think about it!)

The miracle of logics



Purely formal derivations are true in the real world!

2.6.4 Proof Theory for the Hilbert Calculus

We now show one of the meta-properties (soundness) for the Hilbert calculus $\mathcal{H}^0$. The statement of the result is rather simple: it just says that the set of provable formulae is a subset of the set of valid formulae. In other words: If a formula is provable, then it must be valid (a rather comforting property for a calculus).

$\mathcal{H}^0$ is sound (first version)

- ▷ **Theorem 315** $\vdash \mathbf{A}$ *implies* $\models \mathbf{A}$ *for all propositions* $\mathbf{A}$.
- ▷ Proof: show by induction over proof length

P.1 Axioms are valid (we already know how to do this!)

P.2 inference rules preserve validity (let's think)

P.2.1 Subst:

complicated, see next slide

P.2.2 MP:

P.2.2.1 Let $\mathbf{A} \Rightarrow \mathbf{B}$ be valid, and $\varphi\colon \mathcal{V}_o \to \{\mathsf{T}, \mathsf{F}\}$ arbitrary

P.2.2.2 then $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$ (by definition of $\Rightarrow$).

P.2.2.3 Since $\mathbf{A}$ is valid, $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{T} \neq \mathsf{F}$, so $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.

P.2.2.4 As φ was arbitrary, $\mathbf{B}$ is valid. □

□

To complete the proof, we have to prove two more things. The first one is that the axioms are valid. Fortunately, we know how to do this: we just have to show that under all assignments, the axioms are satisfied. The simplest way to do this is just to use truth tables.

$\mathcal{H}^0$ axioms are valid

- ▷ **Lemma 316** *The $\mathcal{H}^0$ axioms are valid.*
- ▷ Proof: We simply check the truth tables

P.1

P	Q	$Q \Rightarrow P$	$P \Rightarrow Q \Rightarrow P$
F	F	T	T
F	T	F	T
T	F	T	T
T	T	T	T

P.2

P	Q	R	$\mathbf{A} := P \Rightarrow Q \Rightarrow R$	$\mathbf{B} := P \Rightarrow Q$	$\mathbf{C} := P \Rightarrow R$	$\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}$
F	F	F	T	T	T	T
F	F	T	T	T	T	T
F	T	F	T	T	T	T
F	T	T	T	T	T	T
T	F	F	T	F	F	T
T	F	T	T	F	T	T
T	T	F	F	T	F	T
T	T	T	T	T	T	T

□

 184 JACOBS UNIVERSITY

The next result encapsulates the soundness result for the substitution rule, which we still owe. We will prove the result by induction on the structure of the formula that is instantiated. To get the induction to go through, we not only show that validity is preserved under instantiation, but we make a concrete statement about the value itself.

A proof by induction on the structure of the formula is something we have not seen before. It can be justified by a normal induction over natural numbers; we just take property of a natural number n to be that all formulae with n symbols have the property asserted by the theorem. The only thing we need to realize is that proper subterms have strictly less symbols than the terms themselves.

Substitution Value Lemma and Soundness

▷ **Lemma 317** *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be formulae, then* $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\psi(\mathbf{A})$, *where* $\psi = \varphi, [\mathcal{I}_\varphi(\mathbf{B})/X]$

▷ Proof: by induction on the depth of $\mathbf{A}$ (number of nested $\Rightarrow$ symbols)

P.1 We have to consider two cases

P.1.1 depth=0, then $\mathbf{A}$ is a variable, say Y.:

P.1.1.1 We have two cases

P.1.1.1.1 $X = Y$:

then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\varphi([\mathbf{B}/X](X)) = \mathcal{I}_\varphi(\mathbf{B}) = \psi(X) = \mathcal{I}_\psi(X) = \mathcal{I}_\psi(\mathbf{A})$.

P.1.1.1.2 $X \neq Y$:

then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\varphi([\mathbf{B}/X](Y)) = \mathcal{I}_\varphi(Y) = \varphi(Y) = \psi(Y) = \mathcal{I}_\psi(Y) = \mathcal{I}_\psi(\mathbf{A})$.

P.1.2 depth> 0, then $\mathbf{A} = \mathbf{C} \Rightarrow \mathbf{D}$:

P.1.2.1 We have $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathsf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{C})) = \mathsf{F}$ or $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{D})) = \mathsf{T}$.

P.1.2.2 This is the case, iff $\mathcal{I}_\psi(\mathbf{C}) = \mathsf{F}$ or $\mathcal{I}_\psi(\mathbf{D}) = \mathsf{T}$ by IH ($\mathbf{C}$ and $\mathbf{D}$ have smaller depth than $\mathbf{A}$).

P.1.2.3 In other words, $\mathcal{I}_\psi(\mathbf{A}) = \mathcal{I}_\psi(\mathbf{C} \Rightarrow \mathbf{D}) = \mathsf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathsf{T}$ by definition. □

P.2 We have considered all the cases and proven the assertion. □

 185 JACOBS UNIVERSITY

Armed with the substitution value lemma, it is quite simple to establish the soundness of the substitution rule. We state the assertion rather succinctly: "Subst preservers validity", which means that if the assumption of the Subst rule was valid, then the conclusion is valid as well, i.e. the validity property is preserved.

Soundness of Substitution

▷ **Lemma 318** *Subst preserves validity.*

▷ Proof: We have to show that $[\mathbf{B}/X](\mathbf{A})$ is valid, if $\mathbf{A}$ is.

P.1 Let $\mathbf{A}$ be valid, $\mathbf{B}$ a formula, $\varphi\colon \mathcal{V}_o \to \{\mathsf{T},\mathsf{F}\}$ a variable assignment, and $\psi := \varphi,[\mathcal{I}_\varphi(\mathbf{B})/X]$.

P.2 then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_{\varphi,[\mathcal{I}_\varphi(\mathbf{B})/X]}(\mathbf{A}) = \mathsf{T}$, since $\mathbf{A}$ is valid.

P.3 As the argumentation did not depend on the choice of φ, $[\mathbf{B}/X](\mathbf{A})$ valid and we have proven the assertion. □

The next theorem shows that the implication connective and the entailment relation are closely related: we can move a hypothesis of the entailment relation into an implication assumption in the conclusion of the entailment relation. Note that however close the relationship between implication and entailment, the two should not be confused. The implication connective is a syntactic formula constructor, whereas the entailment relation lives in the semantic realm. It is a relation between formulae that is induced by the evaluation mapping.

The Entailment Theorem

▷ **Theorem 319** *If* $\mathcal{H},\mathbf{A} \models \mathbf{B}$*, then* $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.

▷ Proof: We show that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{T}$ for all assignments φ with $\mathcal{I}_\varphi(\mathcal{H}) = \mathsf{T}$ whenever $\mathcal{H},\mathbf{A} \models \mathbf{B}$

P.1 Let us assume there is an assignment φ, such that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{F}$.

P.2 Then $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{T}$ and $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{F}$ by definition.

P.3 But we also know that $\mathcal{I}_\varphi(\mathcal{H}) = \mathsf{T}$ and thus $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$, since $\mathcal{H},\mathbf{A} \models \mathbf{B}$.

P.4 This contradicts our assumption $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$ from above.

P.5 So there cannot be an assignment φ that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{F}$; in other words, $\mathbf{A} \Rightarrow \mathbf{B}$ is valid. □

Now, we complete the theorem by proving the converse direction, which is rather simple.

The Entailment Theorem (continued)

▷ **Corollary 320** $\mathcal{H},\mathbf{A} \models \mathbf{B}$*, iff* $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$

▷ Proof: In the light of the previous result, we only need to prove that $\mathcal{H},\mathbf{A} \models \mathbf{B}$, whenever $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$

P.1 To prove that $\mathcal{H},\mathbf{A} \models \mathbf{B}$ we assume that $\mathcal{I}_\varphi(\mathcal{H},\mathbf{A}) = \mathsf{T}$.

P.2 In particular, $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{T}$ since $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.

P.3 Thus we have $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.

P.4 The first cannot hold, so the second does, thus $\mathcal{H},\mathbf{A} \models \mathbf{B}$. □

The entailment theorem has a syntactic counterpart for some calculi. This result shows a close connection between the derivability relation and the implication connective. Again, the two should not be confused, even though this time, both are syntactic.

The main idea in the following proof is to generalize the inductive hypothesis from proving $\mathbf{A} \Rightarrow \mathbf{B}$ to proving $\mathbf{A} \Rightarrow \mathbf{C}$, where $\mathbf{C}$ is a step in the proof of $\mathbf{B}$. The assertion is a special case then, since $\mathcal{B}$ is the last step in the proof of $\mathbf{B}$.

The Deduction Theorem

▷ **Theorem 321** *If* $\mathcal{H}, \mathbf{A} \vdash \mathbf{B}$, *then* $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$

▷ Proof: By induction on the proof length

P.1 Let $\mathbf{C}_1, \ldots, \mathbf{C}_m$ be a proof of $\mathbf{B}$ from the hypotheses $\mathcal{H}$.

P.2 We generalize the induction hypothesis: For all I $(1 \leq i \leq m)$ we construct proofs $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$. (get $\mathbf{A} \Rightarrow \mathbf{B}$ for $i = m$)

P.3 We have to consider three cases

P.3.1 Case 1: $\mathbf{C}_i$ axiom or $\mathbf{C}_i \in \mathcal{H}$:

P.3.1.1 Then $\mathcal{H} \vdash \mathbf{C}_i$ by construction and $\mathcal{H} \vdash \mathbf{C}_i \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Subst from Axiom 1.

P.3.1.2 So $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP. □

P.3.2 Case 2: $\mathbf{C}_i = \mathbf{A}$:

P.3.2.1 We have already proven $\emptyset \vdash \mathbf{A} \Rightarrow \mathbf{A}$, so in particular $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$. (more hypotheses do not hurt)

□

P.3.3 Case 3: everything else:

P.3.3.1 $\mathbf{C}_i$ is inferred by MP from $\mathbf{C}_j$ and $\mathbf{C}_k = \mathbf{C}_j \Rightarrow \mathbf{C}_i$ for $j, k < i$

P.3.3.2 We have $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j$ and $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i$ by IH

P.3.3.3 Furthermore, $(\mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}_j) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Axiom 2 and Subst

P.3.3.4 and thus $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP (twice). □

P.4 We have treated all cases, and thus proven $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ for $(1 \leq i \leq m)$.

P.5 Note that $\mathbf{C}_m = \mathbf{B}$, so we have in particular proven $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$. □

In fact (you have probably already spotted this), this proof is not correct. We did not cover all cases: there are proofs that end in an application of the Subst rule. This is a common situation, we think we have a very elegant and convincing proof, but upon a closer look it turns out that there is a gap, which we still have to bridge.

This is what we attempt to do now. The first attempt to prove the subst case below seems to work at first, until we notice that the substitution $[\mathbf{B}/X]$ would have to be applied to $\mathbf{A}$ as well, which ruins our assertion.

The missing Subst case

▷ Oooops: The proof of the deduction theorem was incomplete (we did not treat the Subst case)

▷ Let's try:

▷ Proof: $\mathbf{C}_i$ is inferred by Subst from $\mathbf{C}_j$ for $j < i$ with $[\mathbf{B}/X]$.

P.1 So $\mathbf{C}_i = [\mathbf{B}/X](\mathbf{C}_j)$; we have $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j$ by IH

P.2 so by Subst we have $\mathcal{H} \vdash [\mathbf{B}/X](\mathbf{A} \Rightarrow \mathbf{C}_j)$. (Oooops! $\neq \mathbf{A} \Rightarrow \mathbf{C}_i$)

□

In this situation, we have to do something drastic, like come up with a totally different proof. Instead we just prove the theorem we have been after for a variant calculus.

Repairing the Subst case by repairing the calculus

▷ Idea: Apply Subst only to axioms (this was sufficient in our example)

▷ $\mathcal{H}^1$ Axiom Schemata: (infinitely many axioms)

$\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}$, $\quad (\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}$
Only one inference rule: MP.

▷ **Definition 322** $\mathcal{H}^1$ introduces a (potentially) different derivability relation than $\mathcal{H}^0$ we call them $\vdash_{\mathcal{H}^0}$ and $\vdash_{\mathcal{H}^1}$

Now that we have made all the mistakes, let us write the proof in its final form.

Deduction Theorem Redone

▷ **Theorem 323** *If* $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^1} \mathbf{B}$, *then* $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{B}$

▷ Proof: Let $\mathbf{C}_1, \ldots, \mathbf{C}_m$ be a proof of $\mathbf{B}$ from the hypotheses $\mathcal{H}$.

P.1 We construct proofs $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ for all $(1 \leq i \leq n)$ by induction on i.

P.2 We have to consider three cases

P.2.1 $\mathbf{C}_i$ is an axiom or hypothesis:

P.2.1.1 Then $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i$ by construction and $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Ax1.

P.2.1.2 So $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i$ by MP □

P.2.2 $\mathbf{C}_i = \mathbf{A}$:

P.2.2.1 We have proven $\emptyset \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{A}$, (check proof in $\mathcal{H}^1$)
We have $\emptyset \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$, so in particular $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ □

P.2.3 else:

P.2.3.1 $\mathbf{C}_i$ is inferred by MP from $\mathbf{C}_j$ and $\mathbf{C}_k = \mathbf{C}_j \Rightarrow \mathbf{C}_i$ for $j, k < i$

P.2.3.2 We have $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j$ and $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i$ by IH

P.2.3.3 Furthermore, $(\mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}_j) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Axiom 2

P.2.3.4 and thus $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP (twice). (no Subst)

□

□

The deduction theorem and the entailment theorem together allow us to understand the claim that the two formulations of soundness ($\mathbf{A} \vdash \mathbf{B}$ implies $\mathbf{A} \models \mathbf{B}$ and $\vdash \mathbf{A}$ implies $\models \mathbf{B}$) are equivalent. Indeed, if we have $\mathbf{A} \vdash \mathbf{B}$, then by the deduction theorem $\vdash \mathbf{A} \Rightarrow \mathbf{B}$, and thus $\models \mathbf{A} \Rightarrow \mathbf{B}$ by soundness, which gives us $\mathbf{A} \models \mathbf{B}$ by the entailment theorem. The other direction and the argument for the corresponding statement about completeness are similar.

Of course this is still not the version of the proof we originally wanted, since it talks about the Hilbert Calculus $\mathcal{H}^1$, but we can show that $\mathcal{H}^1$ and $\mathcal{H}^0$ are equivalent.

But as we will see, the derivability relations induced by the two caluli are the same. So we can prove the original theorem after all.

The Deduction Theorem for $\mathcal{H}^0$

▷ **Lemma 324** $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$

▷ Proof:

P.1 All $\mathcal{H}^1$ axioms are $\mathcal{H}^0$ theorems. (by Subst)

P.2 For the other direction, we need a proof transformation argument:

P.3 We can replace an application of MP followed by Subst by two Subst applications followed by one MP.

P.4 $\ldots \mathbf{A} \Rightarrow \mathbf{B} \ldots \mathbf{A} \ldots \mathbf{B} \ldots [\mathbf{C}/X](\mathbf{B}) \ldots$ is replaced by

$$\ldots \mathbf{A} \Rightarrow \mathbf{B} \ldots [\mathbf{C}/X](\mathbf{A}) \Rightarrow [\mathbf{C}/X](\mathbf{B}) \ldots \mathbf{A} \ldots [\mathbf{C}/X](\mathbf{A}) \ldots [\mathbf{C}/X](\mathbf{B}) \ldots$$

P.5 Thus we can push later Subst applications to the axioms, transforming a $\mathcal{H}^0$ proof into a $\mathcal{H}^1$ proof. □

▷ **Corollary 325** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, *iff* $\mathcal{H} \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

▷ Proof Sketch: by MP and $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$

We can now collect all the pieces and give the full statement of the soundness theorem for $\mathcal{H}^0$

$\mathcal{H}^0$ is sound (full version)

▷ **Theorem 326** *For all propositions* $\mathbf{A}$, $\mathbf{B}$, *we have* $\mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$ *implies* $\mathbf{A} \models \mathbf{B}$.

▷ Proof:

P.1 By deduction theorem $\mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, iff $\vdash \mathbf{A} \Rightarrow \mathbf{C}$,

P.2 by the first soundness theorem this is the case, iff $\models \mathbf{A} \Rightarrow \mathbf{B}$,

P.3 by the entailment theorem this holds, iff $\mathbf{A} \models \mathbf{C}$. □

2.6.5 A Calculus for Mathtalk

In our introduction to Subsection 2.6.0 we have positioned Boolean expressions (and proposition logic) as a system for understanding the mathematical language "mathtalk" introduced in Subsection 2.2.1. We have been using this language to state properties of objects and prove them all through this course without making the rules the govern this activity fully explicit. We will rectify this now: First we give a calculus that tries to mimic the the informal rules mathematicians use int their proofs, and second we show how to extend this "calculus of natural deduction" to the full language of "mathtalk".

We will now introduce the "natural deduction" calculus for propositional logic. The calculus was created in order to model the natural mode of reasoning e.g. in everyday mathematical practice. This calculus was intended as a counter-approach to the well-known Hilbert style calculi, which were mainly used as theoretical devices for studying reasoning in principle, not for modeling particular reasoning styles.

Rather than using a minimal set of inference rules, the natural deduction calculus provides two/three inference rules for every connective and quantifier, one "introduction rule" (an inference rule that derives a formula with that symbol at the head) and one "elimination rule" (an inference rule that acts on a formula with this head and derives a set of subformulae).

Calculi: Natural Deduction ($\mathcal{ND}^0$) [Gentzen'30]

- ▷ Idea: $\mathcal{ND}^0$ tries to mimic human theorem proving behavior (non- minimal)
- ▷ **Definition 327** The $\mathcal{ND}^0$ calculus has rules for the introduction and elimination of connectives

Introduction	Elimination	Axiom
$\dfrac{\mathbf{A} \quad \mathbf{B}}{\mathbf{A} \wedge \mathbf{B}} \wedge I$	$\dfrac{\mathbf{A} \wedge \mathbf{B}}{\mathbf{A}} \wedge E_l \qquad \dfrac{\mathbf{A} \wedge \mathbf{B}}{\mathbf{B}} \wedge E_r$	$\dfrac{}{\mathbf{A} \vee \neg \mathbf{A}} \mathrm{TND}$
$\dfrac{\begin{array}{c}[\mathbf{A}]^1 \\ \hline\hline \mathbf{B}\end{array}}{\mathbf{A} \Rightarrow \mathbf{B}} \Rightarrow I^1$	$\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}} \Rightarrow E$	

- ▷ TND is used only in classical logic (otherwise constructive/intuitionistic)

The most characteristic rule in the natural deduction calculus is the $\Rightarrow I$ rule. It corresponds to the mathematical way of proving an implication $\mathbf{A} \Rightarrow \mathbf{B}$: We assume that $\mathbf{A}$ is true and show $\mathbf{B}$ from this assumption. When we can do this we discharge (get rid of) the assumption and conclude $\mathbf{A} \Rightarrow \mathbf{B}$. This mode of reasoning is called hypothetical reasoning. Note that the local hypothesis is discharged by the rule $\Rightarrow I$, i.e. it cannot be used in any other part of the proof. As the $\Rightarrow I$ rules may be nested, we decorate both the rule and the corresponding assumption with a marker (here the number 1).

Let us now consider an example of hypothetical reasoning in action.

Natural Deduction: Examples

▷ Inference with local hypotheses

$$\cfrac{\cfrac{\cfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{B}}\wedge E_r \qquad \cfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{A}}\wedge E_l}{\mathbf{B}\wedge\mathbf{A}}\wedge I}{\mathbf{A}\wedge\mathbf{B}\Rightarrow\mathbf{B}\wedge\mathbf{A}}\Rightarrow I^1$$

$$\begin{array}{c}[A]^1\\ [B]^2\\ \cfrac{\cfrac{A}{B\Rightarrow A}\Rightarrow I^2}{A\Rightarrow B\Rightarrow A}\Rightarrow I^1\end{array}$$

Another characteristic of the natural deduction calculus is that it has inference rules (introduction and elimination rules) for all connectives. So we extend the set of rules from Definition 327 for disjunction, negation and falsity.

More Rules for Natural Deduction

▷ **Definition 328** ND^0 has the following additional rules for the remaining connectives.

$$\frac{\mathbf{A}}{\mathbf{A}\vee\mathbf{B}}\vee I_l \qquad \frac{\mathbf{B}}{\mathbf{A}\vee\mathbf{B}}\vee I_r \qquad \frac{\mathbf{A}\vee\mathbf{B} \quad \begin{array}{c}[\mathbf{A}]^1\\ \vdots\\ \mathbf{C}\end{array} \quad \begin{array}{c}[\mathbf{B}]^1\\ \vdots\\ \mathbf{C}\end{array}}{\mathbf{C}}\vee E^1$$

$$\frac{\begin{array}{c}[\mathbf{A}]^1\\ \vdots\\ F\end{array}}{\neg\mathbf{A}}\neg I^1 \qquad \frac{\neg\neg\mathbf{A}}{\mathbf{A}}\neg E$$

$$\frac{\neg\mathbf{A} \quad \mathbf{A}}{F}FI \qquad \frac{F}{\mathbf{A}}FE$$

The next step now is to extend the language of propositional logic to include the quantifiers $\forall$ and $\exists$. To do this, we will extend the language PLNQ with formulae of the form $\forall x.\mathbf{A}$ and $\exists x.\mathbf{A}$, where x is a variable and $\mathbf{A}$ is a formula. This system (which ist a little more involved than we make believe now) is called "first-order logic".[10] EdNote:10

Building on the calculus ND^0, we define a first-order calculus for "mathtalk" by providing introduction and elimination rules for the quantifiers.

First-Order Natural Deduction

▷ Rules for propositional connectives just as always

▷ **Definition 329 (New Quantifier Rules)** The $\mathcal{ND}$ extends ND^0 by the following four rules

[10] EdNote: give a forward reference

$$\frac{\mathbf{A}}{\forall X.\mathbf{A}}\forall I^* \qquad \frac{\forall X.\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\forall E$$

$$\frac{[\mathbf{B}/X](\mathbf{A})}{\exists X.\mathbf{A}}\exists I \qquad \frac{\exists X.\mathbf{A} \quad \begin{array}{c}[[c/X](\mathbf{A})]^1\\ \vdots\\ \mathbf{C}\end{array}}{\mathbf{C}}\exists E^1$$

* means that $\mathbf{A}$ does not depend on any hypothesis in which X is free.

The intuition behind the rule $\forall I$ is that a formula $\mathbf{A}$ with a (free) variable X can be generalized to $\forall X.\mathbf{A}$, if X stands for an arbitrary object, i.e. there are no restricting assumptions about X. The $\forall E$ rule is just a substitution rule that allows to instantiate arbitrary terms $\mathbf{B}$ for X in $\mathbf{A}$. The $\exists I$ rule says if we have a witness $\mathbf{B}$ for X in $\mathbf{A}$ (i.e. a concrete term $\mathbf{B}$ that makes $\mathbf{A}$ true), then we can existentially close $\mathbf{A}$. The $\exists E$ rule corresponds to the common mathematical practice, where we give objects we know exist a new name c and continue the proof by reasoning about this concrete object c. Anything we can prove from the assumption $[c/X](\mathbf{A})$ we can prove outright if $\exists X.\mathbf{A}$ is known.

With the $\mathcal{ND}$ calculus we have given a set of inference rules that are (empirically) complete for all the proof we need for the General Computer Science courses. Indeed Mathematicians are convinced that (if pressed hard enough) they could transform all (informal but rigorous) proofs into (formal) $\mathcal{ND}$ proofs. This is however seldom done in practice because it is extremely tedious, and mathematicians are sure that peer review of mathematical proofs will catch all relevant errors.

In some areas however, this quality standard is not safe enough, e.g. for programs that control nuclear power plants. The field of "Formal Methods" which is at the intersection of mathematics and Computer Science studies how the behavior of programs can be specified formally in special logics and how fully formal proofs of safety properties of programs can be developed semi-automatically. Note that given the discussion in Subsection 2.6.2 fully formal proofs (in sound calculi) can be that can be checked by machines since their soundness only depends on the form of the formulae in them.

2.7 Machine-Oriented Calculi

Now we have studied the Hilbert-style calculus in some detail, let us look at two calculi that work via a totally different principle. Instead of deducing new formulae from axioms (and hypotheses) and hoping to arrive at the desired theorem, we try to deduce a contradiction from the negation of the theorem. Indeed, a formula $\mathbf{A}$ is valid, iff $\neg\mathbf{A}$ is unsatisfiable, so if we derive a contradiction from $\neg\mathbf{A}$, then we have proven $\mathbf{A}$. The advantage of such "test-calculi" (also called negative calculi) is easy to see. Instead of finding a proof that ends in $\mathbf{A}$, we have to find any of a broad class of contradictions. This makes the calculi that we will discuss now easier to control and therefore more suited for mechanization.

2.7.1 Calculi for Automated Theorem Proving: Analytical Tableaux

Analytical Tableaux

Before we can start, we will need to recap some nomenclature on formulae.

Recap: Atoms and Literals

- ▷ **Definition 330** We call a formula atomic, or an atom, iff it does not contain connectives. We call a formula complex, iff it is not atomic.
- ▷ **Definition 331** We call a pair $\mathbf{A}^\alpha$ a labeled formula, if $\alpha \in \{\mathsf{T}, \mathsf{F}\}$. A labeled atom is called literal.
- ▷ **Definition 332** Let Φ be a set of formulae, then we use $\Phi^\alpha := \{\mathbf{A}^\alpha \mid \mathbf{A} \in \Phi\}$.

The idea about literals is that they are atoms (the simplest formulae) that carry around their intended truth value.

Now we will also review some propositional identities that will be useful later on. Some of them we have already seen, and some are new. All of them can be proven by simple truth table arguments.

Test Calculi: Tableaux and Model Generation

- ▷ Idea: instead of showing $\emptyset \vdash Th$, show $\neg Th \vdash trouble$ (use $\bot$ for trouble)
- ▷ **Example 333** Tableau Calculi try to construct models.

Tableau Refutation (Validity)	Model generation (Satisfiability)
$\models P \wedge Q \Rightarrow Q \wedge P$	$\models P \wedge (Q \vee \neg R) \wedge \neg Q$
$P \wedge Q \Rightarrow Q \wedge P^{\mathsf{F}}$ $P \wedge Q^{\mathsf{T}}$ $Q \wedge P^{\mathsf{F}}$ P^{T} Q^{T} $P^{\mathsf{F}} \mid Q^{\mathsf{F}}$ $\bot \mid \bot$	$P \wedge (Q \vee \neg R) \wedge \neg Q^{\mathsf{T}}$ $P \wedge (Q \vee \neg R)^{\mathsf{T}}$ $\neg Q^{\mathsf{T}}$ Q^{F} P^{T} $Q \vee \neg R^{\mathsf{T}}$ $Q^{\mathsf{T}} \mid \neg R^{\mathsf{T}}$ $\bot \mid R^{\mathsf{F}}$
No Model	Herbrand Model $\{P^{\mathsf{T}}, Q^{\mathsf{F}}, R^{\mathsf{F}}\}$ $\varphi := \{P \mapsto \mathsf{T}, Q \mapsto \mathsf{F}, R \mapsto \mathsf{F}\}$

Algorithm: Fully expand all possible tableaux, (no rule can be applied)

▷ ▹ Satisfiable, iff there are open branches (correspond to models)

Tableau calculi develop a formula in a tree-shaped arrangement that represents a case analysis on when a formula can be made true (or false). Therefore the formulae are decorated with exponents that hold the intended truth value.

On the left we have a refutation tableau that analyzes a negated formula (it is decorated with the intended truth value F). Both branches contain an elementary contradiction $\bot$.

On the right we have a model generation tableau, which analyzes a positive formula (it is decorated with the intended truth value T. This tableau uses the same rules as the refutation tableau, but makes a case analysis of when this formula can be satisfied. In this case we have a closed branch and an open one, which corresponds a model).

Now that we have seen the examples, we can write down the tableau rules formally.

Analytical Tableaux (Formal Treatment of $\mathcal{T}_0$)

▷ formula is analyzed in a tree to determine satisfiability

▷ branches correspond to valuations (models)

▷ one per connective

$$\frac{\mathbf{A}\wedge\mathbf{B}^{\mathsf{T}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\ \mathbf{B}^{\mathsf{T}}\end{array}}\mathcal{T}_0\wedge \qquad \frac{\mathbf{A}\wedge\mathbf{B}^{\mathsf{F}}}{\mathbf{A}^{\mathsf{F}}\;\big|\;\mathbf{B}^{\mathsf{F}}}\mathcal{T}_0\vee \qquad \frac{\neg\mathbf{A}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{F}}}\mathcal{T}_0\neg^{\mathsf{T}} \qquad \frac{\neg\mathbf{A}^{\mathsf{F}}}{\mathbf{A}^{\mathsf{T}}}\mathcal{T}_0\neg^{\mathsf{F}} \qquad \frac{\begin{array}{c}\mathbf{A}^{\alpha}\\ \mathbf{A}^{\beta}\end{array}\quad \alpha\neq\beta}{\bot}\mathcal{T}_0\mathsf{cut}$$

▷ Use rules exhaustively as long as they contribute new material

▷ **Definition 334** Call a tableau saturated, iff no rule applies, and a branch closed, iff it ends in $\bot$, else open. (open branches in saturated tableaux yield models)

▷ **Definition 335 ($\mathcal{T}_0$-Theorem/Derivability)** $\mathbf{A}$ is a $\mathcal{T}_0$-theorem ($\vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau with $\mathbf{A}^{\mathsf{F}}$ at the root.

$\Phi \subseteq wff_o(\mathcal{V}_o)$ derives $\mathbf{A}$ in $\mathcal{T}_0$ ($\Phi \vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau starting with $\mathbf{A}^{\mathsf{F}}$ and Φ^{T}.

These inference rules act on tableaux have to be read as follows: if the formulae over the line appear in a tableau branch, then the branch can be extended by the formulae or branches below the line. There are two rules for each primary connective, and a branch closing rule that adds the special symbol $\bot$ (for unsatisfiability) to a branch.

We use the tableau rules with the convention that they are only applied, if they contribute new material to the branch. This ensures termination of the tableau procedure for propositional logic (every rule eliminates one primary connective).

Definition 336 We will call a closed tableau with the signed formula $\mathbf{A}^{\alpha}$ at the root a tableau refutation for $\mathcal{A}^{\alpha}$.

The saturated tableau represents a full case analysis of what is necessary to give $\mathbf{A}$ the truth value α; since all branches are closed (contain contradictions) this is impossible.

Definition 337 We will call a tableau refutation for $\mathbf{A}^{\mathsf{F}}$ a tableau proof for $\mathbf{A}$, since it refutes the possibility of finding a model where $\mathbf{A}$ evaluates to F. Thus $\mathbf{A}$ must evaluate to T in all models, which is just our definition of validity.

Thus the tableau procedure can be used as a calculus for propositional logic. In contrast to the calculus in section ?? it does not prove a theorem **A** by deriving it from a set of axioms, but it proves it by refuting its negation. Such calculi are called negative or test calculi. Generally negative calculi have computational advantages over positive ones, since they have a built-in sense of direction.

We have rules for all the necessary connectives (we restrict ourselves to $\wedge$ and $\neg$, since the others can be expressed in terms of these two via the propositional identities above. For instance, we can write $\mathbf{A} \vee \mathbf{B}$ as $\neg(\neg\mathbf{A} \wedge \neg\mathbf{B})$, and $\mathbf{A} \Rightarrow \mathbf{B}$ as $\neg\mathbf{A} \vee \mathbf{B}, \ldots$)

We will now look at an example. Following our introduction of propositional logic in in Example 307 we look at a formulation of propositional logic with fancy variable names. Note that love(mary, bill) is just a variable name like P or X, which we have used earlier.

A Valid Real-World Example

▷ **Example 338** *Mary loves Bill and John loves Mary* entails *John loves Mary*

$$
\begin{array}{c}
\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary}) \Rightarrow \mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}} \\
\neg(\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary})) \wedge \neg\mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{F}} \\
\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary})) \wedge \neg\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{T}} \\
\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{F}} \\
\mathsf{love}(\mathsf{mary},\mathsf{bill}) \wedge \mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\neg\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\mathsf{love}(\mathsf{mary},\mathsf{bill})^{\mathsf{T}} \\
\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}} \\
\bot
\end{array}
$$

Then use the entailment theorem (Corollary 320)

We have used the entailment theorem here: Instead of showing that $\mathbf{A} \models \mathbf{B}$, we have shown that $\mathbf{A} \Rightarrow \mathbf{B}$ is a theorem. Note that we can also use the tableau calculus to try and show entailment (and fail). The nice thing is that the failed proof, we can see what went wrong.

A Falsifiable Real-World Example

▷ **Example 339** *Mary loves Bill or John loves Mary* does not entail *John loves Mary*
Try proving the implication (this fails)

$$
\begin{array}{c}
(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary})) \Rightarrow \mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}} \\
\neg(\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary})) \wedge \neg\mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{F}} \\
\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary})) \wedge \neg\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\neg\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}} \\
\neg\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{T}} \\
\neg(\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary}))^{\mathsf{F}} \\
\mathsf{love}(\mathsf{mary},\mathsf{bill}) \vee \mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
\begin{array}{c|c}
\mathsf{love}(\mathsf{mary},\mathsf{bill})^{\mathsf{T}} & \mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}} \\
 & \bot
\end{array}
\end{array}
$$

Then again the entailment theorem (Corollary 320) yields the assertion. Indeed we can make

$\mathcal{I}_\varphi(\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary})) = T$ but $\mathcal{I}_\varphi(\text{love}(\text{john}, \text{mary})) = F$.

Obviously, the tableau above is saturated, but not closed, so it is not a tableau proof for our initial entailment conjecture. We have marked the literals on the open branch green, since they allow us to read of the conditions of the situation, in which the entailment fails to hold. As we intuitively argued above, this is the situation, where Mary loves Bill. In particular, the open branch gives us a variable assignment (marked in green) that satisfies the initial formula. In this case, *Mary loves Bill*, which is a situation, where the entailment fails.

Practical Enhancements for Tableaux

Propositional Identities

▷ **Definition 340** Let $\top$ and $\bot$ be new logical constants with $\mathcal{I}(\top) = \mathsf{T}$ and $\mathcal{I}(\bot) = \mathsf{F}$ for all assignments φ.

▷ We have to following identities:

Name	for $\wedge$	for $\vee$
Idenpotence	$\varphi \wedge \varphi = \varphi$	$\varphi \vee \varphi = \varphi$
Identity	$\varphi \wedge \top = \varphi$	$\varphi \vee \bot = \varphi$
Absorption I	$\varphi \wedge \bot = \bot$	$\varphi \vee \top = \top$
Commutativity	$\varphi \wedge \psi = \psi \wedge \varphi$	$\varphi \vee \psi = \psi \vee \varphi$
Associativity	$\varphi \wedge (\psi \wedge \theta) = (\varphi \wedge \psi) \wedge \theta$	$\varphi \vee (\psi \vee \theta) = (\varphi \vee \psi) \vee \theta$
Distributivity	$\varphi \wedge (\psi \vee \theta) = \varphi \wedge \psi \vee \varphi \wedge \theta$	$\varphi \vee \psi \wedge \theta = (\varphi \vee \psi) \wedge (\varphi \vee \theta)$
Absorption II	$\varphi \wedge (\varphi \vee \theta) = \varphi$	$\varphi \vee \varphi \wedge \theta = \varphi$
De Morgan's Laws	$\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$	$\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$
Double negation	$\neg\neg\varphi = \varphi$	
Definitions	$\varphi \Rightarrow \psi = \neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

We have seen in the examples above that while it is possible to get by with only the connectives $\vee$ and $\neg$, it is a bit unnatural and tedious, since we need to eliminate the other connectives first. In this section, we will make the calculus less frugal by adding rules for the other connectives, without losing the advantage of dealing with a small calculus, which is good making statements about the calculus.

The main idea is to add the new rules as derived rules, i.e. inference rules that only abbreviate deductions in the original calculus. Generally, adding derived inference rules does not change the derivability relation of the calculus, and is therefore a safe thing to do. In particular, we will add the following rules to our tableau system.

We will convince ourselves that the first rule is a derived rule, and leave the other ones as an exercise.

Derived Rules of Inference

▷ **Definition 341** Let $\mathcal{C}$ be a calculus, a rule of inference $\frac{\mathbf{A}_1 \quad \ldots \quad \mathbf{A}_n}{\mathbf{C}}$ is called a derived inference rule in $\mathcal{C}$, iff there is a $\mathcal{C}$-proof of $\mathbf{A}_1, \ldots, \mathbf{A}_n \vdash \mathbf{C}$.

▷ **Definition 342** We have the following derived rules of inference

$$
\frac{\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{F}}\;\Big|\;\mathbf{B}^{\mathsf{T}}}
\qquad
\frac{\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\ \mathbf{B}^{\mathsf{F}}\end{array}}
\qquad
\frac{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\ \mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}\end{array}}{\mathbf{B}^{\mathsf{T}}}
$$

$$
\frac{\mathbf{A}\vee\mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{T}}\;\Big|\;\mathbf{B}^{\mathsf{T}}}
\qquad
\frac{\mathbf{A}\vee\mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{F}}\\ \mathbf{B}^{\mathsf{F}}\end{array}}
\qquad
\frac{\mathbf{A}\Leftrightarrow\mathbf{B}^{\mathsf{T}}}{\begin{array}{c|c}\mathbf{A}^{\mathsf{T}} & \mathbf{A}^{\mathsf{F}}\\ \mathbf{B}^{\mathsf{T}} & \mathbf{B}^{\mathsf{F}}\end{array}}
\qquad
\frac{\mathbf{A}\Leftrightarrow\mathbf{B}^{\mathsf{F}}}{\begin{array}{c|c}\mathbf{A}^{\mathsf{T}} & \mathbf{A}^{\mathsf{F}}\\ \mathbf{B}^{\mathsf{F}} & \mathbf{B}^{\mathsf{T}}\end{array}}
$$

$$
\begin{array}{c}
\mathbf{A}^{\mathsf{T}}\\
\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}\\
\neg\mathbf{A}\vee\mathbf{B}^{\mathsf{T}}\\
\neg(\neg\neg\mathbf{A}\wedge\neg\mathbf{B})^{\mathsf{T}}\\
\neg\neg\mathbf{A}\wedge\neg\mathbf{B}^{\mathsf{F}}\\
\begin{array}{c|c}
\neg\neg\mathbf{A}^{\mathsf{F}} & \neg\mathbf{B}^{\mathsf{F}}\\
\neg\mathbf{A}^{\mathsf{T}} & \mathbf{B}^{\mathsf{T}}\\
\mathbf{A}^{\mathsf{F}} & \\
\bot &
\end{array}
\end{array}
$$

With these derived rules, theorem proving becomes quite efficient. With these rules, the tableau (??) would have the following simpler form:

Tableaux with derived Rules (example)

Example 343

$$
\begin{array}{c}
\mathsf{love}(\mathsf{mary},\mathsf{bill})\wedge\mathsf{love}(\mathsf{john},\mathsf{mary})\Rightarrow\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}}\\
\mathsf{love}(\mathsf{mary},\mathsf{bill})\wedge\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}}\\
\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{F}}\\
\mathsf{love}(\mathsf{mary},\mathsf{bill})^{\mathsf{T}}\\
\mathsf{love}(\mathsf{john},\mathsf{mary})^{\mathsf{T}}\\
\bot
\end{array}
$$

Another thing that was awkward in (??) was that we used a proof for an implication to prove logical consequence. Such tests are necessary for instance, if we want to check consistency or informativity of new sentences[11]. Consider for instance a discourse $\Delta = \mathbf{D}^1, \ldots, \mathbf{D}^n$, where n is large. To test whether a hypothesis $\mathcal{H}$ is a consequence of Δ ($\Delta \models \mathbf{H}$) we need to show that $\mathbf{C} := (\mathbf{D}^1 \wedge \ldots) \wedge \mathbf{D}^n \Rightarrow \mathbf{H}$ is valid, which is quite tedious, since $\mathcal{C}$ is a rather large formula, e.g. if Δ is a 300 page novel. Moreover, if we want to test entailment of the form ($\Delta \models \mathbf{H}$) often, – for instance to test the informativity and consistency of every new sentence $\mathbf{H}$, then successive Δs will overlap quite significantly, and we will be doing the same inferences all over again; the entailment check is not incremental. EdNote:11

Fortunately, it is very simple to get an incremental procedure for entailment checking in the model-generation-based setting: To test whether $\Delta \models \mathbf{H}$, where we have interpreted Δ in a model generation tableau $\mathcal{T}$, just check whether the tableau closes, if we add $\neg\mathbf{H}$ to the open branches. Indeed, if the tableau closes, then $\Delta \wedge \neg\mathbf{H}$ is unsatisfiable, so $\neg((\Delta \wedge \neg\mathbf{H}))$ is valid[12], but this is equivalent to $\Delta \Rightarrow \mathbf{H}$, which is what we wanted to show. EdNote:12

Example 344 Consider for instance the following entailment in natural language.

Mary loves Bill. John loves Mary $\models$ *John loves Mary*

[13] We obtain the tableau EdNote:13

$$
\begin{array}{c}
\mathrm{love}(\mathrm{mary},\mathrm{bill})^{\mathsf{T}}\\
\mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}}\\
\neg(\mathrm{love}(\mathrm{john},\mathrm{mary}))^{\mathsf{T}}\\
\mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{F}}\\
\bot
\end{array}
$$

[11] EdNote: add reference to presupposition stuff
[12] EdNote: Fix precedence of negation
[13] EdNote: need to mark up the embedding of NL strings into Math

which shows us that the conjectured entailment relation really holds.

Soundness and Termination of Tableaux

As always we need to convince ourselves that the calculus is sound, otherwise, tableau proofs do not guarantee validity, which we are after. Since we are now in a refutation setting we cannot just show that the inference rules preserve validity: we care about unsatisfiability (which is the dual notion to validity), as we want to show the initial labeled formula to be unsatisfiable. Before we can do this, we have to ask ourselves, what it means to be (un)-satisfiable for a labeled formula or a tableau.

Soundness (Tableau)

- ▷ Idea: A test calculus is sound, iff it preserves satisfiability and the goal formulae are unsatisfiable.
- ▷ **Definition 345** A labeled formula $\mathbf{A}^{\alpha}$ is valid under φ, iff $\mathcal{I}_{\varphi}(\mathbf{A}) = \alpha$.
- ▷ **Definition 346** A tableau $\mathcal{T}$ is satisfiable, iff there is a satisfiable branch $\mathcal{P}$ in $\mathcal{T}$, i.e. if the set of formulae in $\mathcal{P}$ is satisfiable.
- ▷ **Lemma 347** *Tableau rules transform satisfiable tableaux into satisfiable ones.*
- ▷ **Theorem 348 (Soundness)** *A set Φ of propositional formulae is valid, if there is a closed tableau $\mathcal{T}$ for Φ^{F}.*
- ▷ Proof: by contradiction: Suppose Φ is not valid.

P.1 then the initial tableau is satisfiable (Φ^{F} satisfiable)

P.2 $\mathcal{T}$ satisfiable, by our Lemma.

P.3 there is a satisfiable branch (by definition)

P.4 but all branches are closed ($\mathcal{T}$ closed)

□

Thus we only have to prove Lemma 347, this is relatively easy to do. For instance for the first rule: if we have a tableau that contains $\mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}$ and is satisfiable, then it must have a satisfiable branch. If $\mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}$ is not on this branch, the tableau extension will not change satisfiability, so we can assue that it is on the satisfiable branch and thus $\mathcal{I}_{\varphi}(\mathbf{A} \wedge \mathbf{B}) = \mathsf{T}$ for some variable assignment φ. Thus $\mathcal{I}_{\varphi}(\mathbf{A}) = \mathsf{T}$ and $\mathcal{I}_{\varphi}(\mathbf{B}) = \mathsf{T}$, so after the extension (which adds the formulae $\mathbf{A}^{\mathsf{T}}$ and $\mathbf{B}^{\mathsf{T}}$ to the branch), the branch is still satisfiable. The cases for the other rules are similar.

The next result is a very important one, it shows that there is a procedure (the tableau procedure) that will always terminate and answer the question whether a given propositional formula is valid or not. This is very important, since other logics (like the often-studied first-order logic) does not enjoy this property.

Termination for Tableaux

- ▷ **Lemma 349** *The tableau procedure terminates, i.e. after a finite set of rule applications, it reaches a tableau, so that applying the tableau rules will only add labeled formulae that are already present on the branch.*
- ▷ Let us call a labeled formulae $\mathbf{A}^{\alpha}$ worked off in a tableau $\mathcal{T}$, if a tableau rule has already been applied to it.

▷ Proof:

P.1 It is easy to see tahat applying rules to worked off formulae will only add formulae that are already present in its branch.

P.2 Let $\mu(\mathcal{T})$ be the number of connectives in a labeled formulae in $\mathcal{T}$ that are not worked off.

P.3 Then each rule application to a labeled formula in $\mathcal{T}$ that is not worked off reduces $\mu(\mathcal{T})$ by at least one. (inspect the rules)

P.4 at some point the tableau only contains worked off formulae and literals.

P.5 since there are only finitely many literals in $\mathcal{T}$, so we can only apply the tableau cut rule a finite number of times. □

The Tableau calculus basically computes the disjunctive normal form: every branch is a disjunct that is a conjunct of literals. The method relies on the fact that a DNF is unsatisfiable, iff each monomial is, i.e. iff each branch contains a contradiction in form of a pair of complementary literals.

2.7.2 Resolution for Propositional Logic

The next calculus is a test calculus based on the conjunctive normal form. In contrast to the tableau method, it does not compute the normal form as it goes along, but has a pre-processing step that does this and a single inference rule that maintains the normal form. The goal of this calculus is to derive the empty clause (the empty disjunction), which is unsatisfiable.

Another Test Calculus: Resolution

▷ **Definition 350** A clause is a disjunction of literals. We will use □ for the empty disjunction (no disjuncts) and call it the empty clause.

▷ **Definition 351 (Resolution Calculus)** The resolution calculus operates a clause sets via a single inference rule:

$$\frac{P^{\mathsf{T}} \vee \mathbf{A} \quad P^{\mathsf{F}} \vee \mathbf{B}}{\mathbf{A} \vee \mathbf{B}}$$

This rule allows to add the clause below the line to a clause set which contains the two clauses above.

▷ **Definition 352 (Resolution Refutation)** Let S be a clause set, and $\mathcal{D}\colon S \vdash_{\mathcal{R}} T$ a $\mathcal{R}$ derivation then we call $\mathcal{D}$ resolution refutation, iff $\Box \in T$.

A calculus for CNF Transformation

▷ **Definition 353 (Transformation into Conjunctive Normal Form)** The CNF transformation calculus $\mathcal{CNF}$ consists of the following four inference rules on clause sets.

$$\frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}} \quad \frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \frac{\mathbf{C} \vee \neg \mathbf{A}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}} \quad \frac{\mathbf{C} \vee \neg \mathbf{A}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}}$$

▷ **Definition 354** We write $CNF(\mathbf{A})$ for the set of all clauses derivable from $\mathbf{A}^{\mathsf{F}}$ via the rules above.

▷ **Definition 355 (Resolution Proof)** We call a resolution refutation $\mathcal{P}\colon \mathit{CNF}(\mathbf{A}) \vdash_{\mathcal{R}} T$ a resolution sproof for $\mathbf{A} \in \mathit{wff}_o(\mathcal{V}_o)$.

Note: Note that the **C**-terms in the definition of the resolution calculus are necessary, since we assumed that the assumptions of the inference rule must match full formulae. The **C**-terms are used with the convention that they are optional. So that we can also simplify $(\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}$ to $\mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}$.

The background behind this notation is that $\mathbf{A}$ and $T \vee \mathbf{A}$ are equivalent for any $\mathbf{A}$. That allows us to interpret the **C**-terms in the assumptions as T and thus leave them out.

The resolution calculus as we have formulated it here is quite frugal; we have left out rules for the connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$, relying on the fact that formulae containing these connectives can be translated into ones without before CNF transformation. The advantage of having a calculus with few inference rules is that we can prove meta-properties like soundness and completeness with less effort (these proofs usually require one case per inference rule). On the other hand, adding specialized inference rules makes proofs shorter and more readable.

Fortunately, there is a way to have your cake and eat it. Derived inference rules have the property that they are formally redundant, since they do not change the expressive power of the calculus. Therefore we can leave them out when proving meta-properties, but include them when actually using the calculus.

Derived Rules of Inference

▷ **Definition 356** Let $\mathcal{C}$ be a calculus, a rule of inference $\dfrac{\mathbf{A}_1 \quad \ldots \quad \mathbf{A}_n}{\mathbf{C}}$ is called a derived inference rule in $\mathcal{C}$, iff there is a $\mathcal{C}$-proof of $\mathbf{A}_1, \ldots, \mathbf{A}_n \vdash \mathbf{C}$.

▷ **Example 357**

$$\dfrac{\dfrac{\dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee (\neg\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}}{\mathbf{C} \vee \neg\mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \mapsto \quad \dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}}$$

▷ Others:

$$\dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \dfrac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{T}}} \quad \dfrac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{F}}}$$

With these derived rules, theorem proving becomes quite efficient. To get a better understanding of the calculus, we look at an example: we prove an axiom of the Hilbert Calculus we have studied above.

Example: Proving Axiom S

▷ **Example 358** Clause Normal Form transformation

$$\frac{\frac{\frac{\frac{(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}{P \Rightarrow Q \Rightarrow R^{\mathsf{T}}; (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee (Q \Rightarrow R)^{\mathsf{T}}; P \Rightarrow Q^{\mathsf{T}}; P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}; P^{\mathsf{F}} \vee Q^{\mathsf{T}}; P^{\mathsf{T}}; R^{\mathsf{F}}}}{}$$

$$CNF = \{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}, P^{\mathsf{F}} \vee Q^{\mathsf{T}}, P^{\mathsf{T}}, R^{\mathsf{F}}\}$$

▷ **Example 359** Resolution Proof

1	$P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}$	initial
2	$P^{\mathsf{F}} \vee Q^{\mathsf{T}}$	initial
3	P^{T}	initial
4	R^{F}	initial
5	$P^{\mathsf{F}} \vee Q^{\mathsf{F}}$	resolve 1.3 with 4.1
6	Q^{F}	resolve 5.1 with 3.1
7	P^{F}	resolve 2.2 with 6.1
8	$\Box$	resolve 7.1 with 3.1

Chapter 3

How to build Computers and the Internet (in principle)

In this chapter, we will learn how to build computational devices (aka. computers) from elementary parts (combinational, arithmetic, and sequential circuits), how to program them with low-level programming languages, and how to interpret/compile higher-level programming languages for these devices. Then we will understand how computers can be networked into the distributed computation system we came to call the Internet and the information system of the world-wide web.

In all of these investigations, we will only be interested on how the underlying devices, algorithms and representations work in principle, clarifying the concepts and complexities involved, while abstracting from much of the engineering particulars of modern microprocessors. In keeping with this, we will conclude this chapterby an investigation into the fundamental properties and limitations of computation.

3.1 Combinational Circuits

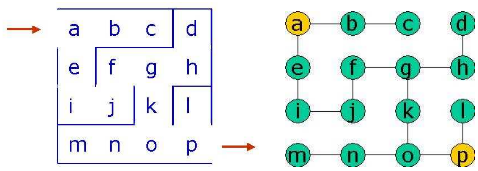
We will now study a new model of computation that comes quite close to the circuits that execute computation on today's computers. Since the course studies computation in the context of computer science, we will abstract away from all physical issues of circuits, in particular the construction of gats and timing issues. This allows to us to present a very mathematical view of circuits at the level of annotated graphs and concentrate on qualitative complexity of circuits. Some of the material in this section is inspired by [KP95].

We start out our foray into circuits by laying the mathematical foundations of graphs and trees in Subsection 3.1.0, and then build a simple theory of combinational circuits in Subsection 3.1.1 and study their time and space complexity in Subsection 3.1.2. We introduce combinational circuits for computing with numbers, by introducing positional number systems and addition in Subsection 3.2.0 and covering 2s-complement numbers and subtraction in Subsection 3.2.1. A basic introduction to sequential logic circuits and memory elements in Section 3.2 concludes our study of circuits.

3.1.1 Graphs and Trees

Some more Discrete Math: Graphs and Trees

▷ Remember our Maze Example from the Intro? (long time ago)



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

- We represented the maze as a graph for clarity.
- Now, we are interested in circuits, which we will also represent as graphs.
- Let us look at the theory of graphs first (so we know what we are doing)

Graphs and trees are fundamental data structures for computer science, they will pop up in many disguises in almost all areas of CS. We have already seen various forms of trees: formula trees, tableaux, ... We will now look at their mathematical treatment, so that we are equipped to talk and think about combinatory circuits.

We will first introduce the formal definitions of graphs (trees will turn out to be special graphs), and then fortify our intuition using some examples.

Basic Definitions: Graphs

- **Definition 360** An undirected graph is a pair $\langle V, E\rangle$ such that
  - V is a set of vertices (or nodes) (draw as circles)
  - $E \subseteq \{\{v, v'\} \mid v, v' \in V \wedge (v \neq v')\}$ is the set of its undirected edges (draw as lines)
- **Definition 361** A directed graph (also called digraph) is a pair $\langle V, E\rangle$ such that
  - V is a set of vertices
  - $E \subseteq V \times V$ is the set of its directed edges
- **Definition 362** Given a graph $G = \langle V, E\rangle$. The in-degree $\mathrm{indeg}(v)$ and the out-degree $\mathrm{outdeg}(v)$ of a vertex $v \in V$ are defined as
  - $\mathrm{indeg}(v) = \#(\{w \mid \langle w, v\rangle \in E\})$
  - $\mathrm{outdeg}(v) = \#(\{w \mid \langle v, w\rangle \in E\})$

  Note: For an undirected graph, $\mathrm{indeg}(v) = \mathrm{outdeg}(v)$ for all nodes v.

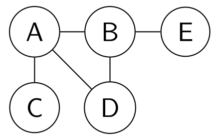
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for indeg and outdeg are the obvious variants: $\mathrm{indeg}(v) = \#(\{w \mid \{w, v\} \in E\})$ and $\mathrm{outdeg}(v) = \#(\{w \mid \{v, w\} \in E\})$

In the following if we do not specify that a graph is undirected, it will be assumed to be directed.

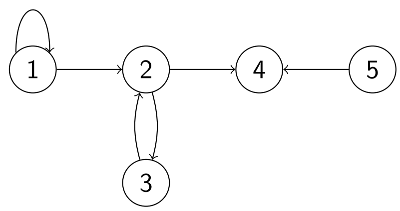
This is a very abstract yet elementary definition. We only need very basic concepts like sets and ordered pairs to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

Examples

▷ ▷ **Example 363** An undirected graph $G_1 = \langle V_1, E_1 \rangle$, where $V_1 = \{A, B, C, D, E\}$ and $E_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{B, E\}\}$



▷ **Example 364** A directed graph $G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 2, 4 \rangle, \langle 5, 4 \rangle\}$



In a directed graph, the edges (shown as the connections between the circular nodes) have a direction (mathematically they are ordered pairs), whereas the edges in an undirected graph do not (mathematically, they are represented as a set of two elements, in which there is no natural order).

Note furthermore that the two diagrams are not graphs in the strict sense: they are only pictures of graphs. This is similar to the famous painting by René Magritte that you have surely seen before.

The Graph Diagrams are not Graphs

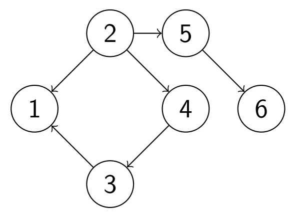
They are pictures of graphs (of course!)

If we think about it for a while, we see that directed graphs are nothing new to us. We have defined a directed graph to be a set of pairs over a base set (of nodes). These objects we have seen in the beginning of this course and called them relations. So directed graphs are special relations. We will now introduce some nomenclature based on this intuition.

Directed Graphs

- ▷ Idea: Directed Graphs are nothing else than relations
- ▷ **Definition 365** Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$
  - ▷ initial, iff there is no $w \in V$ such that $\langle w, v \rangle \in E$. (no predecessor)
  - ▷ terminal, iff there is no $w \in V$ such that $\langle v, w \rangle \in E$. (no successor)

  In a graph G, node v is also called a source (sink) of G, iff it is initial (terminal) in G.
- ▷ **Example 366** The node 2 is initial, and the nodes 1 and 6 are terminal in



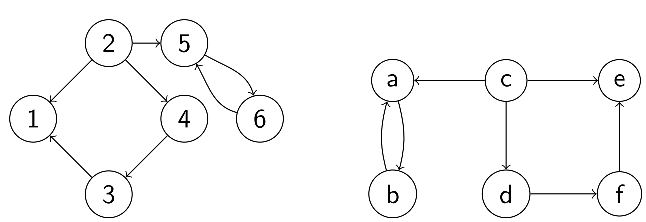
For mathematically defined objects it is always very important to know when two representations are equal. We have already seen this for sets, where $\{a, b\}$ and $\{b, a, b\}$ represent the same set: the set with the elements a and b. In the case of graphs, the condition is a little more involved: we have to find a bijection of nodes that respects the edges.

Graph Isomorphisms

- ▷ **Definition 367** A graph isomorphism between two graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is a bijective function $\psi \colon V \to V'$ with

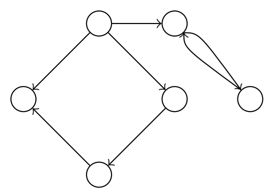
directed graphs	undirected graphs
$\langle a, b \rangle \in E \Leftrightarrow \langle \psi(a), \psi(b) \rangle \in E'$	$\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in E'$

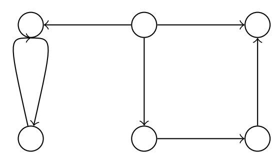
- ▷ **Definition 368** Two graphs G and G' are equivalent iff there is a graph-isomorphism ψ between G and G'.
- ▷ **Example 369** G_1 and G_2 are equivalent as there exists a graph isomorphism $\psi := \{a \mapsto 5, b \mapsto 6, c \mapsto 2, d \mapsto 4, e \mapsto 1, f \mapsto 3\}$ between them.



Note that we have only marked the circular nodes in the diagrams with the names of the elements that represent the nodes for convenience, the only thing that matters for graphs is which nodes are connected to which. Indeed that is just what the definition of graph equivalence via the existence of an isomorphism says: two graphs are equivalent, iff they have the same number of nodes and the same edge connection pattern. The objects that are used to represent them are purely coincidental, they can be changed by an isomorphism at will. Furthermore, as we have seen in the example, the shape of the diagram is purely an artifact of the presentation; It does not matter at all.

So the following two diagrams stand for the same graph, (it is just much more difficult to state the graph isomorphism)



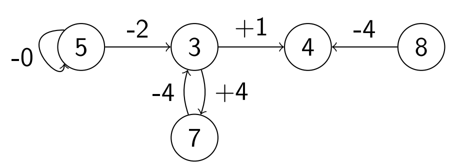


Note that directed and undirected graphs are totally different mathematical objects. It is easy to think that an undirected edge $\{a, b\}$ is the same as a pair $\langle a, b\rangle, \langle b, a\rangle$ of directed edges in both directions, but a priory these two have nothing to do with each other. They are certainly not equivalent via the graph equivalent defined above; we only have graph equivalence between directed graphs and also between undirected graphs, but not between graphs of differing classes.

Now that we understand graphs, we can add more structure. We do this by defining a labeling function from nodes and edges.

Labeled Graphs

- **Definition 370** A labeled graph G is a triple $\langle V, E, f\rangle$ where $\langle V, E\rangle$ is a graph and $f: V \cup E \to R$ is a partial function into a set R of labels.
- **Notation 371** write labels next to their vertex or edge. If the actual name of a vertex does not matter, its label can be written into it.
- **Example 372** $G = \langle V, E, f\rangle$ with $V = \{A, B, C, D, E\}$, where
  - $E = \{\langle A, A\rangle, \langle A, B\rangle, \langle B, C\rangle, \langle C, B\rangle, \langle B, D\rangle, \langle E, D\rangle\}$
  - $f: V \cup E \to \{+, -, \emptyset\} \times \{1, \ldots, 9\}$ with
    - $f(A) = 5,\ f(B) = 3,\ f(C) = 7,\ f(D) = 4,\ f(E) = 8,$
    - $f(\langle A, A\rangle) = -0,\ f(\langle A, B\rangle) = -2,\ f(\langle B, C\rangle) = +4,$
    - $f(\langle C, B\rangle) = -4,\ f(\langle B, D\rangle) = +1,\ f(\langle E, D\rangle) = -4$



Note that in this diagram, the markings in the nodes do denote something: this time the labels given by the labeling function f, not the objects used to construct the graph. This is somewhat confusing, but traditional.

Now we come to a very important concept for graphs. A path is intuitively a sequence of nodes that can be traversed by following directed edges in the right direction or undirected edges.

Paths in Graphs

▷ **Definition 373** Given a directed graph $G = \langle V, E\rangle$, then we call a vector $p = \langle v_0, \ldots, v_n\rangle \in V^{n+1}$ a path in G iff $\langle v_{i-1}, v_i\rangle \in E$ for all $(1 \leq i \leq n)$, $n > 0$.

- ▷ v_0 is called the start of p (write start(p))
- ▷ v_n is called the end of p (write end(p))
- ▷ n is called the length of p (write len(p))

Note: Not all v_i-s in a path are necessarily different.

▷ **Notation 374** For a graph $G = \langle V, E\rangle$ and a path $p = \langle v_0, \ldots, v_n\rangle \in V^{n+1}$, write

- ▷ $v \in p$, iff $v \in V$ is a vertex on the path ($\exists i.v_i = v$)
- ▷ $e \in p$, iff $e = \langle v, v'\rangle \in E$ is an edge on the path ($\exists i.v_i = v \wedge v_{i+1} = v'$)

▷ **Notation 375** We write $\Pi(G)$ for the set of all paths in a graph G.

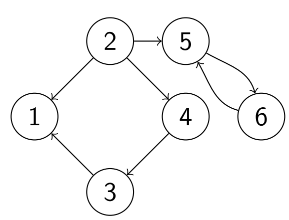
An important special case of a path is one that starts and ends in the same node. We call it a cycle. The problem with cyclic graphs is that they contain paths of infinite length, even if they have only a finite number of nodes.

Cycles in Graphs

▷ **Definition 376** Given a graph $G = \langle V, E\rangle$, then

- ▷ a path p is called cyclic (or a cycle) iff start(p) = end(p).
- ▷ a cycle $\langle v_0, \ldots, v_n\rangle$ is called simple, iff $v_i \neq v_j$ for $1 \leq i, j \leq n$ with $i \neq j$.
- ▷ graph G is called acyclic iff there is no cyclic path in G.

▷ **Example 377** $\langle 2, 4, 3\rangle$ and $\langle 2, 5, 6, 5, 6, 5\rangle$ are paths in



$\langle 2, 4, 3, 1, 2\rangle$ is not a path (no edge from vertex 1 to vertex 2)

The graph is not acyclic ($\langle 5, 6, 5\rangle$ is a cycle)

▷ **Definition 378** We will sometimes use the abbreviation DAG for "directed acyclic graph".

Of course, speaking about cycles is only meaningful in directed graphs, since undirected graphs can only be acyclic, iff they do not have edges at all.

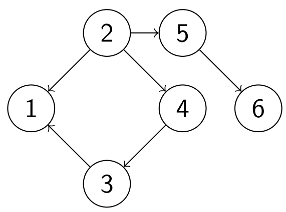
Graph Depth

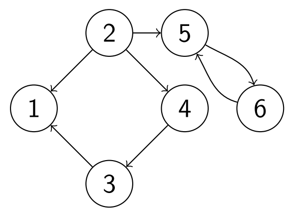
▷ **Definition 379** Let $G := \langle V, E\rangle$ be a digraph, then the depth dp(v) of a vertex $v \in V$ is defined to be 0, if v is a source of G and $\sup\{\text{len}(p) \mid \text{indeg}(\text{start}(p)) = 0 \wedge \text{end}(p) = v\}$

otherwise, i.e. the length of the longest path from a source of G to v. (⚠ can be infinite)

▷ **Definition 380** Given a digraph $G = \langle V, E \rangle$. The depth ($\mathrm{dp}(G)$) of G is defined as $\sup\{\mathrm{len}(p) \mid p \in \Pi(G)\}$, i.e. the maximal path length in G.

▷ **Example 381** The vertex 6 has depth two in the left graph and infine depth in the right one.





The left graph has depth three (cf. node 1), the right one has infinite depth (cf. nodes 5 and 6)

We now come to a very important special class of graphs, called trees.

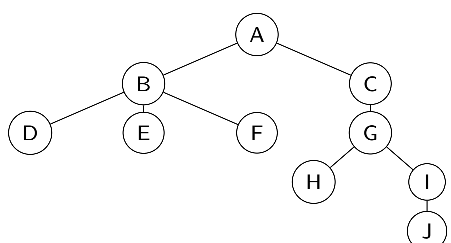
Trees

▷ **Definition 382** A tree is a directed acyclic graph $G = \langle V, E \rangle$ such that

- ▷ There is exactly one initial node $v_r \in V$ (called the root)
- ▷ All nodes but the root have in-degree 1.

We call v the parent of w, iff $\langle v, w \rangle \in E$ (w is a child of v). We call a node v a leaf of G, iff it is terminal, i.e. if it does not have children.

▷ **Example 383** A tree with root A and leaves D, E, F, H, and J.



F is a child of B and G is the parent of H and I.

▷ **Lemma 384** *For any node $v \in V$ except the root v_r, there is exactly one path $p \in \Pi(G)$ with $\mathrm{start}(p) = v_r$ and $\mathrm{end}(p) = v$.* *(proof by induction on the number of nodes)*

In Computer Science trees are traditionally drawn upside-down with their root at the top, and the leaves at the bottom. The only reason for this is that (like in nature) trees grow from the root upwards and if we draw a tree it is convenient to start at the top of the page downwards, since we do not have to know the height of the picture in advance.

Let us now look at a prominent example of a tree: the parse tree of a Boolean expression. Intuitively, this is the tree given by the brackets in a Boolean expression. Whenever we have an expression of the form $\mathbf{A} \circ \mathbf{B}$, then we make a tree with root $\circ$ and two subtrees, which are constructed from $\mathbf{A}$ and $\mathbf{B}$ in the same manner.

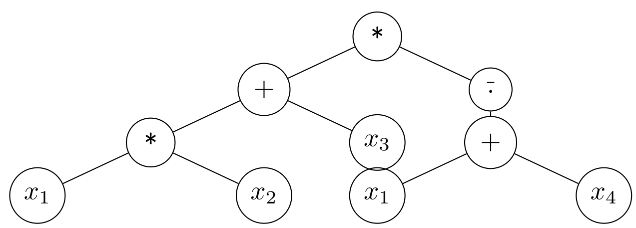
This allows us to view Boolean expressions as trees and apply all the mathematics (nomenclature and results) we will develop for them.

The Parse-Tree of a Boolean Expression

▷ **Definition 385** The parse-tree P_e of a Boolean expression e is a labeled tree $P_e = \langle V_e, E_e, f_e\rangle$, which is recursively defined as

- ▹ if $e = \overline{e'}$ then $V_e := V_{e'} \cup \{v\}$, $E_e := E_{e'} \cup \{\langle v, {v'}_r\rangle\}$, and $f_e := f_{e'} \cup \{v \mapsto \overline{\cdot}\}$, where $P_{e'} = \langle V_{e'}, E_{e'}, f_{e'}\rangle$ is the parse-tree of e', ${v'}_r$ is the root of $P_{e'}$, and v is an object not in $V_{e'}$.
- ▹ if $e = e_1 \circ e_2$ with $\circ \in \{*, +\}$ then $V_e := V_{e_1} \cup V_{e_2} \cup \{v\}$, $E_e := E_{e_1} \cup E_{e_2} \cup \{\langle v, v_1^r\rangle, \langle v, v_2^r\rangle\}$, and $f_e := f_{e_1} \cup f_{e_2} \cup \{v \mapsto \circ\}$, where the $P_{e_i} = \langle V_{e_i}, E_{e_i}, f_{e_i}\rangle$ are the parse-trees of e_i and v_i^r is the root of P_{e_i} and v is an object not in $V_{e_1} \cup V_{e_2}$.
- ▹ if $e \in (V \cup C_{\mathsf{bool}})$ then, $V_e = \{e\}$ and $E_e = \emptyset$.

▷ **Example 386** the parse tree of $(x_1 * x_2 + x_3) * \overline{x_1 + x_4}$ is



3.1.2 Introduction to Combinatorial Circuits

We will now come to another model of computation: combinational circuits (also called combinational circuits). These are models of logic circuits (physical objects made of transistors (or cathode tubes) and wires, parts of integrated circuits, etc), which abstract from the inner structure for the switching elements (called gates) and the geometric configuration of the connections. Thus, combinational circuits allow us to concentrate on the functional properties of these circuits, without getting bogged down with e.g. configuration- or geometric considerations. These can be added to the models, but are not part of the discussion of this course.

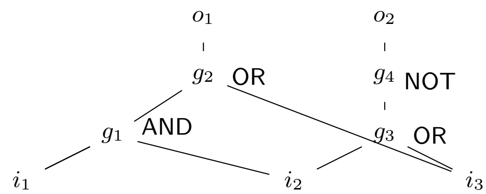
Combinational Circuits as Graphs

▷ **Definition 387** A combinational circuit is a labeled acyclic graph $G = \langle V, E, f_g\rangle$ with label set $\{\mathsf{OR}, \mathsf{AND}, \mathsf{NOT}\}$, such that

- ▹ $\mathrm{indeg}(v) = 2$ and $\mathrm{outdeg}(v) = 1$ for all nodes $v \in {f_g}^{-1}(\{\mathsf{AND}, \mathsf{OR}\})$
- ▹ $\mathrm{indeg}(v) = \mathrm{outdeg}(v) = 1$ for all nodes $v \in {f_g}^{-1}(\{\mathsf{NOT}\})$

We call the set $I(G)$ ($O(G)$) of initial (terminal) nodes in G the input (output) vertices, and the set $F(G) := V \backslash ((I(G) \cup O(G)))$ the set of gates.

▷ **Example 388** The following graph $G_{cir1} = \langle V, E\rangle$ is a combinational circuit



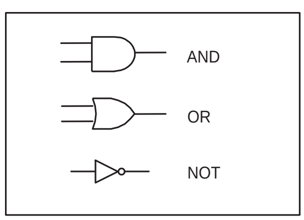
▷ **Definition 389** Add two special input nodes 0, 1 to a combinational circuit G to form a combinational circuit with constants. (will use this from now on)

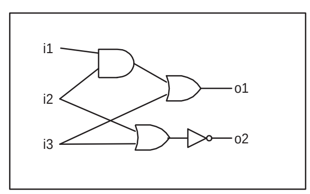
So combinational circuits are simply a class of specialized labeled directed graphs. As such, they inherit the nomenclature and equality conditions we introduced for graphs. The motivation for the restrictions is simple, we want to model computing devices based on gates, i.e. simple computational devices that behave like logical connectives: the AND gate has two input edges and one output edge; the the output edge has value 1, iff the two input edges do too.

Since combinational circuits are a primary tool for understanding logic circuits, they have their own traditional visual display format. Gates are drawn with special node shapes and edges are traditionally drawn on a rectangular grid, using bifurcating edges instead of multiple lines with blobs distinguishing bifurcations from edge crossings. This graph design is motivated by readability considerations (combinational circuits can become rather large in practice) and the layout of early printed circuits.

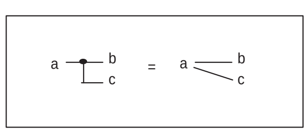
Using Special Symbols to Draw Combinational Circuits

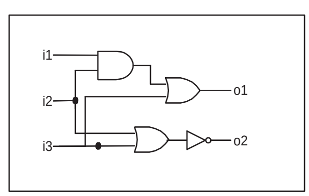
▷ The symbols for the logic gates AND, OR, and NOT.





▷ Junction Symbols as shorthands for several edges





In particular, the diagram on the lower right is a visualization for the combinatory circuit G_{circ1} from the last slide.

To view combinational circuits as models of computation, we will have to make a connection between the gate structure and their input-output behavior more explicit. We will use a tool for this we have studied in detail before: Boolean expressions. The first thing we will do is to annotate all the edges in a combinational circuit with Boolean expressions that correspond to the values on the edges (as a function of the input values of the circuit).

Computing with Combinational Circuits

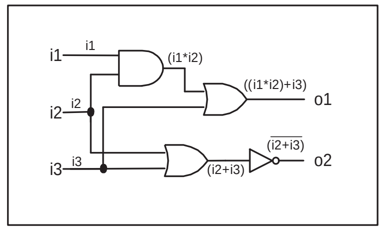
▷ Combinational Circuits and parse trees for Boolean expressions look similar

▷ Idea: Let's annotate edges in combinational circuit with Boolean Expressions!

▷ **Definition 390** Given a combinational circuit $G = \langle V, E, f_g \rangle$ and an edge $e = \langle v, w \rangle \in E$, the expression label $f_L(e)$ is defined as

$f_L(\langle v, w \rangle)$	if
v	$v \in I(G)$
$\overline{f_L(\langle u, v \rangle)}$	$f_g(v) = \text{NOT}$
$f_L(\langle u, v \rangle) * f_L(\langle u', v \rangle)$	$f_g(v) = \text{AND}$
$f_L(\langle u, v \rangle) + f_L(\langle u', v \rangle)$	$f_g(v) = \text{OR}$

▷ **Example 391**



Armed with the expression label of edges we can now make the computational behavior of combinatory circuits explicit. The intuition is that a combinational circuit computes a certain Boolean function, if we interpret the input vertices as obtaining as values the corresponding arguments and passing them on to gates via the edges in the circuit. The gates then compute the result from their input edges and pass the result on to the next gate or an output vertex via their output edge.

Computing with Combinational Circuits

▷ **Definition 392** A combinational circuit $G = \langle V, E, f_g \rangle$ with input vertices $i_1, \ldots, i_n$ and output vertices $o_1, \ldots, o_m$ computes an n-ary Boolean function

$$f\colon \{0,1\}^n \to \{0,1\}^m; \langle i_1, \ldots, i_n \rangle \mapsto \langle f_{e_1}(i_1, \ldots, i_n), \ldots, f_{e_m}(i_1, \ldots, i_n) \rangle$$

where $e_i = f_L(\langle v, o_i \rangle)$.

▷ **Example 393** The circuit in Example 391 computes the Boolean function $f\colon \{0,1\}^3 \to \{0,1\}^2; \langle i_1, i_2, i_3 \rangle \mapsto \langle f_{i_1 * i_2 + i_3}, f_{\overline{i_2 * i_3}} \rangle$

▷ **Definition 394** The cost $C(G)$ of a circuit G is the number of gates in G.

▷ Problem: For a given boolean function f, find combinational circuits of minimal cost and depth that compute f.

Note: The opposite problem, i.e., the conversion of a combinational circuit into a Boolean function, can be solved by determining the related expressions and their parse-trees. Note that there is a canonical graph-isomorphism between the parse-tree of an expression e and a combinational circuit that has an output that computes f_e.

3.1.3 Realizing Complex Gates Efficiently

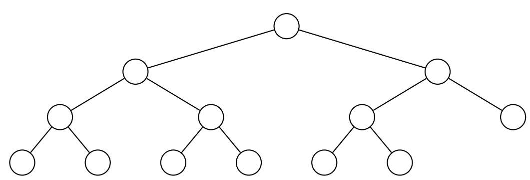
The main properties of combinatory circuits we are interested in studying will be the the number of gates and the depth of a circuit. The number of gates is of practical importance, since it is a measure of the cost that is needed for producing the circuit in the physical world. The depth is interesting, since it is an approximation for the speed with which a combinatory circuit can compute: while in most physical realizations, signals can travel through wires at at (almost) the speed of light, gates have finite computation times.

Therefore we look at special configurations for combinatory circuits that have good depth and cost. These will become important, when we build actual combinational circuits with given input/output behavior.

Balanced Binary Trees

Balanced Binary Trees

- **Definition 395 (Binary Tree)** A binary tree is a tree where all nodes have out-degree 2 or 0.
- **Definition 396** A binary tree G is called balanced iff the depth of all leaves differs by at most by 1, and fully balanced, iff the depth difference is 0.
- Constructing a binary tree $G_{\mathsf{bbt}} = \langle V, E \rangle$ with n leaves
  - step 1: select some $u \in V$ as root, ($V_1 := \{u\}$, $E_1 := \emptyset$)
  - step 2: select $v, w \in V$ not yet in G_{bbt} and add them, ($V_i = V_{i-1} \cup \{v, w\}$)
  - step 3: add two edges $\langle u, v \rangle$ and $\langle u, w \rangle$ where u is the leftmost of the shallowest nodes with $\text{outdeg}(u) = 0$, ($E_i := E_{i-1} \cup \{\langle u, v \rangle, \langle u, w \rangle\}$)
  - repeat steps 2 and 3 until $i = n$ ($V = V_n$, $E = E_n$)
- **Example 397** 7 leaves



We will now establish a few properties of these balanced binary trees that show that they are good building blocks for combinatory circuits.

Size Lemma for Balanced Trees

- **Lemma 398** *Let $G = \langle V, E \rangle$ be a balanced binary tree of depth $n > i$, then the set $V_i := \{v \in V \mid dp(v) = i\}$ of nodes at depth i has cardinality 2^i.*
- Proof: via induction over the depth i.

P.1 We have to consider two cases

P.1.1 $i = 0$:

then $V_i = \{v_r\}$, where v_r is the root, so $\#(V_0) = \#(\{v_r\}) = 1 = 2^0$.

P.1.2 $i > 0$:

then V_{i-1} contains 2^{i-1} vertices (IH)

P.1.2.2 By the definition of a binary tree, each $v \in V_{i-1}$ is a leaf or has two children that are at depth i.

P.1.2.3 As G is balanced and $\text{dp}(G) = n > i$, V_{i-1} cannot contain leaves.

P.1.2.4 Thus $\#(V_i) = 2 \cdot \#(V_{i-1}) = 2 \cdot 2^{i-1} = 2^i$. □

□

▷ **Corollary 399** *A fully balanced tree of depth d has $2^{d+1}-1$ nodes.*

Proof:

▷ **P.1** Let $G := \langle V, E\rangle$ be a fully balanced tree

Then $\#(V) = \sum_{i=1}^{d} 2^i = 2^{d+1}-1$. □

This shows that balanced binary trees grow in breadth very quickly, a consequence of this is that they are very shallow (and this compute very fast), which is the essence of the next result.

Depth Lemma for Balanced Trees

P.2 ▷ **Lemma 400** *Let $G = \langle V, E\rangle$ be a balanced binary tree, then $dp(G) = \lfloor log_2(\#(V))\rfloor$.*

▷ Proof: by calculation

P.1 Let $V' := V \backslash W$, where W is the set of nodes at level $d = \mathrm{dp}(G)$

P.2 By the size lemma, $\#(V') = 2^{d-1+1} - 1 = 2^d - 1$

P.3 then $\#(V) = 2^d - 1 + k$, where $k = \#(W)$ and $(1 \leq k \leq 2^d)$

P.4 so $\#(V) = c \cdot 2^d$ where $c \in \mathbb{R}$ and $1 \leq c < 2$, or $0 \leq \log_2(c) < 1$

P.5 thus $\log_2(\#(V)) = \log_2(c \cdot 2^d) = \log_2(c) + d$ and

P.6 hence $d = \log_2(\#(V)) - \log_2(c) = \lfloor \log_2(\#(V))\rfloor$. □

Leaves of Binary Trees

▷ **Lemma 401** *Any binary tree with m leaves has $2m-1$ vertices.*

▷ Proof: by induction on m.

P.1 We have two cases $m = 1$:

then $V = \{v_r\}$ and $\#(V) = 1 = 2 \cdot 1 - 1$.

P.1.2 $m > 1$:

P.1.2.1 then any binary tree G with $m-1$ leaves has $2m-3$ vertices (IH)

P.1.2.2 To get m leaves, add 2 children to some leaf of G. (add two to get one more)

P.1.2.3 Thus $\#(V) = 2 \cdot m - 3 + 2 = 2 \cdot m - 1$. □

□

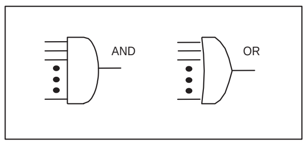
In particular, the size of a binary tree is independent of the its form if we fix the number of leaves. So we can optimimze the depth of a binary tree by taking a balanced one without a size penalty. This will become important for building fast combinatory circuits.

Realizing n-ary Gates

We now use the results on balanced binary trees to build generalized gates as building blocks for combinational circuits.

n-ary Gates as Subgraphs

- ▷ Idea: Identify (and abbreviate) frequently occurring subgraphs
- ▷ **Definition 402** $\mathsf{AND}(x_1, \ldots, x_n) := 1\prod_{i=1}^{n} x_i$ and $\mathsf{OR}(x_1, \ldots, x_n) := 1\sum_{i=1}^{n} x_i$
- ▷ Note: These can be realized as balanced binary trees G_n
- ▷ **Corollary 403** *$C(G_n) = n - 1$ and $dp(G_n) = \lfloor log_2(n) \rfloor$.*
- ▷ **Notation 404**



Using these building blocks, we can establish a worst-case result for the depth of a combinatory circuit computing a given Boolean function.

Worst Case Depth Theorem for Combinational Circuits

- ▷ **Theorem 405** *The worst case depth $dp(G)$ of a combinational circuit G which realizes an $k \times n$-dimensional boolean function is bounded by $dp(G) \leq n + \lceil log_2(n) \rceil + 1$.*
- ▷ Proof: The main trick behind this bound is that AND and OR are associative and that the according gates can be arranged in a balanced binary tree.

  P.1 Function f corresponding to the output o_j of the circuit G can be transformed in DNF

  P.2 each monomial consists of at most n literals

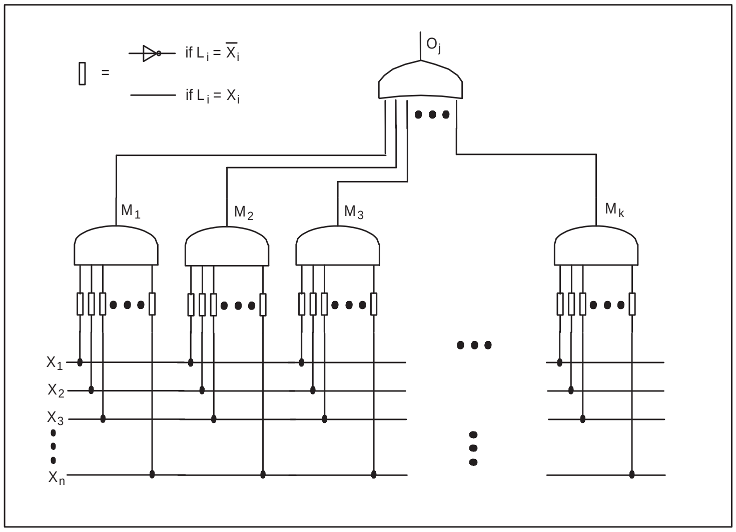
  P.3 the possible negation of inputs for some literals can be done in depth 1

  P.4 for each monomial the ANDs in the related circuit can be arranged in a balanced binary tree of depth $\lceil \log_2(n) \rceil$

  P.5 there are at most 2^n monomials which can be ORed together in a balanced binary tree of depth $\lceil \log_2(2^n) \rceil = n$. □

Of course, the depth result is related to the first worst-case complexity result for Boolean expressions (Theorem 270); it uses the same idea: to use the disjunctive normal form of the Boolean function. However, instead of using a Boolean expression, we become more concrete here and use a combinational circuit.

An example of a DNF circuit

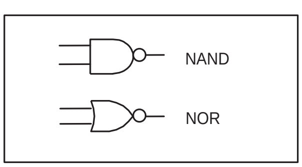


In the circuit diagram above, we have of course drawn a very particular case (as an example for possible others.) One thing that might be confusing is that it looks as if the lower n-ary conjunction operators look as if they have edges to all the input variables, which a DNF does not have in general.

Of course, by now, we know how to do better in practice. Instead of the DNF, we can always compute the minimal polynomial for a given Boolean function using the Quine-McCluskey algorithm and derive a combinational circuit from this. While this does not give us any theoretical mileage (there are Boolean functions where the DNF is already the minimal polynomial), but will greatly improve the cost in practice.

Until now, we have somewhat arbitrarily concentrated on combinational circuits with AND, OR, and NOT gates. The reason for this was that we had already developed a theory of Boolean expressions with the connectives $\vee$, $\wedge$, and $\neg$ that we can use. In practical circuits often other gates are used, since they are simpler to manufacture and more uniform. In particular, it is sufficient to use only one type of gate as we will see now.

Other Logical Connectives and Gates

- ▷ Are the gates AND, OR, and NOT ideal?
- ▷ Idea: Combine NOT with the binary ones to NAND, NOR (enough?)
- ▷ 

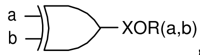
NAND	1	0
1	0	1
0	1	1

and

NOR	1	0
1	0	0
0	0	1

- ▷ Corresponding logical conectives are written as $\uparrow$ (NAND) and $\downarrow$ (NOR).
- ▷ We will also need the exclusive or (XOR) connective that returns 1 iff either of its operands is 1.

XOR	1	0
1	0	1
0	1	0

▷ The gate is written as 

, the logical connective as $\oplus$.

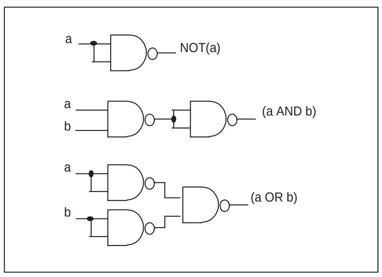
The Universality of NAND and NOR

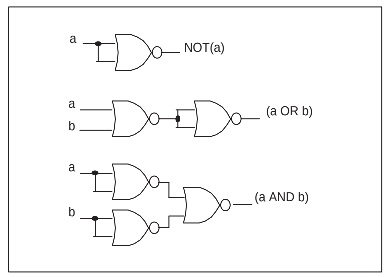
▷ **Theorem 406** *NAND and NOR are universal; i.e. any Boolean function can be expressed in terms of them.*

▷ Proof Sketch: Express AND, OR, and NOT via NAND and NOR respectively:

NOT(a)	NAND(a, a)	NOR(a, a)
AND(a, b)	NAND(NAND(a, b), NAND(a, b))	NOR(NOR(a, a), NOR(b, b))
OR(a, b)	NAND(NAND(a, a), NAND(b, b))	NOR(NOR(a, b), NOR(a, b))

▷ here are the corresponding diagrams for the combinational circuits.





Of course, a simple substitution along these lines will blow up the cost of the circuits by a factor of up to three and double the depth, which would be prohibitive. To get around this, we would have to develop a theory of Boolean expressions and complexity using the NAND and NOR connectives, along with suitable replacements for the Quine-McCluskey algorithm. This would give cost and depth results comparable to the ones developed here. This is beyond the scope of this course.

3.2 Arithmetic Circuits

3.2.1 Basic Arithmetics with Combinational Circuits

We have seen that combinational circuits are good models for implementing Boolean functions: they allow us to make predictions about properties like costs and depths (computation speed), while abstracting from other properties like geometrical realization, etc.

We will now extend the analysis to circuits that can compute with numbers, i.e. that implement the basic arithmetical operations (addition, multiplication, subtraction, and division on integers). To be able to do this, we need to interpret sequences of bits as integers. So before we jump into arithmetical circuits, we will have a look at number representations.

Positional Number Systems

Positional Number Systems

- Problem: For realistic arithmetics we need better number representations than the unary natural numbers ($|\varphi_n(unary)| \in \Theta(n)$ [number of /])
- Recap: the unary number system
  - build up numbers from /es (start with ' ' and add /)
  - addition $\oplus$ as concatenation ($\odot$, exp, ... defined from that)

Idea: build a clever code on the unary numbers

- interpret sequences of /es as strings: ϵ stands for the number 0
- **Definition 407** A positional number system $\mathcal{N}$ is a triple $\mathcal{N} = \langle D_b, \varphi_b, \psi_b \rangle$ with
  - D_b is a finite alphabet of b digits. ($b := \#(D_b)$ base or radix of $\mathcal{N}$)
  - $\varphi_b \colon D_b \to \{\epsilon, /, \ldots, /^{[b-1]}\}$ is bijective (first b unary numbers)
  - $\psi_b \colon \ {D_b}^+ \to \{/\}^*; \langle n_k, \ldots, n_1 \rangle \mapsto \bigoplus_{i=1}^{k} \varphi_b(n_i) \odot \mathrm{exp}(/^{[b]}, /^{[i-1]})$ (extends φ_b to string code)

 238 JACOBS UNIVERSITY

In the unary number system, it was rather simple to do arithmetics, the most important operation (addition) was very simple, it was just concatenation. From this we can implement the other operations by simple recursive procedures, e.g. in SML or as abstract procedures in abstract data types. To make the arguments more transparent, we will use special symbols for the arithmetic operations on unary natural numbers: $\oplus$ (addition), $\odot$ (multiplication), $\bigoplus_{i=1}^{n}$ (sum over n numbers), and $\bigodot_{i=1}^{n}$ (product over n numbers).

The problem with the unary number system is that it uses enormous amounts of space, when writing down large numbers. Using the Landau notation we introduced earlier, we see that for writing down a number n in unary representation we need n slashes. So if $|\varphi_n(unary)|$ is the "cost of representing n in unary representation", we get $|\varphi_n(unary)| \in \Theta(n)$. Of course that will never do for practical chips. We obviously need a better encoding.

If we look at the unary number system from a greater distance (now that we know more CS, we can interpret the representations as strings), we see that we are not using a very important feature of strings here: position. As we only have one letter in our alphabet (/), we cannot, so we should use a larger alphabet. The main idea behind a positional number system $\mathcal{N} = \langle D_b, \varphi_b, \psi_b \rangle$ is that we encode numbers as strings of digits (characters in the alphabet D_b), such that the position matters, and to give these encoding a meaning by mapping them into the unary natural numbers via a mapping ψ_b. This is the the same process we did for the logics; we are now doing it for number systems. However, here, we also want to ensure that the meaning mapping ψ_b is a bijection, since we want to define the arithmetics on the encodings by reference to The arithmetical operators on the unary natural numbers.

We can look at this as a bootstrapping process, where the unary natural numbers constitute the seed system we build up everything from.

Just like we did for string codes earlier, we build up the meaning mapping ψ_b on characters from D_b first. To have a chance to make ψ bijective, we insist that the "character code" φ_b is is a bijection from D_b and the first b unary natural numbers. Now we extend φ_b from a character code to a string code, however unlike earlier, we do not use simple concatenation to induce the string code, but a much more complicated function based on the arithmetic operations on unary natural numbers. We will see later[14] that this give us a bijection between ${D_b}^+$ and the unary natural numbers. EdNote:14

Commonly Used Positional Number Systems

▷ **Example 408** The following positional number systems are in common use.

name	set	base	digits	example
unary	$\mathbb{N}_1$	1	/	$/////_1$
binary	$\mathbb{N}_2$	2	0,1	0101000111_2
octal	$\mathbb{N}_8$	8	0,1,...,7	63027_8
decimal	$\mathbb{N}_{10}$	10	0,1,...,9	162098_{10} or 162098
hexadecimal	$\mathbb{N}_{16}$	16	0,1,...,9,A,...,F	$FF3A12_{16}$

▷ **Notation 409** attach the base of $\mathcal{N}$ to every number from $\mathcal{N}$. (default: decimal)

Trick: Group triples or quadruples of binary digits into recognizable chunks (add leading zeros as needed)

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

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

▷ $F3A_{16} = \underbrace{F_{16}}_{1111_2}\underbrace{3_{16}}_{0011_2}\underbrace{A_{16}}_{1010_2} = 111100111010_2$, $4721_8 = \underbrace{4_8}_{100_2}\underbrace{7_8}_{111_2}\underbrace{2_8}_{010_2}\underbrace{1_8}_{001_2} = 100111010001_2$

 239 JACOBS UNIVERSITY

We have all seen positional number systems: our decimal system is one (for the base 10). Other systems that important for us are the binary system (it is the smallest non-degenerate one) and the octal- (base 8) and hexadecimal- (base 16) systems. These come from the fact that binary numbers are very hard for humans to scan. Therefore it became customary to group three or four digits together and introduce we (compound) digits for them. The octal system is mostly relevant for historic reasons, the hexadecimal system is in widespread use as syntactic sugar for binary numbers, which form the basis for circuits, since binary digits can be represented physically by current/no current.

Now that we have defined positional number systems, we want to define the arithmetic operations on the these number representations. We do this by using an old trick in math. If we have an operation $f_T : T \to T$ on a set T and a well-behaved mapping ψ from a set S into T, then we can "pull-back" the operation on f_T to S by defining the operation $f_S : S \to S$ by $f_S(s) := \psi^{-1}(f_T(\psi(s)))$ according to the following diagram.

[14] EdNote: reference

$$
\begin{tikzcd}
S \arrow[rr, bend left, "\psi"] & & T \arrow[ll, bend left, "\psi^{-1}"] \\
S \arrow[u, "f_S = \psi^{-1} \circ f_T \circ \psi"'] \arrow[rr, "\psi"] & & T \arrow[u, "f_T"']
\end{tikzcd}
$$

n Obviously, this construction can be done in any case, where ψ is bijective (and thus has an inverse function). For defining the arithmetic operations on the positional number representations, we do the same construction, but for binary functions (after we have established that ψ is indeed a bijection).

The fact that ψ_b is a bijection a posteriori justifies our notation, where we have only indicated the base of the positional number system. Indeed any two positional number systems are isomorphic: they have bijections ψ_b into the unary natural numbers, and therefore there is a bijection between them.

Arithmetics for PNS

- ▷ **Lemma 410** *Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS, then ψ_b is bijective.*
- ▷ Proof Sketch: Construct ${\psi_b}^{-1}$ by successive division modulo the base of $\mathcal{N}$.
- ▷ Idea: use this to define arithmetics on $\mathcal{N}$.
- ▷ **Definition 411** Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS of base b, then we define a binary function $+_b : \mathbb{N}_b \times \mathbb{N}_b \to \mathbb{N}_b$ by $x +_b y := {\psi_b}^{-1}(\psi_b(x) \oplus \psi_b(y))$.
- ▷ Note: The addition rules (carry chain addition) generalize from the decimal system to general PNS
- ▷ Idea: Do the same for other arithmetic operations. (works like a charm)
- ▷ Future: Concentrate on binary arithmetics. (implement into circuits)

Adders

The next step is now to implement the induced arithmetical operations into combinational circuits, starting with addition. Before we can do this, we have to specify which (Boolean) function we really want to implement. For convenience, we will use the usual decimal (base 10) representations of numbers and their operations to argue about these circuits. So we need conversion functions from decimal numbers to binary numbers to get back and forth. Fortunately, these are easy to come by, since we use the bijections ψ from both systems into the unary natural numbers, which we can compose to get the transformations.

Arithmetic Circuits for Binary Numbers

- ▷ Idea: Use combinational circuits to do basic arithmetics.
- ▷ **Definition 412** Given the (abstract) number $a \in \mathbb{N}$, $B(a)$ denotes from now on the binary representation of a.

  For the opposite case, i.e., the natural number represented by a binary string

$a = \langle a_{n-1}, \ldots, a_0 \rangle \in \mathbb{B}^n$, the notation $\langle\langle a \rangle\rangle$ is used, i.e.,

$$\langle\langle a \rangle\rangle = \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

▷ **Definition 413** An n-bit adder is a circuit computing the function $f^n_{+2} \colon \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^{n+1}$ with

$$f^n_{+2}(a;b) := B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$$

If we look at the definition again, we see that we are again using a pull-back construction. These will pop up all over the place, since they make life quite easy and safe.

Before we actually get a combinational circuit for an n-bit adder, we will build a very useful circuit as a building block: the "half adder" (it will take two to build a full adder).

The Half-Adder

▷ There are different ways to implement an adder. All of them build upon two basic components, the half-adder and the full-adder.

▷ **Definition 414** A half adder is a circuit HA implementing the function f_{HA} in the truth table on the right.

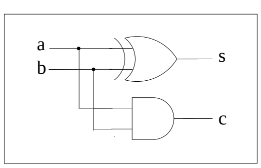
$$f_{\mathsf{HA}} \colon \mathbb{B}^2 \to \mathbb{B}^2 \quad \langle a, b \rangle \mapsto \langle c, s \rangle$$

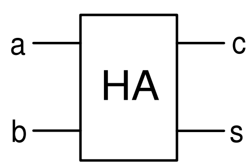
s is called the sum bit and c the carry bit.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

▷ Note: The carry can be computed by a simple AND, i.e., $c = \mathsf{AND}(a,b)$, and the sum bit by a XOR function.

Building and Evaluating the Half-Adder



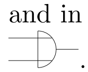


▷ So, the half-adder corresponds to the Boolean function $f_{\mathsf{HA}} \colon \mathbb{B}^2 \to \mathbb{B}^2; \langle a, b \rangle \mapsto \langle a \oplus b, a \wedge b \rangle$

▷ Note: $f_{\mathsf{HA}}(a,b) = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$, i.e., it is indeed an adder.

▷ We count XOR as one gate, so $C(\mathsf{HA}) = 2$ and $\mathrm{dp}(\mathsf{HA}) = 1$.

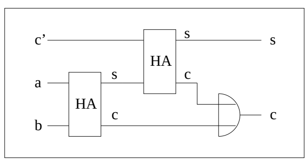
Now that we have the half adder as a building block it is rather simple to arrive at a full adder circuit.

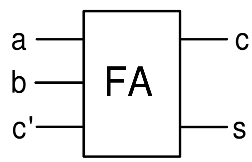
⚠, in the diagram for the full adder, and in the following, we will sometimes use a variant gate symbol for the OR gate: The symbol . It has the same outline as an AND gate, but the input lines go all the way through.

The Full Adder

a	b	c'	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- ▷ **Definition 415** The 1-bit full adder is a circuit FA^1 that implements the function $f^1_{\mathsf{FA}} : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}^2$ with $(\mathsf{FA}^1(a, b, c')) = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$
- ▷ The result of the full-adder is also denoted with $\langle c, s \rangle$, i.e., a carry and a sum bit. The bit c' is called the input carry.
- ▷ the easiest way to implement a full adder is to use two half adders and an OR gate.
- ▷ **Lemma 416 (Cost and Depth)** $C(FA^1) = 2C(HA) + 1 = 5$ *and* $dp(FA^1) = 2dp(HA) + 1 = 3$

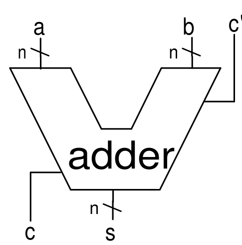




Of course adding single digits is a rather simple task, and hardly worth the effort, if this is all we can do. What we are really after, are circuits that will add n-bit binary natural numbers, so that we arrive at computer chips that can add long numbers for us.

Full n-bit Adder

- ▷ **Definition 417** An n-bit full adder ($n > 1$) is a circuit that corresponds to $f^n_{\mathsf{FA}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n; \langle a, b, c' \rangle \mapsto B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$
- ▷ **Notation 418** We will draw the n-bit full adder with the following symbol in circuit diagrams.



Note that we are abbreviating n-bit input and output edges with a single one that has a slash and the number n next to it.

- ▷ There are various implementations of the full n-bit adder, we will look at two of them

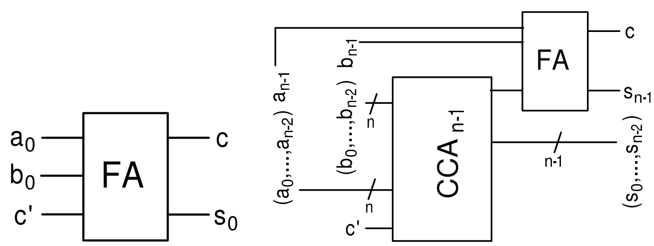
This implementation follows the intuition behind elementary school addition (only for binary numbers): we write the numbers below each other in a tabulated fashion, and from the least significant digit, we follow the process of

- adding the two digits with carry from the previous column
- recording the sum bit as the result, and
- passing the carry bit on to the next column

until one of the numbers ends.

The Carry Chain Adder

▷ The inductively designed circuit of the carry chain adder.



- ▹ $n = 1$: the CCA^1 consists of a full adder
- ▹ $n > 1$: the CCA^n consists of an $(n-1)$-bit carry chain adder CCA^{n-1} and a full adder that sums up the carry of CCA^{n-1} and the last two bits of a and b

▷ **Definition 419** An n-bit carry chain adder CCA^n is inductively defined as

- ▹ $(f^1_{\mathsf{CCA}}(a_0, b_0, c)) = (\mathsf{FA}^1(a_0, b_0, c))$
- ▹ $(f^n_{\mathsf{CCA}}(\langle a_{n-1}, \ldots, a_0\rangle, \langle b_{n-1}, \ldots, b_0\rangle, c')) = \langle c, s_{n-1}, \ldots, s_0\rangle$ with
  - ▹ $\langle c, s_{n-1}\rangle = (\mathsf{FA}^{n-1}(a_{n-1}, b_{n-1}, c_{n-1}))$
  - ▹ $\langle c_{n-1}, \ldots, c_s\rangle 0 = (f^{n-1}_{\mathsf{CCA}}(\langle a_{n-2}, \ldots, a_0\rangle, \langle b_{n-2}, \ldots, b_0\rangle, c'))$

▷ **Lemma 420 (Cost)** $C(CCA^n) \in O(n)$

▷ Proof Sketch: $C(\mathsf{CCA}^n) = C(\mathsf{CCA}^{n-1}) + C(\mathsf{FA}^1) = C(\mathsf{CCA}^{n-1}) + 5 = 5n$

▷ **Lemma 421 (Depth)** $dp(CCA^n) \in O(n)$

▷ Proof Sketch: $\mathsf{dp}(\mathsf{CCA}^n) \leq \mathsf{dp}(\mathsf{CCA}^{n-1}) + \mathsf{dp}(\mathsf{FA}^1) \leq \mathsf{dp}(\mathsf{CCA}^{n-1}) + 3 \leq 3n$

▷ The carry chain adder is simple, but cost and depth are high. (depth is critical (speed))

▷ Question: Can we do better?

▷ Problem: the carry ripples up the chain (upper parts wait for carries from lower part)

 246 JACOBS UNIVERSITY

A consequence of using the carry chain adder is that if we go from a 32-bit architecture to a 64-bit architecture, the speed of additions in the chips would not increase, but decrease (by 50%). Of course, we can carry out 64-bit additions now, a task that would have needed a special routine at the software level (these typically involve at least 4 32-bit additions so there is a speedup for such additions), but most addition problems in practice involve small (under 32-bit) numbers, so we will have an overall performance loss (not what we really want for all that cost).

If we want to do better in terms of depth of an n-bit adder, we have to break the dependency on the carry, let us look at a decimal addition example to get the idea. Consider the following snapshot of an carry chain addition

first summand		3	4	7	9	8	3	4	7	9	2
second summand		$2_?$	$5_?$	$1_?$	$8_?$	$1_?$	$7_?$	8_1	7_1	2_0	1_0
partial sum	?	?	?	?	?	?	?	?	5	1	3

We have already computed the first three partial sums. Carry chain addition would simply go on and ripple the carry information through until the left end is reached (after all what can we do? we need the carry information to carry out left partial sums). Now, if we only knew what the carry would be e.g. at column 5, then we could start a partial summation chain there as well.

The central idea in the "*conditional sum adder*" we will pursue now, is to trade time for space, and just compute both cases (with and without carry), and then later choose which one was the correct one, and discard the other. We can visualize this in the following schema.

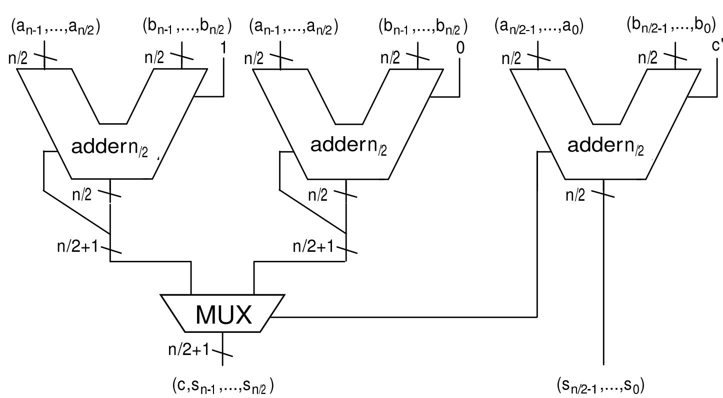
first summand		3	4	7	9	8	3	4	7	9	2
second summand		$2_?$	5_0	1_1	$8_?$	$1_?$	$7_?$	8_1	7_1	2_0	1_0
lower sum							?	?	5	1	3
upper sum. with carry	?	?	?	9	8	0					
upper sum. no carry	?	?	?	9	7	9					

Here we start at column 10 to compute the lower sum, and at column 6 to compute two upper sums, one with carry, and one without. Once we have fully computed the lower sum, we will know about the carry in column 6, so we can simply choose which upper sum was the correct one and combine lower and upper sum to the result.

Obviously, if we can compute the three sums in parallel, then we are done in only five steps not ten as above. Of course, this idea can be iterated: the upper and lower sums need not be computed by carry chain addition, but can be computed by conditional sum adders as well.

The Conditional Sum Adder

- ▷ Idea: pre-compute both possible upper sums (e.g. upper half) for carries 0 and 1, then choose (via MUX) the right one according to lower sum.
- ▷ the inductive definition of the circuit of a conditional sum adder (CSA).



- ▷ **Definition 422** An n-bit conditional sum adder CSA^n is recursively defined as
  - ▷ $(f^n_{\mathsf{CSA}}(\langle a_{n-1},\ldots,a_0\rangle,\langle b_{n-1},\ldots,b_0\rangle,c')) = \langle c,s_{n-1},\ldots,s_0\rangle$ where
    - ▷ $\langle c_{n/2},s_{n/2-1},\ldots,s_0\rangle = (f^{n/2}_{\mathsf{CSA}}(\langle a_{n/2-1},\ldots,a_0\rangle,\langle b_{n/2-1},\ldots,b_0\rangle,c'))$
    - ▷ $\langle c,s_{n-1},\ldots,s_{n/2}\rangle = \begin{cases} (f^{n/2}_{\mathsf{CSA}}(\langle a_{n-1},\ldots,a_{n/2}\rangle,\langle b_{n-1},\ldots,b_{n/2}\rangle,0)) & \text{if } c_{n/2}=0 \\ (f^{n/2}_{\mathsf{CSA}}(\langle a_{n-1},\ldots,a_{n/2}\rangle,\langle b_{n-1},\ldots,b_{n/2}\rangle,1)) & \text{if } c_{n/2}=1 \end{cases}$
    - ▷ $(f^1_{\mathsf{CSA}}(a_0,b_0,c)) = (\mathsf{FA}^1(a_0,b_0,c))$

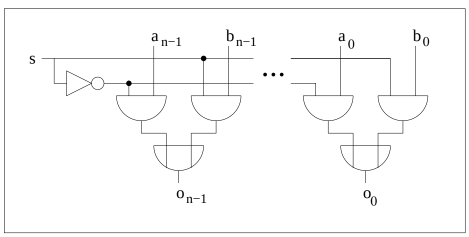
The only circuit that we still have to look at is the one that chooses the correct upper sums. Fortunately, this is a rather simple design that makes use of the classical trick that “if C, then A, else B” can be expressed as “(C and A) or (not C and B)”.

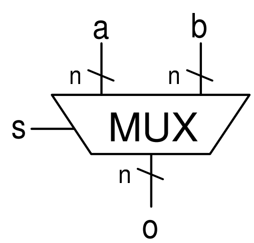
The Multiplexer

▷ **Definition 423** An n-bit multiplexer MUX^n is a circuit which implements the function $f^n_{\mathsf{MUX}} \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B}^n$ with

$$f(a_{n-1},\ldots,a_0,b_{n-1},\ldots,b_0,s) = \begin{cases} \langle a_{n-1},\ldots,a_0\rangle & \text{if } s=0 \\ \langle b_{n-1},\ldots,b_0\rangle & \text{if } s=1 \end{cases}$$

▷ Idea: A multiplexer chooses between two n-bit input vectors A and B depending on the value of the control bit s.





▷ Cost and depth: $C(\mathsf{MUX}^n) = 3n+1$ and $\mathsf{dp}(\mathsf{MUX}^n) = 3$.

Now that we have completely implemented the conditional lookahead adder circuit, we can analyze it for its cost and depth (to see whether we have really made things better with this design). Analyzing the depth is rather simple, we only have to solve the recursive equation that combines the recursive call of the adder with the multiplexer. Conveniently, the 1-bit full adder has the same depth as the multiplexer.

The Depth of CSA

▷ $\mathsf{dp}(\mathsf{CSA}^n) \leq \mathsf{dp}(\mathsf{CSA}^{n/2}) + \mathsf{dp}(\mathsf{MUX}^{n/2+1})$

▷ solve the recursive equation:

$$\begin{aligned}
\mathsf{dp}(\mathsf{CSA}^n) &\leq \mathsf{dp}(\mathsf{CSA}^{n/2}) + \mathsf{dp}(\mathsf{MUX}^{n/2+1}) \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/2}) + 3 \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/4}) + 3 + 3 \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/8}) + 3 + 3 + 3 \\
&\quad \ldots \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n2^{-i}}) + 3i \\
&\leq \mathsf{dp}(\mathsf{CSA}^1) + 3\log_2(n) \\
&\leq 3\log_2(n) + 3
\end{aligned}$$

The analysis for the cost is much more complex, we also have to solve a recursive equation, but a more difficult one. Instead of just guessing the correct closed form, we will use the opportunity to show a more general technique: using Master's theorem for recursive equations. There are many similar theorems which can be used in situations like these, going into them or proving Master's theorem would be beyond the scope of the course.

The Cost of CSA

▷ $C(\mathsf{CSA}^n) = 3C(\mathsf{CSA}^{n/2}) + C(\mathsf{MUX}^{n/2+1})$.

▷ Problem: How to solve this recursive equation?

▷ Solution: Guess a closed formula, prove by induction. (if we are lucky)

▷ Solution2: Use a general tool for solving recursive equations.

▷ **Theorem 424 (Master's Theorem for Recursive Equations)** *Given the recursively defined function $f \colon \mathbb{N} \to \mathbb{R}$, such that $f(1) = c \in \mathbb{R}$ and $f(b^k) = af(b^{k-1}) + g(b^k)$ for some $a \in \mathbb{R}$, $1 \leq a$, $k \in \mathbb{N}$, and $g \colon \mathbb{N} \to \mathbb{R}$, then $f(b^k) = ca^k + \sum_{i=0}^{k-1} a^i g(b^{k-i})$*

▷ We have $C(\mathsf{CSA}^n) = 3C(\mathsf{CSA}^{n/2}) + C(\mathsf{MUX}^{n/2+1}) = 3C(\mathsf{CSA}^{n/2}) + 3(n/2+1) + 1 = 3C(\mathsf{CSA}^{n/2}) + \frac{3}{2}n + 4$

▷ So, $C(\mathsf{CSA}^n)$ is a function that can be handled via Master's theorem with $a = 3$, $b = 2$, $n = b^k$, $g(n) = 3/2n + 4$, and $c = C(f^1_{\mathsf{CSA}}) = C(\mathsf{FA}^1) = 5$

▷ thus $C(\mathsf{CSA}^n) = 5 \cdot 3^{\log_2(n)} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4$

▷ Note: $a^{\log_2(n)} = 2^{\log_2(a)^{\log_2(n)}} = 2^{\log_2(a) \cdot \log_2(n)} = 2^{\log_2(n)^{\log_2(a)}} = n^{\log_2(a)}$

$$\begin{aligned}
C(\mathsf{CSA}^n) &= 5 \cdot 3^{\log_2(n)} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4 \\
&= 5n^{\log_2(3)} + \sum_{i=1}^{\log_2(n)} n\frac{3}{2}^i n + 4 \\
&= 5n^{\log_2(3)} + n \cdot \sum_{i=1}^{\log_2(n)} \frac{3}{2}^i + 4\log_2(n) \\
&= 5n^{\log_2(3)} + 2n \cdot \frac{3}{2}^{\log_2(n)+1} - 1 + 4\log_2(n) \\
&= 5n^{\log_2(3)} + 3n \cdot n^{\log_2(\frac{3}{2})} - 2n + 4\log_2(n) \\
&= 8n^{\log_2(3)} - 2n + 4\log_2(n) \in O(n^{\log_2(3)})
\end{aligned}$$

▷ **Theorem 425** *The cost and the depth of the conditional sum adder are in the following complexity classes:*

$$C(\mathit{CSA}^n) \in O(n^{\log_2(3)}) \qquad dp(\mathit{CSA}^n) \in O(\log_2(n))$$

▷ Compare with: $C(\mathsf{CCA}^n) \in O(n)$ $\quad$ $\mathrm{dp}(\mathsf{CCA}^n) \in O(n)$

▷ So, the conditional sum adder has a smaller depth than the carry chain adder. This smaller depth is paid with higher cost.

▷ There is another adder that combines the small cost of the carry chain adder with the low depth of the conditional sum adder. This carry lookahead adder CLA^n has a cost $C(\mathsf{CLA}^n) \in O(n)$ and a depth of $\mathrm{dp}(\mathsf{CLA}^n) \in O(\log_2(n))$.

Instead of perfecting the n-bit adder further (and there are lots of designs and optimizations out there, since this has high commercial relevance), we will extend the range of arithmetic operations. The next thing we come to is subtraction.

3.2.2 Arithmetics for Two's Complement Numbers

This of course presents us with a problem directly: the n-bit binary natural numbers, we have used for representing numbers are closed under addition, but not under subtraction: If we have two n-bit binary numbers $B(n)$, and $B(m)$, then $B(n+m)$ is an $n+1$-bit binary natural number. If we count the most significant bit separately as the carry bit, then we have a n-bit result. For subtraction this is not the case: $B(n-m)$ is only a n-bit binary natural number, if $m \geq n$ (whatever we do with the carry). So we have to think about representing negative binary natural numbers first. It turns out that the solution using sign bits that immediately comes to mind is not the best one.

Negative Numbers and Subtraction

▷ Note: So far we have completely ignored the existence of negative numbers.

▷ Problem: Subtraction is a partial operation without them.

▷ Question: Can we extend the binary number systems for negative numbers?

▷ Simple Solution: Use a sign bit. (additional leading bit that indicates whether the number is positive)

▷ **Definition 426 ($(n+1)$-bit signed binary number system)**

$$\langle\langle a_n, \ldots, a_0 \rangle\rangle^- := \begin{cases} \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle & \text{if } a_n = 0 \\ -\langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle & \text{if } a_n = 1 \end{cases}$$

▷ Note: We need to fix string length to identify the sign bit. (leading zeroes)

▷ **Example 427** In the 8-bit signed binary number system

- ▷ 10011001 represents -25 $((\langle\langle 10011001 \rangle\rangle^-) = -(2^4 + 2^3 + 2^0))$
- ▷ 00101100 corresponds to a positive number: 44

Here we did the naive solution, just as in the decimal system, we just added a sign bit, which specifies the polarity of the number representation. The first consequence of this that we have to keep in mind is that we have to fix the width of the representation: Unlike the representation for binary natural numbers which can be arbitrarily extended to the left, we have to know which bit is the sign bit. This is not a big problem in the world of combinational circuits, since we have a fixed width of input/output edges anyway.

Problems of Sign-Bit Systems

- Generally: An n-bit signed binary number system allows to represent the integers from $-2^{n-1}+1$ to $+2^{n-1}-1$.
- $2^{n-1}-1$ positive numbers, $2^{n-1}-1$ negative numbers, and the zero
- Thus we represent $\#(\{\langle\!\langle s\rangle\!\rangle^- \mid s\in\mathbb{B}^n\}) = 2\cdot(2^{n-1}-1)+1 = 2^n-1$ numbers all in all
- One number must be represented twice (But there are 2^n strings of length n.)
- $10\ldots0$ and $00\ldots0$ both represent the zero as $-1\cdot 0 = 1\cdot 0$.

signed binary				$\mathbb{Z}$
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	0	0	0	-0
1	0	0	1	-1
1	0	1	0	-2
1	0	1	1	-3
1	1	0	0	-4
1	1	0	1	-5
1	1	1	0	-6
1	1	1	1	-7

- We could build arithmetic circuits using this, but there is a more elegant way!

 252 JACOBS UNIVERSITY

All of these problems could be dealt with in principle, but together they form a nuisance, that at least prompts us to look for something more elegant. The two's complement representation also uses a sign bit, but arranges the lower part of the table in the last slide in the opposite order, freeing the negative representation of the zero. The technical trick here is to use the sign bit (we still have to take into account the width n of the representation) not as a mirror, but to translate the positive representation by subtracting 2^n.

The Two's Complement Number System

- **Definition 428** Given the binary string $a=\langle a_n,\ldots,a_0\rangle \in \mathbb{B}^{n+1}$, where $n>1$. The integer represented by a in the $(n+1)$-bit two's complement, written as $\langle\!\langle a\rangle\!\rangle_n^{2s}$, is defined as

$$\begin{aligned}\langle\!\langle a\rangle\!\rangle_n^{2s} &= -a_n\cdot 2^n + \langle\!\langle a_{n-1},\ldots,a_0\rangle\!\rangle\\ &= -a_n\cdot 2^n + \sum_{i=0}^{n-1} a_i\cdot 2^i\end{aligned}$$

- **Notation 429** Write $B_n^{2s}(z)$ for the binary string that represents z in the two's complement number system, i.e., $\langle\!\langle B_n^{2s}(z)\rangle\!\rangle_n^{2s}=z$.

2's compl.				$\mathbb{Z}$
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

 253 JACOBS UNIVERSITY

We will see that this representation has much better properties than the naive sign-bit representation we experimented with above. The first set of properties are quite trivial, they just formalize the intuition of moving the representation down, rather than mirroring it.

Properties of Two's Complement Numbers (TCN)

- ▷ Let $b = \langle b_n, \ldots, b_0 \rangle$ be a number in the $n+1$-bit two's complement system, then
- ▷ Positive numbers and the zero have a sign bit 0, i.e., $b_n = 0 \Leftrightarrow (\langle\!\langle b \rangle\!\rangle_n^{2s} \geq 0)$.
- ▷ Negative numbers have a sign bit 1, i.e., $b_n = 1 \Leftrightarrow \langle\!\langle b \rangle\!\rangle_n^{2s} < 0$.
- ▷ For positive numbers, the two's complement representation corresponds to the normal binary number representation, i.e., $b_n = 0 \Leftrightarrow \langle\!\langle b \rangle\!\rangle_n^{2s} = \langle\!\langle b \rangle\!\rangle$
- ▷ There is a unique representation of the number zero in the n-bit two's complement system, namely $B_n^{2s}(0) = \langle 0, \ldots, 0 \rangle$.
- ▷ This number system has an asymmetric range $\mathcal{R}_n^{2s} := \{-2^n, \ldots, 2^n - 1\}$.

The next property is so central for what we want to do, it is upgraded to a theorem. It says that the mirroring operation (passing from a number to it's negative sibling) can be achieved by two very simple operations: flipping all the zeros and ones, and incrementing.

The Structure Theorem for TCN

- ▷ **Theorem 430** *Let $a \in \mathbb{B}^{n+1}$ be a binary string, then $-\langle\!\langle a \rangle\!\rangle_n^{2s} = \langle\!\langle \overline{a} \rangle\!\rangle_n^{2s} + 1$, where $\overline{a}$ is the pointwise bit complement of a.*
- ▷ Proof Sketch: By calculation using the definitions:

$$
\begin{aligned}
\langle\!\langle \overline{a_n}, \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle_n^{2s} &= -\overline{a_n} \cdot 2^n + \langle\!\langle \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle \\
&= \overline{a_n} \cdot -2^n + \sum_{i=0}^{n-1} \overline{a_i} \cdot 2^i \\
&= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 1 - a_i \cdot 2^i \\
&= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= -2^n + a_n \cdot 2^n + 2^{n-1} - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle \\
&= (-2^n + 2^n) + a_n \cdot 2^n - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle - 1 \\
&= -(a_n \cdot -2^n + \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle) - 1 \\
&= -\langle\!\langle a \rangle\!\rangle_n^{2s} - 1
\end{aligned}
$$

A first simple application of the TCN structure theorem is that we can use our existing conversion routines (for binary natural numbers) to do TCN conversion (for integers).

Application: Converting from and to TCN?

- ▷ to convert an integer $-z \in \mathbb{Z}$ with $z \in \mathbb{N}$ into an n-bit TCN

- ▷ generate the n-bit binary number representation $B(z) = \langle b_{n-1}, \ldots, b_0 \rangle$
  - ▷ complement it to $\overline{B(z)}$, i.e., the bitwise negation $\overline{b_i}$ of $B(z)$
  - ▷ increment (add 1) $\overline{B(z)}$, i.e. compute $B(\langle\langle \overline{B(z)} \rangle\rangle + 1)$
- ▷ to convert a negative n-bit TCN $b = \langle b_{n-1}, \ldots, b_0 \rangle$, into an integer
  - ▷ decrement b, (compute $B(\langle\langle b \rangle\rangle - 1)$)
  - ▷ complement it to $\overline{B(\langle\langle b \rangle\rangle - 1)}$
  - ▷ compute the decimal representation and negate it to $-\langle\langle \overline{B(\langle\langle b \rangle\rangle - 1)} \rangle\rangle$

Subtraction and Two's Complement Numbers

- ▷ Idea: With negative numbers use our adders directly
- ▷ **Definition 431** An n-bit subtracter is a circuit that implements the function $f^n_{\mathsf{SUB}} \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n$ such that

$$f^n_{\mathsf{SUB}}(a, b, b') = B^{\mathsf{2s}}_n(\langle\langle a \rangle\rangle^{\mathsf{2s}}_n - \langle\langle b \rangle\rangle^{\mathsf{2s}}_n - b')$$

  for all $a, b \in \mathbb{B}^n$ and $b' \in \mathbb{B}$. The bit b' is called the input borrow bit.

- ▷ Note: We have $\langle\langle a \rangle\rangle^{\mathsf{2s}}_n - \langle\langle b \rangle\rangle^{\mathsf{2s}}_n = \langle\langle a \rangle\rangle^{\mathsf{2s}}_n + (-\langle\langle b \rangle\rangle^{\mathsf{2s}}_n) = \langle\langle a \rangle\rangle^{\mathsf{2s}}_n + \langle\langle \overline{b} \rangle\rangle^{\mathsf{2s}}_n + 1$
- ▷ Idea: Can we implement an n-bit subtracter as $f^n_{\mathsf{SUB}}(a, b, b') = (\mathsf{FA}^n(a, \overline{b}, \overline{b'}))$?
- ▷ not immediately: We have to make sure that the full adder plays nice with twos complement numbers

In addition to the unique representation of the zero, the two's complement system has an additional important property. It is namely possible to use the adder circuits introduced previously without any modification to add integers in two's complement representation.

Addition of TCN

- ▷ Idea: use the adders without modification for TCN arithmetic
- ▷ **Definition 432** An n-bit two's complement adder $(n > 1)$ is a circuit that corresponds to the function $f^n_{\mathsf{TCA}} \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n$, such that $f^n_{\mathsf{TCA}}(a, b, c') = B^{\mathsf{2s}}_n(\langle\langle a \rangle\rangle^{\mathsf{2s}}_n + \langle\langle b \rangle\rangle^{\mathsf{2s}}_n + c')$ for all $a, b \in \mathbb{B}^n$ and $c' \in \mathbb{B}$.
- ▷ **Theorem 433** $f^n_{TCA} = f^n_{FA}$ *(first prove some Lemmas)*

It is not obvious that the same circuits can be used for the addition of binary and two's complement numbers. So, it has to be shown that the above function $TCAcircFNn$ and the full adder function f^n_{FA} from definition**??** are identical. To prove this fact, we first need the following lemma stating that a $(n+1)$-bit two's complement number can be generated from a n-bit two's complement number without changing its value by duplicating the sign-bit:

TCN Sign Bit Duplication Lemma

▷ Idea: An $n+1$-bit TCN can be generated from a n-bit TCN without changing its value by duplicating the sign-bit.

▷ **Lemma 434** *Let* $a = \langle a_n, \ldots, a_0 \rangle \in \mathbb{B}^{n+1}$ *be a binary string, then* $\langle\langle a_n, \ldots, a_0 \rangle\rangle_{n+1}^{2s} = \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle_n^{2s}$.

▷ Proof Sketch: By calculation:

$$\begin{aligned}
\langle\langle a_n, \ldots, a_0 \rangle\rangle_{n+1}^{2s} &= -a_n \cdot 2^{n+1} + \langle\langle a_n, \ldots, a_0 \rangle\rangle \\
&= -a_n \cdot 2^{n+1} + a_n \cdot 2^n + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= a_n \cdot (-2^{n+1} + 2^n) + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= a_n \cdot (-2 \cdot 2^n + 2^n) + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= -a_n \cdot 2^n + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle_n^{2s}
\end{aligned}$$

We will now come to a major structural result for two's complement numbers. It will serve two purposes for us:

1. It will show that the same circuits that produce the sum of binary numbers also produce proper sums of two's complement numbers.
2. It states concrete conditions when a valid result is produced, namely when the last two carry-bits are identical.

The TCN Main Theorem

▷ **Definition 435** Let $a, b \in \mathbb{B}^{n+1}$ and $c \in \mathbb{B}$ with $a = \langle a_n, \ldots, a_0 \rangle$ and $b = \langle b_n, \ldots, b_0 \rangle$, then we call $(\mathrm{ic}_k(a, b, c))$, the k-th intermediate carry of a, b, and c, iff

$$\langle\langle \mathrm{ic}_k(a, b, c), s_{k-1}, \ldots, s_0 \rangle\rangle = \langle\langle a_{k-1}, \ldots, a_0 \rangle\rangle + \langle\langle b_{k-1}, \ldots, b_0 \rangle\rangle + c$$

for some $s_i \in \mathbb{B}$.

▷ **Theorem 436** *Let* $a, b \in \mathbb{B}^n$ *and* $c \in \mathbb{B}$*, then*

1. $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \in \mathcal{R}_n^{2s}$*, iff* $(\mathrm{ic}_{n+1}(a, b, c)) = (\mathrm{ic}_n(a, b, c))$.
2. *If* $(\mathrm{ic}_{n+1}(a, b, c)) = (\mathrm{ic}_n(a, b, c))$*, then* $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = \langle\langle s \rangle\rangle_n^{2s}$*, where* $\langle\langle \mathrm{ic}_{n+1}(a, b, c), s_n, \ldots, s_0 \rangle\rangle = \langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + c$.

Unfortunately, the proof of this attractive and useful theorem is quite tedious and technical

Proof of the TCN Main Theorem

Proof: Let us consider the sign-bits a_n and b_n separately from the value-bits $a' = \langle a_{n-1}, \ldots, a_0 \rangle$ and $b' = \langle b_{n-1}, \ldots, b_0 \rangle$.

P.1 Then

$$\begin{aligned}
\langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c &= \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle + \langle\langle b_{n-1}, \ldots, b_0 \rangle\rangle + c \\
&= \langle\langle \mathrm{ic}_n(a, b, c), s_{n-1}, \ldots, s_0 \rangle\rangle
\end{aligned}$$

and $a_n + b_n + (\mathsf{ic}_n(a,b,c)) = \langle\!\langle \mathsf{ic}_{n+1}(a,b,c), s_n \rangle\!\rangle$.

We have to consider three cases

P.2.2.1 $a_n = b_n = 0$:

P.2.1.1 $\langle\!\langle a \rangle\!\rangle_n^{2s}$ and $\langle\!\langle b \rangle\!\rangle_n^{2s}$ are both positive, so $(\mathsf{ic}_{n+1}(a,b,c)) = 0$ and furthermore

$$\begin{aligned}(\mathsf{ic}_n(a,b,c)) = 0 \quad &\Leftrightarrow \quad \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow \quad \langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c \leq 2^n - 1\end{aligned}$$

P.2.1.2 Hence,

$$\begin{aligned}\langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c &= \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \\ &= \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= \langle\!\langle 0, s_{n-1}, \ldots, s_0 \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle_n^{2s}\end{aligned}$$

$\square$

P.2.2 $a_n = b_n = 1$:

P.2.2.1 $\langle\!\langle a \rangle\!\rangle_n^{2s}$ and $\langle\!\langle b \rangle\!\rangle_n^{2s}$ are both negative, so $(\mathsf{ic}_{n+1}(a,b,c)) = 1$ and furthermore $(\mathsf{ic}_n(a,b,c)) = 1$, iff $\langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \geq 2^n$, which is the case, iff $\langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c = -2^{n+1} + \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \geq -2^n$

P.2.2.2 Hence,

$$\begin{aligned}\langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c &= -2^n + \langle\!\langle a' \rangle\!\rangle + -2^n + \langle\!\langle b' \rangle\!\rangle + c \\ &= -2^{n+1} + \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \\ &= -2^{n+1} + \langle\!\langle 1, s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= -2^n + \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= \langle\!\langle s \rangle\!\rangle_n^{2s}\end{aligned}$$

$\square$

P.2.3 $a_n \neq b_n$:

P.2.3.1 Without loss of generality assume that $a_n = 0$ and $b_n = 1$. (then $(\mathsf{ic}_{n+1}(a,b,c)) = (\mathsf{ic}_n(a,b,c))$)

P.2.3.2 Hence, the sum of $\langle\!\langle a \rangle\!\rangle_n^{2s}$ and $\langle\!\langle b \rangle\!\rangle_n^{2s}$ is in the admissible range $\mathcal{R}_n^{2s}$ as

$$\langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c = \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c - 2^n$$

and $(0 \leq \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \leq 2^{n+1} - 1)$

P.2.3.3 So we have

$$\begin{aligned}\langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle b \rangle\!\rangle_n^{2s} + c &= -2^n + \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \\ &= -2^n + \langle\!\langle \mathsf{ic}_n(a,b,c), s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= -(1 - (\mathsf{ic}_n(a,b,c))) \cdot 2^n + \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= \langle\!\langle \overline{\mathsf{ic}_n(a,b,c)}, s_{n-1}, \ldots, s_0 \rangle\!\rangle_n^{2s}\end{aligned}$$

P.2.3.4 Furthermore, we can conclude that $\langle\!\langle \overline{\mathsf{ic}_n(a,b,c)}, s_{n-1}, \ldots, s_0 \rangle\!\rangle_n^{2s} = \langle\!\langle s \rangle\!\rangle_n^{2s}$ as $s_n = a_n \oplus b_n \oplus (\mathsf{ic}_n(a,b,c)) = 1 \oplus (\mathsf{ic}_n(a,b,c)) = \overline{\mathsf{ic}_n(a,b,c)}$. $\square$

Thus we have considered all the cases and completed the proof. $\square$

The Main Theorem for TCN again

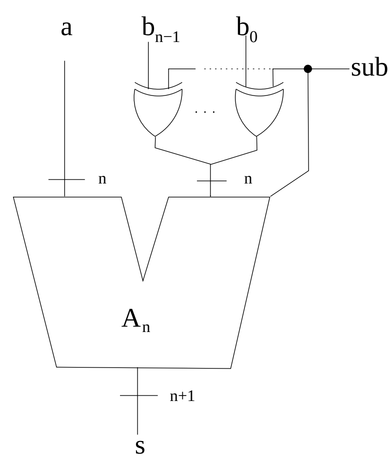
P.3 ▷ Given two $(n+1)$-bit two's complement numbers a and b. The above theorem tells us that the result s of an $(n+1)$-bit adder is the proper sum in two's complement representation iff the last two carries are identical.

▷ If not, a and b were too large or too small. In the case that s is larger than $2^n - 1$, we say that an overflow occurred.In the opposite error case of s being smaller than -2^n, we say that an underflow occurred.

3.2.3 Towards an Algorithmic-Logic Unit

The most important application of the main TCN theorem is that we can build a combinational circuit that can add and subtract (depending on a control bit). This is actually the first instance of a concrete programmable computation device we have seen up to date (we interpret the control bit as a program, which changes the behavior of the device). The fact that this is so simple, it only runs two programs should not deter us; we will come up with more complex things later.

Building an Add/Subtract Unit



▷ Idea: Build a Combinational Circuit that can add and subtract (sub $= 1 \leadsto$ subtract)

▷ If sub $= 0$, then the circuit acts like an adder $(a \oplus 0 = a)$

▷ If sub $= 1$, let $S := \langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle_n^{2s} + 1$ $(a \oplus 0 = 1 - a)$

▷ For $s \in \mathcal{R}_n^{2s}$ the TCN main theorem and the TCN structure theorem together guarantee

$$\begin{aligned} s &= \langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle_n^{2s} + 1 \\ &= \langle\!\langle a \rangle\!\rangle_n^{2s} - \langle\!\langle b \rangle\!\rangle_n^{2s} - 1 + 1 \end{aligned}$$

▷ Summary: We have built a combinational circuit that can perform 2 arithmetic operations depending on a control bit.

▷ Idea: Extend this to a arithmetic logic unit (ALU) with more operations ($+$, -, *, /, n-AND, n-OR,...)

 263

In fact extended variants of the very simple Add/Subtract unit are at the heart of any computer. These are called arithmetic logic units.

3.3 Sequential Logic Circuits and Memory Elements

So far we have only considered combinational logic, i.e. circuits for which the output depends only on the inputs. In such circuits, the output is just a combination of the inputs, and they can be modelde as acyclic labled graphs as we have so far. In many instances it is desirable to have the next output depend on the current output. This allows circuits to represent state as we will see; the price we pay for this is that we have to consider cycles in the underlying graphs. In this sectionwe will first look at sequential circuits in general and at flipflop as stateful circuits in particular. Then go briefly discuss how to combine flipflops into random access memory banks.

3.3.1 Sequential Logic Circuits

Sequential Logic Circuits

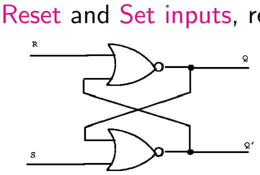
- ▷ In combinational circuits, outputs only depend on inputs (no state)
- ▷ We have disregarded all timing issues (except for favoring shallow circuits)
- ▷ **Definition 437** Circuits that remember their current output or state are often called sequential logic circuits.
- ▷ **Example 438** A *counter*, where the next number to be output is determined by the current number stored.
- ▷ Sequential logic circuits need some ability to store the current state

Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits. We will investigate basic circuits that have the ability to store bits of data. We will start with the simplest possible memory element, and develop more elaborate versions from it.

The circuit we are about to introduce is the simplest circuit that can keep a state, and thus act as a (precursor to) a storage element. Note that we are leaving the realm of acyclic graphs here. Indeed storage elements cannot be realized with combinational circuits as defined above.

RS Flip-Flop

- ▷ **Definition 439** A RS-flipflop (or RS-latch)is constructed by feeding the outputs of two NOR gates back to the other NOR gates input. The inputs R and S are referred to as the Reset and Set inputs, respectively.



R	S	Q	Q'	Comment
0	1	1	0	Set
1	0	0	1	Reset
0	0	Q	Q'	Hold state
1	1	?	?	Avoid

- ▷ Note: the output Q' is simply the inverse of Q. (supplied for convenience)
- ▷ Note: An RS flipflop can also be constructed from NAND gates.

↓	T	F
0	1	0
1	0	0

To understand the operation of the RS-flipflop we first remind ourselves of the truth table of the NOR gate on the right: If one of the inputs is 1, then the output is 0, irrespective of the other. To understand the RS-flipflop, we will go through the input combinations summarized in the table above in detail. Consider the following scenarios:

$S = 1$ **and** $R = 0$ The output of the bottom NOR gate is 0, and thus $Q' = 0$ irrespective of the other input. So both inputs to the top NOR gate are 0, thus, $Q = 1$. Hence, the input combination $S = 1$ and $R = 0$ leads to the flipflop being *set* to $Q = 1$.

$S = 0$ **and** $R = 1$ The argument for this situation is symmetric to the one above, so the outputs become $Q = 0$ and $Q' = 1$. We say that the flipflop is *reset*.

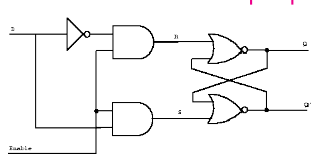
$S = 0$ **and** $R = 0$ Assume the flipflop is set ($Q = 1$ and $Q' = 0$), then the output of the top NOR gate remains at $Q = 1$ and the bottom NOR gate stays at $Q' = 0$. Similarly, when the flipflop is in a reset state ($Q = 0$ and $Q' = 1$), it will remain there with this input combination. Therefore, with inputs $S = 0$ and $R = 0$, the flipflop remains in its state.

$S = 1$ **and** $R = 1$ This input combination will be avoided, we have all the functionality (*set*, *reset*, and *hold*) we want from a memory element.

An RS-flipflop is rarely used in actual sequential logic. However, it is the fundamental building block for the very useful D-flipflop.

The D-Flipflop: the simplest memory device

- ▷ Recap: A RS-flipflop can store a state (set Q to 1 or reset Q to 0)
- ▷ Problem: We would like to have a single data input and avoid $R = S$ states.
- ▷ Idea: Add interface logic to do just this
- ▷ **Definition 440** A D-flipflop is an RS-flipflop with interface logic as below.



E	D	R	S	Q	Comment
1	1	0	1	1	set Q to 1
1	0	1	0	0	reset Q to 0
0	D	0	0	Q	hold Q

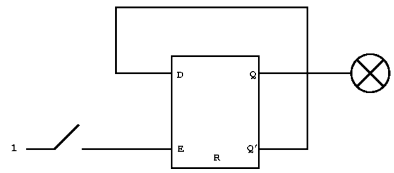
The inputs D and E are called the data and enable inputs.

- ▷ When $E = 1$ the value of D determines the value of the output Q, when E returns to 0, the most recent input D is "remembered."

Sequential logic circuits are constructed from memory elements and combinational logic gates. The introduction of the memory elements allows these circuits to remember their state. We will illustrate this through a simple example.

Example: On/Off Switch

- ▷ Problem: Pushing a button toggles a LED between on and off. (first push switches the LED on, second push off, ...)
- ▷ Idea: Use a D-flipflop (to remember whether the LED is currently on or off) connect its Q' output to its D input (next state is inverse of current state)



In the on/off circuit, the external inputs (buttons) were connected to the E input.

Definition 441 Such circuits are often called asynchronous as they keep track of events that occur at arbitrary instants of time, synchronous circuits in contrast operate on a periodic basis and the Enable input is connected to a common clock signal.

3.3.2 Random Access Memory

We will now discuss how single memory cells (D-flipflops) can be combined into larger structures that can be addressed individually. The name "random access memory" highlights individual addressability in contrast to other forms of memory, e.g. magnetic tapes that can only be read sequentially (i.e. one memory cell after the other).

Random Access Memory Chips

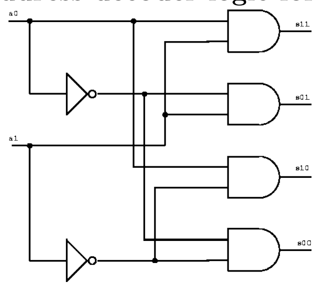
- *Random access memory* (RAM) is used for storing a large number of bits.
- RAM is made up of storage elements similar to the D-flipflops we discussed.
- Principally, each storage element has a unique number or address represented in binary form.
- When the address of the storage element is provided to the RAM chip, the corresponding memory element can be written to or read from.
- We will consider the following questions:
  - What is the physical structure of RAM chips?
  - How are addresses used to select a particular storage element?
  - What do individual storage elements look like?
  - How is reading and writing distinguished?

So the main topic here is to understand the logic of addressing; we need a circuit that takes as input an "address" – e.g. the number of the D-flipflop d we want to address – and data-input and enable inputs and route them through to d.

Address Decoder Logic

- Idea: Need a circuit that activates the storage element given the binary address:
  - At any time, only 1 output line is "on" and all others are off.
  - The line that is "on" specifies the desired element
- **Definition 442** The n-bit address decoder ADL^n has a n inputs and 2^n outputs. $f^m_{\mathsf{ADL}}(a) = \langle b_1, \ldots, b_{2^n}\rangle$, where $b_i = 1$, iff $i = \langle\!\langle a \rangle\!\rangle$.

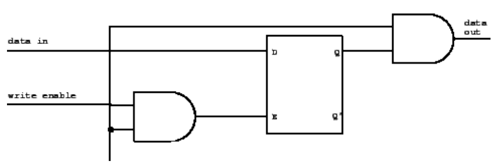
▷ **Example 443 (Address decoder logic for 2-bit addresses)**



Now we can combine an n-bit address decoder as sketched by the example above, with n D-flipflops to get a RAM element.

Storage Elements

- ▷ Idea (Input): Use a D-flipflop connect its E input to the ADL output. Connect the D-input to the common RAM data input line. (input only if addressed)
- ▷ Idea (Output): Connect the flipflop output to common RAM output line. But first AND with ADL output (output only if addressed)
- ▷ Problem: The read process should leave the value of the gate unchanged.
- ▷ Idea: Introduce a "write enable" signal (protect data during read) AND it with the ADL output and connect it to the flipflop's E input.
- ▷ **Definition 444** A Storage Element is given by the following diagram



So we have arrived at a solution for the problem how to make random access memory. In keeping with an introductory course, this the exposition above only shows a "solution in principle"; as RAM storage elements are crucial parts of computers that are produced by the billions, a great deal of engineering has been invested into their design, and as a consequence our solution above is not exactly what we actually have in our laptops nowadays.

Remarks: Actual Storage Elements

- ▷ The storage elements are often simplified to reduce the number of transistors.
- ▷ For example, with care one can replace the flipflop by a capacitor.
- ▷ Also, with large memory chips it is not feasible to connect the data input and output and write enable lines directly to all storage elements.
- ▷ Also, with care one can use the same line for data input and data output.
- ▷ Today, multi-gigabyte RAM chips are on the market.

▷ The capacity of RAM chips doubles approximately every year.

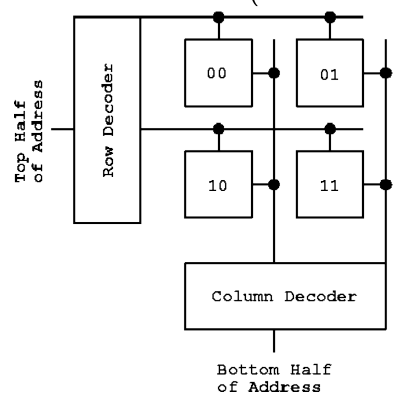
One aspect of this is particularly interesting – and user-visible in the sense that the division of storage addresses is divided into a high- and low part of the address. So we we will briefly discuss it here.

Layout of Memory Chips

▷ To take advantage of the two-dimensional nature of the chip, storage elements are arranged on a square grid. (columns and rows of storage elements)

▷ For example, a 1 Megabit RAM chip has of 1024 rows and 1024 columns.

▷ identify storage element by its row and column "coordinates". (AND them for addressing)



▷ Hence, to select a particular storage location the address information must be translated into row and column specification.

▷ The address information is divided into two halves; the top half is used to select the row and the bottom half is used to select the column.

3.4 Computing Devices and Programming Languages

The main focus of this sectionis a discussion of the languages that can be used to program register machines: simple computational devices we can realize by combining algorithmic/logic circuits with memory. We start out with a simple assembler language which is largely given by the ALU employed and build up towards higher-level, more structured programming languages.

We build up language expressivity in levels, first defining a simple imperative programming language `SW` with arithmetic expressions, and block-structured control. One way to make this language run on our register machine would be via a compiler that transforms `SW` programs into assembler programs. As this would be very complex, we will go a different route: we first build an intermediate, stack-based programming language $\mathcal{L}(\mathtt{VM})$ and write a $\mathcal{L}(\mathtt{VM})$-interpreter in `ASM`, which acts as a stack-based virtual machine, into which we can compile `SW` programs.

The next level of complexity is to add (static) procedure calls to `SW`, for which we have to extend the $\mathcal{L}(\mathtt{VM})$ language and the interpreter with stack frame functionality. Armed with this, we can build a simple functional programming language μML and a full compiler into $\mathcal{L}(\mathtt{VM})$ for it.

We conclude this sectionby an investigation into the fundamental properties and limitations of computation, discussing Turing machines, universal machines, and the halting problem.

Acknowledgement: Some of the material in this sectionis inspired by and adapted from Gert Smolka excellent introduction to Computer Science based on SML [Smo11].

3.4.1 How to Build and Program a Computer (in Principle)

In this subsection, we will combine the arithmetic/logical units from Section 3.1 with the storage elements (RAM) from Subsection 3.3.1 to a fully programmable device: the register machine. The "von Neumann" architecture for computing we use in the register machine, is the prevalent architecture for general-purpose computing devices, such as personal computers nowadays. This architecture is widely attribute to the mathematician John von Neumann because of [vN45], but is already present in Konrad Zuse's 1936 patent application [Zus36].

REMA, a simple Register Machine

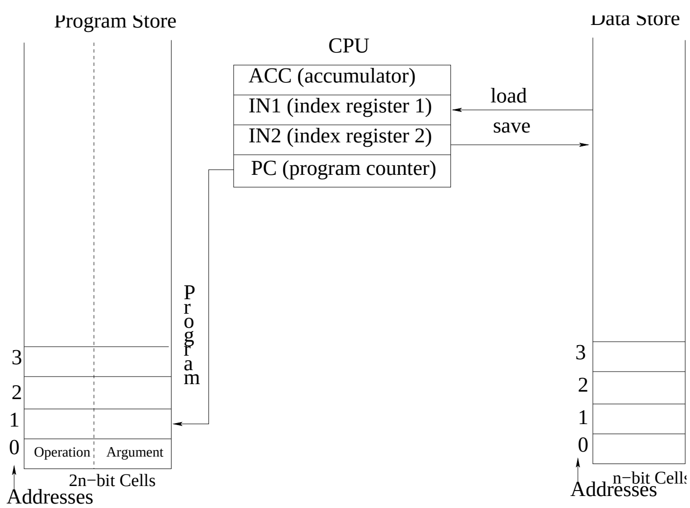
- ▷ Take an n-bit arithmetic logic unit (ALU)
- ▷ add registers: few (named) n-bit memory cells near the ALU
  - ▷ program counter (PC) (points to current command in program store)
  - ▷ accumulator (ACC) (the a input and output of the ALU)
- ▷ add RAM: lots of random access memory (elsewhere)
  - ▷ program store: $2n$-bit memory cells (addressed by $P\colon \mathbb{N} \to \mathbb{B}^{2n}$)
  - ▷ data store: n-bit memory cells (words addressed by $P\colon \mathbb{N} \to \mathbb{B}^{n}$)
- ▷ add a memory management unit(MMU) (move values between RAM and registers)
- ▷ program it in assembler language (lowest level of programming)

We have three kinds of memory areas in the `REMA` register machine: The registers (our architecture has two, which is the minimal number, real architectures have more for convenience) are just simple n-bit memory cells.

The programstore is a sequence of up to 2^n memory $2n$-bit memory cells, which can be accessed (written to and queried) randomly i.e. by referencing their position in the sequence; we do not have to access them by some fixed regime, e.g. one after the other, in sequence (hence the name random access memory: RAM). We address the Program store by a function $P\colon \mathbb{N} \to \mathbb{B}^{2n}$. The data store is also RAM, but a sequence or n-bit cells, which is addressed by the function $P\colon \mathbb{N} \to \mathbb{B}^{n}$.

The value of the program counter is interpreted as a binary number that addresses a $2n$-bit cell in the program store. The accumulator is the register that contains one of the inputs to the ALU before the operation (the other is given as the argument of the program instruction); the result of the ALU is stored in the accumulator after the instruction is carried out.

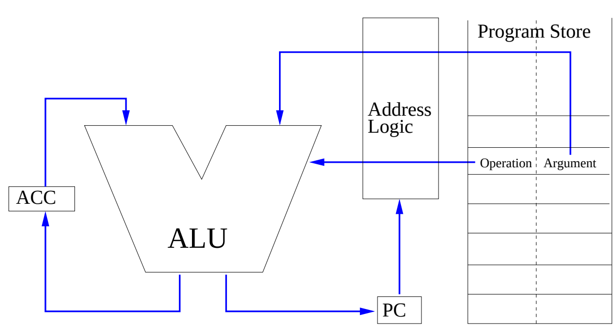
Memory Plan of a Register Machine



 ©: Michael Kohlhase 274 JACOBS UNIVERSITY

The ALU and the MMU are control circuits, they have a set of n-bit inputs, and n-bit outputs, and an n-bit control input. The prototypical ALU, we have already seen, applies arithmetic or logical operator to its regular inputs according to the value of the control input. The MMU is very similar, it moves n-bit values between the RAM and the registers according to the value at the control input. We say that the MMU moves the (n-bit) value from a register R to a memory cell C, iff after the move both have the same value: that of R. This is usually implemented as a query operation on R and a write operation to C. Both the ALU and the MMU could in principle encode 2^n operators (or commands), in practice, they have fewer, since they share the command space.

Circuit Overview over the CPU



 ©: Michael Kohlhase 275 JACOBS UNIVERSITY

In this architecture (called the register machine architecture), programs are sequences of $2n$-bit numbers. The first n-bit part encodes the instruction, the second one the argument of the instruction. The program counter addresses the current instruction (operation + argument).

Our notion of time is in this construction is very simplistic, in our analysis we assume a series of discrete clock ticks that synchronize all events in the circuit. We will only observe the circuits on each clock tick and assume that all computational devices introduced for the register machine complete computation before the next tick. Real circuits, also have a clock that synchronizes events (the clock frequency (currently around 3 GHz for desktop CPUs) is a common approximation measure of processor performance), but the assumption of elementary computations taking only one click is wrong in production systems.

We will now instantiate this general register machine with a concrete (hypothetical) realization, which is sufficient for general programming, in principle. In particular, we will need to identify a set of program operations. We will come up with 18 operations, so we need to set $n \geq 5$. It is possible to do programming with $n = 4$ designs, but we are interested in the general principles more than optimization.

The main idea of programming at the circuit level is to map the operator code (an n-bit binary number) of the current instruction to the control input of the ALU and the MMU, which will then perform the action encoded in the operator.

Since it is very tedious to look at the binary operator codes (even it we present them as hexadecimal numbers). Therefore it has become customary to use a mnemonic encoding of these in simple word tokens, which are simpler to read, the "assembler language".

Assembler Language

▷ Idea: Store program instructions as n-bit values in program store, map these to control inputs of ALU, MMU.

▷ **Definition 445** assembler language (ASM)as mnemonic encoding of n-bit binary codes.

instruction	effect	PC	comment
LOAD i	$ACC\colon = P(i)$	$PC\colon = PC + 1$	load data
STORE i	$P(i)\colon = ACC$	$PC\colon = PC + 1$	store data
ADD i	$ACC\colon = ACC + P(i)$	$PC\colon = PC + 1$	add to ACC
SUB i	$ACC\colon = ACC - P(i)$	$PC\colon = PC + 1$	subtract from ACC
LOADI i	$ACC\colon = i$	$PC\colon = PC + 1$	load number
ADDI i	$ACC\colon = ACC + i$	$PC\colon = PC + 1$	add number
SUBI i	$ACC\colon = ACC - i$	$PC\colon = PC + 1$	subtract number

Definition 446 The meaning of the program instructions are specified in their ability to change the state of the memory of the register machine. So to understand them, we have to trace the state of the memory over time (looking at a snapshot after each clock tick; this is what we do in the comment fields in the tables on the next slide). We speak of an imperative programming language, if this is the case.

Example 447 This is in contrast to the programming language SML that we have looked at before. There we are not interested in the state of memory. In fact state is something that we want to avoid in such functional programming languages for conceptual clarity; we relegated all things that need state into special constructs: effects.

To be able to trace the memory state over time, we also have to think about the initial state of the register machine (e.g. after we have turned on the power). We assume the state of the registers and the data store to be arbitrary (who knows what the machine has dreamt). More interestingly, we assume the state of the program store to be given externally. For the moment, we may assume (as was the case with the first computers) that the program store is just implemented as a large array of binary switches; one for each bit in the program store. Programming a computer at that time was done by flipping the switches ($2n$) for each instructions. Nowadays, parts of the initial program of a computer (those that run, when the power is turned on and bootstrap the operating system) is still given in special memory (called the firmware) that keeps its state even when power is shut off. This is conceptually very similar to a bank of switches.

Example Programs

▷ **Example 448** Exchange the values of cells 0 and 1 in the data store

P	instruction	comment
0	`LOAD` 0	$ACC\colon = P(0) = x$
1	`STORE` 2	$P(2)\colon = ACC = x$
2	`LOAD` 1	$ACC\colon = P(1) = y$
3	`STORE` 0	$P(0)\colon = ACC = y$
4	`LOAD` 2	$ACC\colon = P(2) = x$
5	`STORE` 1	$P(1)\colon = ACC = x$

▷ **Example 449** Let $P(1) = a$, $P(2) = b$, and $P(3) = c$, store $a + b + c$ in data cell 4

P	instruction	comment
0	`LOAD` 1	$ACC\colon = P(1) = a$
1	`ADD` 2	$ACC\colon = ACC + P(2) = a + b$
2	`ADD` 3	$ACC\colon = ACC + P(3) = a + b + c$
3	`STORE` 4	$P(4)\colon = ACC = a + b + c$

▷ use `LOADI` i, `ADDI` i, `SUBI` i to set/increment/decrement ACC (impossible otherwise)

So far, the problems we have been able to solve are quite simple. They had in common that we had to know the addresses of the memory cells we wanted to operate on at programming time, which is not very realistic. To alleviate this restriction, we will now introduce a new set of instructions, which allow to calculate with addresses.

Index Registers

▷ Problem: Given $P(0) = x$ and $P(1) = y$, how to we store y into cell x of the data store? (impossible, as we have only absolute addressing)

▷ **Definition 450 (Idea)** introduce more registers and register instructions ($IN1$, $IN2$ suffice)

instruction	effect	PC	comment
`LOADIN` j i	$ACC\colon = P(INj + i)$	$PC\colon = PC + 1$	relative load
`STOREIN` j i	$P(INj + i)\colon = ACC$	$PC\colon = PC + 1$	relative store
`MOVE` S T	$T\colon = S$	$PC\colon = PC + 1$	move register S (source) to register T (target)

▷ Problem Solution:

P	instruction	comment
0	`LOAD` 0	$ACC\colon = P(0) = x$
1	`MOVE` ACC $IN1$	$IN1\colon = ACC = x$
2	`LOAD` 1	$ACC\colon = P(1) = y$
3	`STOREIN` 1 0	$P(x) = P(IN1 + 0)\colon = ACC = y$

Note that the `LOADIN` are not binary instructions, but that this is just a short notation for unary instructions `LOADIN` 1 and `LOADIN` 2 (and similarly for `MOVE` S T).

Note furthermore, that the addition logic in `LOADIN` j is simply for convenience (most assembler languages have it, since working with address offsets is commonplace). We could have always imitated this by a simpler relative load command and an `ADD` instruction.

A very important ability we have to add to the language is a set of instructions that allow us to re-use program fragments multiple times. If we look at the instructions we have seen so far, then we see that they all increment the program counter. As a consequence, program execution is a linear walk through the program instructions: every instruction is executed exactly once. The set of problems we can solve with this is extremely limited. Therefore we add a new kind of instruction. Jump instructions directly manipulate the program counter by adding the argument to it (note that this partially invalidates the circuit overview slide above[15], but we will not worry about this). EdNote:15

Another very important ability is to be able to change the program execution under certain conditions. In our simple language, we will only make jump instructions conditional (this is sufficient, since we can always jump the respective instruction sequence that we wanted to make conditional). For convenience, we give ourselves a set of comparison relations (two would have sufficed, e.g. $=$ and $<$) that we can use to test.

Jump Instructions

- ▷ Problem: Until now, we can only write linear programs (A program with n steps executes n instructions)
- ▷ Idea: Need instructions that manipulate the PC directly
- ▷ **Definition 451** Let $\mathcal{R} \in \{<,=,>,\leq,\neq,\geq\}$ be a comparison relation

instruction	effect	PC	comment
`JUMP` i		$PC := PC + i$	jump forward i steps
`JUMP`$_{\mathcal{R}}$ i		$PC := \begin{cases} PC + i & \text{if } \mathcal{R}(ACC, 0) \\ PC + 1 & \text{else} \end{cases}$	conditional jump

- ▷ **Definition 452 (Two more)**

instruction	effect	PC	comment
`NOP` i `STOP` i		$PC := PC + 1$	no operation stop computation

 279 JACOBS UNIVERSITY

The final addition to the language are the `NOP` (no operation) and `STOP` operations. Both do not look at their argument (we have to supply one though, so we fit our instruction format). the `NOP` instruction is sometimes convenient, if we keep jump offsets rational, and the `STOP` instruction terminates the program run (e.g. to give the user a chance to look at the results.)

Example Program

- ▷ Now that we have completed the language, let us see what we can do.
- ▷ **Example 453** Let $P(0) = n$, $P(1) = a$, and $P(2) = b$, copy the values of cells $a, \ldots, a + n - 1$ to cells $b, \ldots, b + n - 1$, while $a, b \geq 3$ and $|a - b| \geq n$.

[15] EdNote: reference

P	instruction	comment	P	instruction	comment
0	`LOAD` 1	$ACC\colon = a$	10	`MOVE` ACC $IN1$	$IN1\colon = IN1+1$
1	`MOVE` ACC $IN1$	$IN1\colon = a$	11	`MOVE` $IN2$ ACC	
2	`LOAD` 2	$ACC\colon = b$	12	`ADDI` 1	
3	`MOVE` ACC $IN2$	$IN2\colon = b$	13	`MOVE` ACC $IN2$	$IN2\colon = IN2+1$
4	`LOAD` 0	$ACC\colon = n$	14	`LOAD` 0	
5	`JUMP`$_=$ 13	if $n = 0$ then stop	15	`SUBI` 1	
6	`LOADIN` 1 0	$ACC\colon = P(IN1)$	16	`STORE` 0	$P(0)\colon = P(0) - 1$
7	`STOREIN` 2 0	$P(IN2)\colon = ACC$	17	`JUMP` -12	goto step 5
8	`MOVE` $IN1$ ACC		18	`STOP` 0	Stop
9	`ADDI` 1				

▷ **Lemma 454** *We have $P(0) = n - (i - 1)$, $IN1 = a + i - 1$, and $IN2 = b + i - 1$ for all $(1 \leq i \leq n + 1)$.* *(the program does what we want)*

▷ proof by induction on n.

▷ **Definition 455** The induction hypotheses are called loop invariants.

3.4.2 A Stack-based Virtual Machine

We have seen that our register machine runs programs written in assembler, a simple machine language expressed in two-word instructions. Machine languages should be designed such that on the processors that can be built machine language programs can execute efficiently. On the other hand machine languages should be built, so that programs in a variety of high-level programming languages can be transformed automatically (i.e. compiled) into efficient machine programs. We have seen that our assembler language `ASM` is a serviceable, if frugal approximation of the first goal for very simple processors. We will (eventually) show that it also satisfies the second goal by exhibiting a compiler for a simple SML-like language.

In the last 20 years, the machine languages for state-of-the art processors have hardly changed. This stability was a precondition for the enormous increase of computing power we have witnessed during this time. At the same time, high-level programming languages have developed considerably, and with them, their needs for features in machine-languages. This leads to a significant mismatch, which has been bridged by the concept of a *virtual machine*.

Definition 456 A virtual machine is a simple machine-language program that interprets a slightly higher-level program — the "byte code" — and simulates it on the existing processor.

Byte code is still considered a machine language, just that it is realized via software on a real computer, instead of running directly on the machine. This allows to keep the compilers simple while only paying a small price in efficiency.

In our compiler, we will take this approach, we will first build a simple virtual machine (an `ASM` program) and then build a compiler that translates functional programs into byte code.

Virtual Machines

▷ Question: How to run high-level programming languages (like SML) on `REMA`?

▷ Answer: By providing a compiler, i.e. an `ASM` program that reads SML programs (as data) and transforms them into `ASM` programs.

▷ But: `ASM` is optimized for building simple, efficient processors, not as a translation target!

▷ Idea: Build an `ASM` program `VM` that interprets a better translation target language (interpret `REMA`+`VM` as a "virtual machine")

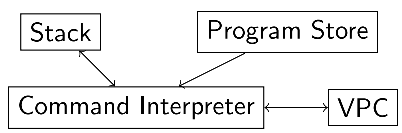
▷ **Definition 457** An `ASM` program `VM` is called a virtual machine for $\mathcal{L}$(`VM`), iff `VM` inputs a $\mathcal{L}$(`VM`) program (as data) and runs it on `REMA`.

▷ Plan: Instead of building a compiler for SML to `ASM`, build a virtual machine `VM` for `REMA` and a compiler from SML to $\mathcal{L}$(`VM`). (simpler and more transparent)

The main difference between the register machine `REMA` and the virtual machine `VM` construct is the way it organizes its memory. The `REMA` gives the assembler language full access to its internal registers and the data store, which is convenient for direct programming, but not suitable for a language that is mainly intended as a compilation target for higher-level languages which have regular (tree-like) structures. The virtual machine `VM` builds on the realization that tree-like structures are best supported by stack-like memory organization.

A Virtual Machine for Functional Programming

▷ We will build a stack-based virtual machine; this will have four components



- ▷ The stack is a memory segment operated as a "last-in-first-out" LIFO sequence
- ▷ The program store is a memory segment interpreted as a sequence of instructions
- ▷ The command interpreter is a `ASM` program that interprets commands from the program store and operates on the stack.
- ▷ The virtual program counter (VPC) is a register that acts as a the pointer to the current instruction in the program store.
- ▷ The virtual machine starts with the empty stack and VPC at the beginning of the program.

A Stack-based Programming Language

Now we are in a situation, where we can introduce a programming language for `VM`. The main difference to `ASM` is that the commands obtain their arguments by popping them from the stack (as opposed to the accumulator or the `ASM` instructions) and return them by pushing them to the stack (as opposed to just leaving them in the registers).

A Stack-Based VM language (Arithmetic Commands)

▷ **Definition 458** VM Arithmetic Commands act on the stack

instruction	effect	VPC
`con` i	pushes i onto stack	VPC$:=$VPC$+2$
`add`	pop x, pop y, push $x+y$	VPC$:=$VPC$+1$
`sub`	pop x, pop y, push $x-y$	VPC$:=$VPC$+1$
`mul`	pop x, pop y, push $x\cdot y$	VPC$:=$VPC$+1$
`leq`	pop x, pop y, if $x\leq y$ push 1, else push 0	VPC$:=$VPC$+1$

▷ **Example 459** The $\mathcal{L}$(`VM`) program "`con` 4 `con` 7 `add`" pushes $7+4=11$ to the stack.

▷ **Example 460** Note the order of the arguments: the program "`con 4 con 7 sub`" first pushes 4, and then 7, then pops x and then y (so $x = 7$ and $y = 4$) and finally pushes $x - y = 7 - 4 = 3$.

Stack-based operations work very well with the recursive structure of arithmetic expressions: we can compute the value of the expression $4 \cdot 3 - 7 \cdot 2$ with

▷

`con 2 con 7 mul`	$7 \cdot 2$
`con 3 con 4 mul`	$4 \cdot 3$
`sub`	$4 \cdot 3 - 7 \cdot 2$

Note: A feature that we will see time and again is that every (syntactically well-formed) expression leaves only the result value on the stack. In the present case, the computation never touches the part of the stack that was present before computing the expression. This is plausible, since the computation of the value of an expression is purely functional, it should not have an effect on the state of the virtual machine `VM` (other than leaving the result of course).

A Stack-Based VM language (Control)

▷ **Definition 461** Control operators

instruction	effect	VPC
`jp` i		VPC: $=$ VPC $+ i$
`cjp` i	pop x	if $x = 0$, then VPC: $=$ VPC $+ i$ else VPC: $=$ VPC $+ 2$
`halt`		—

▷ `cjp` is a "jump on false"-type expression.(if the condition is false, we jump else we continue)

▷ **Example 462** For conditional expressions we use the conditional jump expressions: We can express "if $1 \leq 2$ then $4 - 3$ else $7 \cdot 5$" by the program

`con 2 con 1 leq cjp 9`	if $1 \leq 2$
`con 3 con 4 sub jp 7`	then $4 - 3$
`con 5 con 7 mul`	else $7 \cdot 5$
`halt`	

In the example, we first push 2, and then 1 to the stack. Then `leq` pops (so $x = 1$), pops again (making $y = 2$) and computes $x \leq y$ (which comes out as true), so it pushes 1, then it continues (it would jump to the else case on false).

Note: Again, the only effect of the conditional statement is to leave the result on the stack. It does not touch the contents of the stack at and below the original stack pointer.

The next two commands break with the nice principled stack-like memory organization by giving "random access" to lower parts of the stack. We will need this to treat variables in high-level programming languages

A Stack-Based VM language (Imperative Variables)

▷ **Definition 463** Imperative access to variables: Let $\mathcal{S}(i)$ be the number at stack position i.

instruction	effect	VPC
peek i	push $\mathcal{S}(i)$	VPC: $=$ VPC $+ 2$
poke i	pop x $\mathcal{S}(i) := x$	VPC: $=$ VPC $+ 2$

▷ **Example 464** The program "con 5 con 7 peek 0 peek 1 add poke 1 mul halt" computes $5 \cdot (7 + 5) = 60$.

Of course the last example is somewhat contrived, this is certainly not the best way to compute $5 \cdot (7+5) = 60$, but it does the trick. In the intended application of $\mathcal{L}(\mathtt{VM})$ as a compilation target, we will only use `peek` and $VMpoke$ for read and write access for variables. In fact `poke` will not be needed if we are compiling purely functional programming languages.

To convince ourselves that $\mathcal{L}(\mathtt{VM})$ is indeed expressive enough to express higher-level programming constructs, we will now use it to model a simple while loop in a `C`-like language.

Extended Example: A `while` Loop

▷ **Example 465** Consider the following program that computes $(12)!$ and the corresponding $\mathcal{L}(\mathtt{VM})$ program:

`var n := 12; var a := 1;`	con 12 con 1
`while 2 <= n do (`	peek 0 con 2 leq cjp 18
`  a := a * n;`	peek 0 peek 1 mul poke 1
`  n := n - 1;`	con 1 peek 0 sub poke 0
`)`	jp -21
`return a;`	peek 1 halt

▷ Note that variable declarations only push the values to the stack, (memory allocation)

▷ they are referenced by peeking the respective stack position

▷ they are assigned by pokeing the stack position (must remember that)

We see that again, only the result of the computation is left on the stack. In fact, the code snippet consists of two variable declarations (which extend the stack) and one `while` statement, which does not, and the `return` statement, which extends the stack again. In this case, we see that even though the `while` statement does not extend the stack it does change the stack below by the variable assignments (implemented as `poke` in $\mathcal{L}(\mathtt{VM})$). We will use the example above as guiding intuition for a compiler from a simple imperative language to $\mathcal{L}(\mathtt{VM})$ byte code below. But first we build a virtual machine for $\mathcal{L}(\mathtt{VM})$.

Building a Virtual Machine

We will now build a virtual machine for $\mathcal{L}(\mathtt{VM})$ along the specification above.

A Virtual Machine for $\mathcal{L}(\mathtt{VM})$

▷ We need to build a concrete `ASM` program that acts as a virtual machine for $\mathcal{L}(\mathtt{VM})$.

▷ Choose a concrete register machine size: e.g. 32-bit words (like in a PC)

▷ Choose memory layout in the data store

  ▷ the `VM` stack: $P(8)$ to $P(2^{24} - 1)$, and (need the first 8 cells for `VM` data)

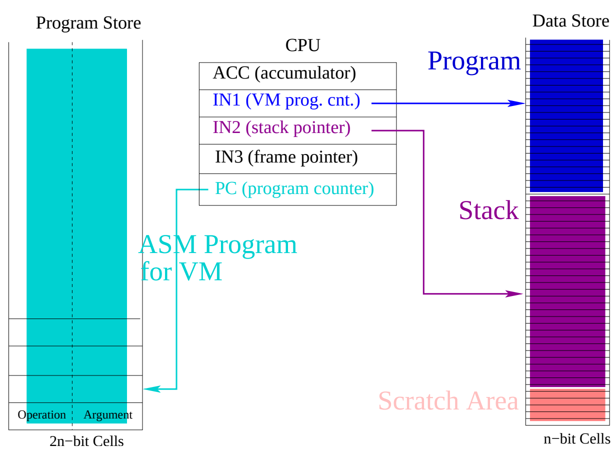
- the $\mathcal{L}(\texttt{VM})$ program store: $P(2^{24})$ to $P(2^{32}-1)$
- We represent the virtual program counter VPC by the index register $IN1$ and the stack pointer by the index register $IN2$ (with offset 8).
- We will use $P(0)$ as an argument store.

▷ choose a numerical representation for the $\mathcal{L}(\texttt{VM})$ instructions: (have lots of space)

`halt` $\mapsto 0$, `add` $\mapsto 1$, `sub` $\mapsto 2$, ...

Recall that the virtual machine `VM` is a `ASM` program, so it will reside in the `REMA` program store. This is the program executed by the register machine. So both the `VM` stack and the $\mathcal{L}(\texttt{VM})$ program have to be stored in the `REMA` data store (therefore we treat $\mathcal{L}(\texttt{VM})$ programs as sequences of words and have to do counting acrobatics for instructions of differing length). We somewhat arbitrarily fix a boundary in the data store of `REMA` at cell number $2^{24}-1$. We will also need a little piece of scratch-pad memory, which we locate at cells 0-7 for convenience (then we can simply address with absolute numbers as addresses).

Memory Layout for the Virtual Machine



To make our implementation of the virtual more convenient, we will extend `ASM` with a couple of convenience features. Note that these features do not extend the theoretical expressivity of `ASM` (i.e. they do not extend the range of programs that `ASM`), since all new commands can be replaced by regular language constructs.

Extending `REMA` and `ASM`

▷ Give ourselves another register $IN3$ (and `LOADIN` 3, `STOREIN` 3, `MOVE` $*$ $IN3$, `MOVE` $IN3$ $*$)

▷ We will use a syntactic variant of `ASM` for transparency

- `JUMP` and `JUMP`$_{\mathcal{R}}$ with labels of the form $\langle foo\rangle$ (compute relative jump distances automatically)
- `inc` R for `MOVE` R ACC, `ADDI` 1, `MOVE` ACC R (`dec` R similar)
- note that `inc` R and `dec` R overwrite the current ACC (take care of it)

▷ All additions can be eliminated by substitution.

With these extensions, it is quite simple to write the ASM code that implements the virtual machine VM.

The first part of VM is a simple jump table, a piece of code that does nothing else than distributing the program flow according to the (numerical) instruction head. We assume that this program segment is located at the beginning of the program store, so that the REMA program counter points to the first instruction. This initializes the VM program counter and its stack pointer to the first cells of their memory segments. We assume that the $\mathcal{L}$(VM) program is already loaded in its proper location, since we have not discussed input and output for REMA.

Starting VM: the Jump Table

label	instruction	effect	comment
	LOADI 2^{24}	$ACC := 2^{24}$	load VM start address
	MOVE ACC $IN1$	VPC $:= ACC$	set VPC
	LOADI 7	$ACC := 7$	load top of stack address
	MOVE ACC $IN2$	SP $:= ACC$	set SP
$\langle jt \rangle$	LOADIN 1 0	$ACC := P(IN1)$	load instruction
	JUMP$_=$ ⟨halt⟩		goto ⟨halt⟩
	SUBI 1		next instruction code
	JUMP$_=$ ⟨add⟩		goto ⟨add⟩
	SUBI 1		next instruction code
	JUMP$_=$ ⟨sub⟩		goto ⟨sub⟩
	⋮	⋮	⋮
⟨halt⟩	STOP 0		stop
	⋮	⋮	⋮

Now it only remains to present the ASM programs for the individual $\mathcal{L}$(VM) instructions. We will start with the arithmetical operations.

The code for con is absolutely straightforward: we increment the VM program counter to point to the argument, read it, and store it to the cell the (suitably incremented) VM stack pointer points to. Once procedure has been executed we increment the VM program counter again, so that it points to the next $\mathcal{L}$(VM) instruction, and jump back to the beginning of the jump table.

For the add instruction we have to use the scratch pad area, since we have to pop two values from the stack (and we can only keep one in the accumulator). We just cache the first value in cell 0 of the program store.

Implementing Arithmetic Operators

label	instruction	effect	comment
⟨con⟩	inc $IN1$	VPC $:=$ VPC $+ 1$	point to arg
	inc $IN2$	SP $:=$ SP $+ 1$	prepare push
	LOADIN 1 0	$ACC := P(\text{VPC})$	read arg
	STOREIN 2 0	$P(\text{SP}) := ACC$	store for push
	inc $IN1$	VPC $:=$ VPC $+ 1$	point to next
	JUMP $\langle jt \rangle$		jump back
⟨add⟩	LOADIN 2 0	$ACC := P(\text{SP})$	read arg 1
	STORE 0	$P(0) := ACC$	cache it
	dec $IN2$	SP $:=$ SP $- 1$	pop
	LOADIN 2 0	$ACC := P(\text{SP})$	read arg 2
	ADD 0	$ACC := ACC + P(0)$	add cached arg 1
	STOREIN 2 0	$P(\text{SP}) := ACC$	store it
	inc $IN1$	VPC $:=$ VPC $+ 1$	point to next
	JUMP $\langle jt \rangle$		jump back

▷ sub, similar to add.

▷ mul, and leq need some work.

We will not go into detail for the other arithmetic commands, for example, `mul` could be implemented as follows:

label	instruction	effect	comment
⟨mul⟩	dec $IN2$	$\mathrm{SP}: = \mathrm{SP} - 1$	
	LOADI 0		
	STORE 1	$P(1): = 0$	initialize result
	LOADIN 2 1	$ACC: = P(\mathrm{SP}+1)$	read arg 1
	STORE 0	$P(0): = ACC$	initialize counter to arg 1
⟨*loop*⟩	$\mathtt{JUMP}_=$ ⟨*end*⟩		if counter=0, we are finished
	LOADIN 2 0	$ACC: = P(\mathrm{SP})$	read arg 2
	ADD 1	$ACC: = ACC + P(1)$	current sum increased by arg 2
	STORE 1	$P(1): = ACC$	cache result
	LOAD 0		
	SUBI 1		
	STORE 0	$P(0): = P(0) - 1$	decrease counter by 1
	JUMP *loop*		repeat addition
⟨*end*⟩	LOAD 1		load result
	STOREIN 2 0		push it on stack
	inc $IN1$		
	JUMP ⟨*jt*⟩		back to jump table

Note that `mul` and `leq` are the only two instruction whose corresponding piece of code is not of the unit complexity.[16] EdNote:16

For the jump instructions, we do exactly what we would expect, we load the jump distance, add it to the register $IN1$, which we use to represent the `VM` program counter VPC. Incidentally, we can use the code for `jp` for the conditional jump `cjp`.

Control Instructions

label	instruction	effect	comment
⟨jp⟩	MOVE $IN1$ ACC	$ACC: = \mathrm{VPC}$	
	STORE 0	$P(0): = ACC$	cache VPC
	LOADIN 1 1	$ACC: = P(\mathrm{VPC}+1)$	load i
	ADD 0	$ACC: = ACC + P(0)$	compute new VPC value
	MOVE ACC $IN1$	$IN1: = ACC$	update VPC
	JUMP ⟨*jt*⟩		jump back
⟨cjp⟩	dec $IN2$	$\mathrm{SP}: = \mathrm{SP} - 1$	update for pop
	LOADIN 2 1	$ACC: = P(\mathrm{SP}+1)$	pop value to ACC
	$\mathtt{JUMP}_=$ ⟨jp⟩		perform jump if $ACC = 0$
	MOVE $IN1$ ACC		otherwise, go on
	ADDI 2		
	MOVE ACC $IN1$	$\mathrm{VPC}: = \mathrm{VPC} + 2$	point to next
	JUMP ⟨*jt*⟩		jump back

The imperative stack operations use the index register heavily. Note the use of the offset 8 in the `LOADIN` , this comes from the layout of `VM` that uses the bottom eight cells in the data store as a scratchpad.

[16] EdNote: MK: explain this better

Imperative Stack Operations: peek

label	instruction	effect	comment
⟨peek⟩	MOVE $IN1$ ACC	$ACC{:}=IN1$	
	STORE 0	$P(0){:}=ACC$	cache VPC
	LOADIN 1 1	$ACC{:}=P(\mathsf{VPC}+1)$	load i
	MOVE ACC $IN1$	$IN1{:}=ACC$	
	inc $IN2$		prepare push
	LOADIN 1 8	$ACC{:}=P(IN1+8)$	load $\mathcal{S}(i)$
	STOREIN 2 0		push $\mathcal{S}(i)$
	LOAD 0	$ACC{:}=P(0)$	load old VPC
	ADDI 2		compute new value
	MOVE ACC $IN1$		update VPC
	JUMP ⟨jt⟩		jump back

 293

Imperative Stack Operations: poke

label	instruction	effect	comment
⟨poke⟩	MOVE $IN1$ ACC	$ACC{:}=IN1$	
	STORE 0	$P(0){:}=ACC$	cache VPC
	LOADIN 1 1	$ACC{:}=P(\mathsf{VPC}+1)$	load i
	MOVE ACC $IN1$	$IN1{:}=ACC$	
	LOADIN 2 0	$ACC{:}=\mathcal{S}(i)$	pop to ACC
	STOREIN 1 8	$P(IN1+8){:}=ACC$	store in $\mathcal{S}(i)$
	dec $IN2$	$IN2{:}=IN2-1$	
	LOAD 0	$ACC{:}=P(0)$	get old VPC
	ADD 2	$ACC{:}=ACC+2$	add 2
	MOVE ACC $IN1$		update VPC
	JUMP ⟨jt⟩		jump back

 294

3.4.3 A Simple Imperative Language

We will now build a compiler for a simple imperative language to warm up to the task of building one for a functional one. We will write this compiler in SML, since we are most familiar with this. The first step is to define the language we want to talk about.

A very simple Imperative Programming Language

- Plan: Only consider the bare-bones core of a language. (we are only interested in principles)
  - We will call this language `SW` (Simple While Language)
  - no types: all values have type `int`, use `0` for `false` all other numbers for `true`.
- **Definition 466** The simple while language `SW` is a simple programming languages with
  - named variables (declare with `var` ⟪name⟫`:=`⟪exp⟫, assign with ⟪name⟫`:=`⟪exp⟫)
  - arithmetic/logic expressions with variables referenced by name
  - block-structured control structures (called statements), e.g.
    `while` ⟪exp⟫ `do` ⟪statement⟫ `end` and
    `if` ⟪exp⟫ `then` ⟪statement⟫ `else` ⟪statement⟫ `end`.
  - output via `return` ⟪exp⟫

 295

To make the concepts involved concrete, we look at a concrete example.

Example: An SW Program for 12 Factorial

▷ **Example 467 (Computing Twelve Factorial)**

```
var n:= 12; var a:= 1;    # declarations
while 2<=n do             # while block
  a:= a*n;                # assignment
  n:= n-1                 # another
end                       # end while block
return a                  # output
```

 296

Note that SW is a great improvement over ASM for a variety of reasons

- it introduces the concept of named variables that can be referenced and assigned to, without having to remember memory locations. Named variables are an important cognitive tool that allows programmers to associate concepts with (changing) values.
- It introduces the notion of (arithmetical) expressions made up of operators, constants, and variables. These can be written down declaratively (in fact they are very similar to the mathematical formula language that has revolutionized manual computation in everyday life).
- finally, SW introduces structured programming features (notably while loops) and avoids "spagetti code" induced by jump instructions (also called goto). See Edsgar Dijkstra's famous letter "Goto Considered Harmful". [Dij68] for a discussion.

The following slide presents the SML data types for SW programs.

Abstract Syntax of SW

▷ **Definition 468** `type id = string (* identifier *)`

```
datatype exp = (* expression *)
  Con of int (* constant *)
| Var of id (* variable *)
| Add of exp* exp (* addition *)
| Sub of exp * exp (* subtraction *)
| Mul of exp * exp (* multiplication *)
| Leq of exp * exp (* less or equal test *)

datatype sta = (* statement *)
  Assign of id * exp (* assignment *)
| If of exp * sta * sta (* conditional *)
| While of exp * sta (* while loop *)
| Seq of sta list (* sequentialization *)

type declaration = id * exp

type program = declaration list * sta * exp
```

 297

A SW program (see the next slide for an example) first declares a set of variables (type `declaration`), executes a statement (type `sta`), and finally returns an expression (type `exp`). Expressions of SW can read the values of variables, but cannot change them. The statements of SW can read and change the values of variables, but do not return values (as usual in imperative languages). Note that SW follows common practice in imperative languages and models the conditional as a statement.

Concrete vs. Abstract Syntax of a SW Program

▷ **Example 469 (Abstract SW Syntax)** We apply the abstract syntax to the SW program from Example 467:

var n:= 12; **var** a:= 1;
while 2<=n **do**
  a:= a*n;
  n:= n−1
end
return a
	([("n", Con 12),("a", Con 1)],
	While(Leq(Con 2, Var"n"),
	    Seq [Assign("a", Mul(Var"a", Var"n")),
	      Assign("n", Sub(Var"n", Con 1))]
	    ),
	Var"a")

As expected, the program is represented as a triple: the first component is a list of declarations, the second is a statement, and the third is an expression (in this case, the value of a single variable). We will use this example as the guiding intuition for building a compiler.

We will also need an SML type for $\mathcal{L}(\mathtt{VM})$ programs. Fortunately, this is very simple.

An SML Data Type for $\mathcal{L}(\mathtt{VM})$ Programs

```
type index = int
type noi = int (* number of instructions *)

datatype instruction =
    con of int
  | add | sub | mul (* addition, subtraction, multiplication *)
  | leq (* less or equal test *)
  | jp of noi (* unconditional jump *)
  | cjp of noi (* conditional jump *)
  | peek of index (* push value from stack *)
  | poke of index (* update value in stack *)
  | halt (* halt machine *)

type code = instruction list

fun wlen (xs:code) = foldl (fn (x,y) => wln(x)+y) 0 xs
fun wln(con _)=2 | wln(add)=1 | wln(sub)=1 | wln(mul)=1 | wln(leq)=1
  | wln(jp _)=2 | wln(cjp _)=2
  | wln(peek _)=2 | wln(poke _)=2 | wln(halt)=1
```


Before we can come to the implementation of the compiler, we will need an infrastructure for environments.

Needed Infrastructure: Environments

▷ Need a structure to keep track of the values of declared identifiers. (take shadowing into account)

▷ **Definition 470** An environment is a finite partial function from keys (identifiers) to values.

▷ We will need the following operations on environments:

- ▷ creation of an empty environment (⇝ the empty function)
- ▷ insertion of a key/value pair $\langle k, v\rangle$ into an environment φ: ($\leadsto \varphi, [v/k]$)
- ▷ lookup of the value v for a key k in φ ($\leadsto \varphi(k)$)

▷ Realization in SML by a structure with the following signature

```
type 'a env (* a is the value type *)
exception Unbound of id (* Unbound *)
val empty : 'a env
val insert : id * 'a * 'a env -> 'a env (* id is the key type *)
val lookup : id * 'a env -> 'a
```

 300 JACOBS UNIVERSITY

The next slide has the main SML function for compiling SW programs. Its argument is a SW program (type program) and its result is an expression of type code, i.e. a list of $\mathcal{L}$(VM) instructions. From there, we only need to apply a simple conversion (which we omit) to numbers to obtain $\mathcal{L}$(VM) byte code.

Compiling SW programs

▷ SML function from SW programs (type program) to $\mathcal{L}$(VM) programs (type code).

▷ uses three auxiliary functions for compiling declarations (compileD), statements (compileS), and expressions (compileE).

▷ these use an environment to relate variable names with their stack index.

▷ the initial environment is created by the declarations. (therefore compileD has an environment as return value)

```
type env = index env
fun compile ((ds,s,e) : program) : code =
  let
    val (cds, env) = compileD(ds, empty, ~1)
  in
    cds @ compileS(s,env) @ compileE(e,env) @ [halt]
  end
```

 301 JACOBS UNIVERSITY

The next slide has the function for compiling SW expressions. It is realized as a case statement over the structure of the expression.

Compiling SW Expressions

▷ constants are pushed to the stack.

▷ variables are looked up in the stack by the index determined by the environment (and pushed to the stack).

▷ arguments to arithmetic operations are pushed to the stack in reverse order.

```
fun compileE (e:exp, env:env) : code =
  case e of
    Con i => [con i]
  | Var i => [peek (lookup(i,env))]
  | Add(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [add]
  | Sub(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [sub]
  | Mul(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [mul]
  | Leq(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [leq]
```

 302 JACOBS UNIVERSITY

Compiling SW statements is only slightly more complicated: the constituent statements and expressions are compiled first, and then the resulting code fragments are combined by $\mathcal{L}$(VM) control instructions (as the fragments already exist, the relative jump distances can just be looked up). For a sequence of statements, we just map compileS over it using the respective environment.

Compiling SW Statements

```
fun compileS (s:sta, env:env) : code =
    case s of
      Assign(i,e) => compileE(e, env) @ [poke (lookup(i,env))]
    | If(e,s1,s2) =>
       let
         val ce = compileE(e, env)
         val cs1 = compileS(s1, env)
         val cs2 = compileS(s2, env)
       in
         ce @ [cjp (wlen cs1 + 4)] @ cs1 @ [jp (wlen cs2 + 2)] @ cs2
       end
    | While(e, s) =>
        let
         val ce = compileE(e, env)
         val cs = compileS(s, env)
        in
         ce @ [cjp (wlen cs + 4)] @ cs @ [jp (~(wlen cs + wlen ce + 2))]
        end
    | Seq ss => foldr (fn (s,c) => compileS(s,env) @ c) nil ss
```

 303 JACOBS UNIVERSITY

As we anticipated above, the compileD function is more complex than the other two. It gives $\mathcal{L}$(VM) program fragment and an environment as a value and takes a stack index as an additional argument. For every declaration, it extends the environment by the key/value pair k/v, where k is the variable name and v is the next stack index (it is incremented for every declaration). Then the expression of the declaration is compiled and prepended to the value of the recursive call.

Compiling SW Declarations

```
fun compileD (ds: declaration list, env:env, sa:index): code*env =
    case ds of
      nil => (nil,env)
    | (i,e)::dr => let
                     val env' = insert(i, sa+1, env)
                     val (cdr,env'') = compileD(dr, env', sa+1)
                   in
```

```
                (compileE(e,env) @ cdr, env'')
              end
```

 304

This completes the compiler for SW (except for the byte code generator which is trivial and an implementation of environments, which is available elsewhere). So, together with the virtual machine for $\mathcal{L}$(VM) we discussed above, we can run SW programs on the register machine REMA.

If we now use the REMA simulator from exercise[17], then we can run SW programs on our computers outright. EdNote:17

One thing that distinguishes SW from real programming languages is that it does not support procedure declarations. This does not make the language less expressive in principle, but makes structured programming much harder. The reason we did not introduce this is that our virtual machine does not have a good infrastructure that supports this. Therefore we will extend $\mathcal{L}$(VM) with new operations next.

Note that the compiler we have seen above produces $\mathcal{L}$(VM) programs that have what is often called "memory leaks". Variables that we declare in our SW program are not cleaned up before the program halts. In the current implementation we will not fix this (We would need an instruction for our VM that will "pop" a variable without storing it anywhere or that will simply decrease virtual stack pointer by a given value.), but we will get a better understanding for this when we talk about the static procedures next.

Compiling the Extended Example: A `while` Loop

▷ **Example 471** Consider the following program that computes (12)! and the corresponding $\mathcal{L}$(VM) program:

`var n := 12; var a := 1;`	con 12 con 1
`while 2 <= n do (`	peek 0 con 2 leq cjp 18
`  a := a * n;`	peek 0 peek 1 mul poke 1
`  n := n - 1;`	con 1 peek 0 sub poke 0
`)`	jp −21
`return a;`	peek 1 halt

▷ Note that variable declarations only push the values to the stack, (memory allocation)

▷ they are referenced by peeking the respective stack position

▷ they are assigned by pokeing the stack position (must remember that)

 305

The next step in our endeavor to understand programming languages is to extend the language SW with another structuring concept: procedures. Just like named variables allow to give (numerical) values a name and reference them under this name, procedures allow to encapsulate parts of programs, name them and reference them in multiple places. But rather than just adding procedures to SW, we will go one step further and directly design a functional language.

3.4.4 Basic Functional Programs

We will now study a minimal core of the functional programming language SML, which we will call μML.

[17] EdNote: include the exercises into the course materials and reference the right one here

μML, a very simple Functional Programming Language

- Plan: Only consider the bare-bones core of a language (we only interested in principles)
  - We will call this language μML (micro ML)
  - no types: all values have type `int`, use 0 for `false` all other numbers for `true`.
- **Definition 472** microML μML is a simple functional programming languages with
  - functional variables (declare and bind with `val` ⟪`name`⟫ `=` ⟪`exp`⟫)
  - named functions (declare with `fun` ⟪`name`⟫ `(`⟪`args`⟫`)` `=` ⟪`exp`⟫)
  - arithmetic/logic/control expressions with variables/functions referenced by name (no statements)

 306

To make the concepts involved concrete, we look at a concrete example: the procedure on the next slide computes 10^2..

Example: A μML Program for 10 Squared

- **Example 473 (Computing Twelve Factorial)**

```
let                          (* begin declarations *)
  fun exp(x,n) =             (* function declaration *)
    if n<=0                  (* if expression *)
    then 1                   (* then part *)
    else x*exp(x,n-1)        (* else part *)
 val y 10               (* value declaration *)
in                           (* end declarations *)
  exp(2,y)                  (* return value *)
end                          (* end program *)
```

 307

We will now extend the virtual machine by four instructions that allow to represent procedures with arbitrary numbers of arguments.

A Virtual Machine with Procedures

Adding Instructions for Procedures to $\mathcal{L}$(VM)

- **Definition 474** We obtain the language $\mathcal{L}$(VMP) by adding the following four commands to $\mathcal{L}$(VM):
  - `proc` a l contains information about the number a of arguments and the length l of the procedure in the number of words needed to store it. The command `proc` a l simply jumps $l+3$ words ahead.
  - `arg` i pushes the i^{th} argument from the current frame to the stack.
  - `call` p pushes the current program address (opens a new frame), and jumps to the program address p.
  - `return` takes the current frame from the stack, jumps to previous program address.

We will explain the meaning of these extensions by translating the μML function from Example 473 to $\mathcal{L}(\mathtt{VMP})$.

A μML Program and its $\mathcal{L}(\mathtt{VMP})$ Translation

▷ **Example 475 (A μML Program and its $\mathcal{L}(\mathtt{VMP})$ Translation)**

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

```
fun exp(x,n) =
   if n<=0
   then 1
   else x*exp(x,n-1)
 in
   exp(10,2)
 end
```


To see how these four commands together can simulate procedures, we simulate the program from the last slide, keeping track of the stack.

Static Procedures (Simulation)

Example 476 ▷

```
proc 2 26,
[con 0, arg 2, leq, cjp 5,
con 1, return,
con 1, arg 2, sub, arg 1,
call 0, arg 1, mul,
return,
con 2, con 10, call 0,
halt]
```

empty stack

▷ proc jumps over the body of the procedure declaration (with the help of its second argument.)

▷

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, jp 13,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10
2

▷ We push the arguments onto the stack

▷

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

32	0
10	-1
2	-2

▷ `call` pushes the return address (of the `call` statement in the $\mathcal{L}(\mathtt{VM})$ program)

▷ then it jumps to the first body instruction.

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

2	
0	
32	0
10	-1
2	-2

▷ arg i pushes the i^{th} argument onto the stack

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

0	
32	0
10	-1
2	-2

▷ Comparison turns out false, so we push 0.

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

32	0
10	-1
2	-2

▷ cjp pops the truth value and jumps (on false).

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

2	
1	
32	0
10	-1
2	-2

▷ we first push 1

▷ then we push the second argument (from the call frame position -2)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
32	0
10	-1
2	-2

▷ we subtract

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
1	
32	0
10	-1
2	-2

▷ then we push the second argument (from the call frame position -1)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
1	-2
32	
10	
2	

▷ call jumps to the first body instruction,

▷ and pushes the return address (22 this time) onto the stack.

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
0	
22	0
10	-1
1	-2
32	
10	
2	

▷ we augment the stack

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
1	-2
32	
10	
2	

▷ we compare the top two, and jump ahead (on false)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
1	
22	0
10	-1
1	-2
32	
10	
2	

▷ we augment the stack again

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
0	
22	0
10	-1
1	-2
32	
10	
2	

▷ subtract and push the first argument

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷ call pushes the return address and moves the current frame up

0	
0	
22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷
```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

▷ we augment the stack again,

22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷
```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

▷ leq compares the top two numbers, cjp pops the result and does not jump.

1	
22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷
```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

▷ we push the result value 1

1	
22	0
10	-1
1	-2
32	
10	
2	

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
▷ call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address memorized right below the top of the stack,
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack.

10	
1	
22	0
10	-1
1	-2
32	
10	
2	

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
▷ call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

- ▷ arg pushes the first argument from the (new) current frame

10	
22	0
10	-1
1	-2
32	
10	
2	

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
▷ call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

- ▷ `mul` multiplies, pops the arguments and pushes the result.

10	
32	0
10	-1
2	-2

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
▷ call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address,
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack.

```
  [proc 2 26,
   con 0, arg 2, leq, cjp 5,
   con 1, return,
   con 1, arg 2, sub, arg 1,
▷  call 0, arg 1, mul,
   return,
   con 2, con 10, call 0,
   halt]
```

100	
32	0
10	-1
2	-2

- ▷ we push argument 1 (in this case 10), multiply the top two numbers, and push the result to the stack

```
  [proc 2 26,
   con 0, arg 2, leq, cjp 5,
   con 1, return,
   con 1, arg 2, sub, arg 1,
▷  call 0, arg 1, mul,
   return,
   con 2, con 10, call 0,
   halt]
```

100

- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address (32 this time),
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack (which is empty here).

```
  [proc 2 26,
   con 0, arg 2, leq, cjp 5,
   con 1, return,
   con 1, arg 2, sub, arg 1,
▷  call 0, arg 1, mul,
   return,
   con 2, con 10, call 0,
   halt]
```

100

- ▷ we are finally done; the result is on the top of the stack. Note that the stack below has not changed.

 310 JACOBS UNIVERSITY

What have we seen?

▷ The four new `VMP` commands allow us to model recursive functions.

`proc` a l contains information about the number a of arguments and the length l of the procedure

`arg` i pushes the i^{th} argument from the current frame to the stack. (Note that arguments are stored in reverse order on the stack)

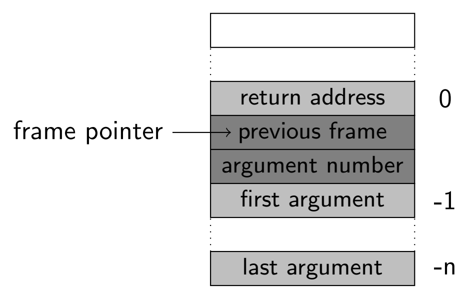
`call` p pushes the current program address (opens a new frame), and jumps to the program address p

`return` takes the current frame from the stack, jumps to previous program address. (which is cached in the frame)

▷ `call` and `return` jointly have the effect of replacing the arguments by the result of the procedure.

We will now extend our implementation of the virtual machine by the new instructions. The central idea is that we have to realize call frames on the stack, so that they can be used to store the data for managing the recursion.

Realizing Call Frames on the Stack



▷ Problem: How do we know what the current frame is? (after all, `return` has to pop it)

▷ Idea: Maintain another register: the frame pointer (FP), and cache information about the previous frame and the number of arguments in the frame.

▷ Add two internal cells to the frame, that are hidden to the outside. The upper one is called the anchor cell.

▷ In the anchor cell we store the stack address of the anchor cell of the previous frame.

▷ The frame pointer points to the anchor cell of the uppermost frame.

With this memory architecture realizing the four new commands is relatively straightforward.

Realizing `proc`

▷ `proc` a l jumps over the procedure with the help of the length l of the procedure.

label	instruction	effect	comment
⟨proc⟩	`MOVE` $IN1$ ACC	$ACC :=$ VPC	
	`STORE` 0	$P(0) := ACC$	cache VPC
	`LOADIN` 1 2	$ACC := P(\text{VPC} + 2)$	load length
	`ADD` 0	$ACC := ACC + P(0)$	compute new VPC value
	`MOVE` ACC $IN1$	$IN1 := ACC$	update VPC
	`JUMP` ⟨jt⟩		jump back

Realizing arg

▷ arg i pushes the i^{th} argument from the current frame to the stack.

▷ use the register $IN3$ for the frame pointer. (extend for first frame)

label	instruction	effect	comment
⟨arg⟩	LOADIN 1 1	$ACC := P(\mathsf{VPC}+1)$	load i
	STORE 0	$P(0) := ACC$	cache i
	MOVE $IN3$ ACC		
	STORE 1	$P(1) := \mathsf{FP}$	cache FP
	SUBI 1		
	SUB 0	$ACC := \mathsf{FP}-1-i$	load argument position
	MOVE ACC $IN3$	$\mathsf{FP} := ACC$	move it to FP
	inc $IN2$	$\mathsf{SP} := \mathsf{SP}+1$	prepare push
	LOADIN 3 0	$ACC := P(\mathsf{FP})$	load arg i
	STOREIN 2 0	$P(\mathsf{SP}) := ACC$	push arg i
	LOAD 1	$ACC := P(1)$	load FP
	MOVE ACC $IN3$	$\mathsf{FP} := ACC$	recover FP
	MOVE $IN1$ ACC		
	ADDI 2		
	MOVE ACC $IN1$	$\mathsf{VPC} := \mathsf{VPC}+2$	next instruction
	JUMP ⟨jt⟩		jump back

Realizing call

▷ call p pushes the current program address, and jumps to the program address p (pushes the internal cells first!)

label	instruction	effect	comment
⟨call⟩	MOVE $IN1$ ACC		
	STORE 0	$P(0) := IN1$	cache current VPC
	inc $IN2$	$\mathsf{SP} := \mathsf{SP}+1$	prepare push for later
	LOADIN 1 1	$ACC := P(\mathsf{VPC}+1)$	load argument
	ADDI $2^{24}+3$	$ACC := ACC + 2^{24}+3$	add displacement and skip proc a l
	MOVE ACC $IN1$	$\mathsf{VPC} := ACC$	point to the first instruction
	LOADIN 1 -2	$ACC := P(\mathsf{VPC}-2)$	stealing a from proc a l
	STOREIN 2 0	$P(\mathsf{SP}) := ACC$	push the number of arguments
	inc $IN2$	$\mathsf{SP} := \mathsf{SP}+1$	prepare push
	MOVE $IN3$ ACC	$ACC := IN3$	load FP
	STOREIN 2 0	$P(\mathsf{SP}) := ACC$	create anchor cell
	MOVE $IN2$ $IN3$	$\mathsf{FP} := \mathsf{SP}$	update FP
	inc $IN2$	$\mathsf{SP} := \mathsf{SP}+1$	prepare push
	LOAD 0	$ACC := P(0)$	load VPC
	ADDI 2	$ACC := ACC + 2$	point to next instruction
	STOREIN 2 0	$P(\mathsf{SP}) := ACC$	push the return address
	JUMP ⟨jt⟩		jump back

Note that with these instructions we have maintained the linear quality. Thus the virtual machine is still linear in the speed of the underlying register machine REMA.

Realizing return

▷ return takes the current frame from the stack, jumps to previous program address. (which is cached in the frame)

label	instruction	effect	comment
⟨return⟩	LOADIN 2 0	$ACC\colon = P(\mathsf{SP})$	load top value
	STORE 0	$P(0)\colon = ACC$	cache it
	LOADIN 2 -1	$ACC\colon = P(\mathsf{SP}-1)$	load return address
	MOVE ACC $IN1$	$IN1\colon = ACC$	set VPC to it
	LOADIN 3 -1	$ACC\colon = P(\mathsf{FP}-1)$	load the number n of arguments
	STORE 1	$P(1)\colon = P(\mathsf{FP}-1)$	cache it
	MOVE $IN3$ ACC	$ACC\colon = \mathsf{FP}$	$ACC = \mathsf{FP}$
	SUBI 1	$ACC\colon = ACC-1$	$ACC = \mathsf{FP}-1$
	SUB 1	$ACC\colon = ACC-P(1)$	$ACC = \mathsf{FP}-1-n$
	MOVE ACC $IN2$	$IN2\colon = ACC$	$\mathsf{SP} = ACC$
	LOADIN 3 0	$ACC\colon = P(\mathsf{FP})$	load anchor value
	MOVE ACC $IN3$	$IN3\colon = ACC$	point to previous frame
	LOAD 0	$ACC\colon = P(0)$	load cached return value
	STOREIN 2 0	$P(IN2)\colon = ACC$	pop return value
	JUMP ⟨jt⟩		jump back

 316 JACOBS UNIVERSITY

Note that all the realizations of the $\mathcal{L}$(VM) instructions are linear code segments in the assembler code, so they can be executed in linear time. Thus the virtual machine language is only a constant factor slower than the clock speed of REMA. This is characteristic for virtual machines.

The next step is to build a compiler for μML into programs in the extended $\mathcal{L}$(VM). Just as above, we will write this compiler in SML.

Compiling Basic Functional Programs

For the μML compiler we will proceed as above: we first introduce SML data types for the abstract syntax of $\mathcal{L}$(VMP) and μML and then we define a SML function that converts abstract μML programs to abstract VMP programs.

Abstract Syntax of μML

```
type id = string                        (* identifier            *)

datatype exp =                          (* expression            *)
    Con  of int                            (* constant              *)
  | Id   of id                             (* argument              *)
  | Add  of exp * exp                      (* addition              *)
  | Sub  of exp * exp                      (* subtraction           *)
  | Mul  of exp * exp                      (* multiplication        *)
  | Leq  of exp * exp                      (* less or equal test *)
  | App  of id  * exp list                 (* application           *)
  | If   of exp * exp * exp                (* conditional           *)

type declaration = id * id list * exp

type program = declaration list * exp
```

 317 JACOBS UNIVERSITY

Concrete vs. Abstract Syntax of μML

▷ A μML program first declares procedures, then evaluates expression for the return value.

```
let                              ([
  fun exp(x,n) =                    ("exp", ["x", "n"],
    if n<=0                          If(Leq(Id"n", Con 0),
    then 1                              Con 1,
    else x*exp(x,n-1)                   Mul(Id"x", App("exp", [Id"x", Sub(Id"n", Con 1)]))))
in                                 ],
  exp(2,10)                        App("exp", [Con 2, Con 10])
end                              )
```


Abstract Syntax for $\mathcal{L}$(VMP)

- Extensions to the Virtual Machine

```
type index = int
type noi = int (* number of instructions *)
type noa = int (* number of arguments *)
type ca = int (* code address *)

datatype instruction =
    ...
  | proc of noa*noi (* begin of procedure code *)
  | arg of index (* push value from frame *)
  | call of ca (* call procedure *)
  | return (* return from procedure call *)
```


For our μML compiler, we first need to define some auxiliary functions.

Compiling μML: Auxiliaries

```
exception Error of string
datatype idType = Arg of index | Proc of ca
type env = idType env

fun lookupA (i,env) =
  case lookup(i,env) of
    Arg i => i
  | _ => raise Error("Argument␣expected:␣" \^ i)

fun lookupP (i,env) =
  case lookup(i,env) of
    Proc ca => ca
  | _ => raise Error("Procedure␣expected:␣" \^ i)
```


Next we define a function that compiles abstract μML expressions into lists of abstract $\mathcal{L}$(VMP) instructions. As expressions also appear in argument sequences, it is convenient to define a function that compiles μML expression lists via left folding. Note that the two expression compilers are very naturally mutually recursive. Another trick we already do is that we give the expression compiler an argument `tail`, which can be used to append a list of $\mathcal{L}$(VMP) commands to the result; this will be useful in the declaration compiler later to take care of the `return` statment needed to return from recursive functions.

Compiling μML Expressions (Continued)

```
fun compileE (e:exp, env:env, tail:code) : code =
  case e of
    Con i => [con i] @ tail
  | Id i => [arg((lookupA(i,env)))] @ tail
  | Add(e1,e2) => compileEs([e1,e2], env) @ [add] @ tail
  | Sub(e1,e2) => compileEs([e1,e2], env) @ [sub] @ tail
  | Mul(e1,e2) => compileEs([e1,e2], env) @ [mul] @ tail
  | Leq(e1,e2) => compileEs([e1,e2], env) @ [leq] @ tail
  | If(e1,e2,e3) => let
                      val c1 = compileE(e1,env,nil)
                      val c2 = compileE(e2,env,tail)
                      val c3 = compileE(e3,env,tail)
                    in if null tail
                      then c1 @ [cjp (4+wlen c2)] @ c2
                              @ [jp (2+wlen c3)] @ c3
                      else c1 @ [cjp (2+wlen c2)] @ c2 @ c3
                    end
  | App(i, es) => compileEs(es,env) @ [call (lookupP(i,env))] @ tail
  and (* mutual recursion with compileE *)
fun compileEs (es : exp list, env:env) : code =
  foldl (fn (e,c) => compileE(e, env, nil) @ c) nil es
```


Now we turn to the declarations compiler. This is considerably more complex than the one for SW we had before due to the presence of formal arguments in the function declarations. We first define a function that inserts function arguments into an environment. Then we use that in the expression compiler to insert the function name and the list of formal arugments into the environment for later reference. In this environment env'' we compile the body of the function (which may contain the formal arugments). Observe the use of the tail argument for compileE to pass the return command. Note that we compile the rest of the declarations in the environment env' that contains the function name, but not the function arguments.

Compiling μML Expressions (Continued)

```
fun insertArgs' (i, (env, ai)) = (insert(i,Arg ai,env), ai+1)

fun insertArgs (is, env) = (foldl insertArgs' (env,1) is)

fun compileD (ds: declaration list, env:env, ca:ca) : code*env =
  case ds of
    nil => (nil,env)
  | (i,is,e)::dr =>
       let
         val env' = insert(i, Proc(ca+1), env)
         val env'' = insertArgs(is, env')
         val ce = compileE(e, env'', [return])
         val cd = [proc (length is, 3+wlen ce)] @ ce
                                    (* 3+wlen ce = wlen cd *)
         val (cdr,env'') = compileD(dr, env', ca + wlen cd)
       in
         (cd @ cdr, env'')
       end
```

As μML are programs are pairs consisting of declaration lists and an expression, we have a main function `compile` that first analyzes the declarations (getting a command sequence and an environment back from the declaration compiler) and then appends the command sequence, the compiled expression and the halt command. Note that the expression is compiled with respect to the environment computed in the compilation of the declarations.

Compiling μML

```
fun compile ((ds,e) : program) : code =
  let
    val (cds,env) = compileD(ds, empty, ~1)
  in
    cds @ compileE(e,env,nil) @ [halt]
  end
  handle
  Unbound i => raise Error("Unbound␣identifier:␣" \^ i)
```


Now that we have seen a couple of models of computation, computing machines, programs, . . . , we should pause a moment and see what we have achieved.

Where To Go Now?

- We have completed a μML compiler, which generates $\mathcal{L}$(VMP) code from μML programs.
- μML is minimal, but Turing-Complete (has conditionals and procedures)

3.4.5 Turing Machines: A theoretical View on Computation

In this subsection, we will present a very important notion in theoretical Computer Science: The Turing Machine. It supplies a very simple model of a (hypothetical) computing device that can be used to understand the limits of computation.

What have we achieved

- what have we done? We have sketched
  - a concrete machine model (combinatory circuits)
  - a concrete algorithm model (assembler programs)

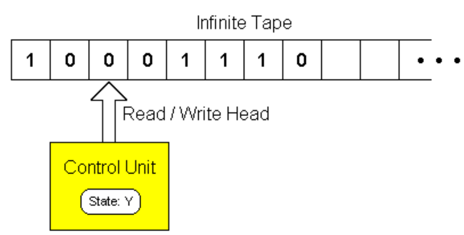
  Evaluation: (is this good?)
- - how does it compare with SML on a laptop?
  - Can we compute all (string/numerical) functions in this model?
  - Can we always prove that our programs do the right thing?
- Towards Theoretical Computer Science (as a tool to answer these)
  - look at a much simpler (but less concrete) machine model (Turing Machine)
  - show that TM can [encode/be encoded in] SML, assembler, Java,. . .
- **Conjecture 477** *[Church/Turing]* *(unprovable, but accepted)*

  All non-trivial machine models and programming languages are equivalent

We want to explore what the "simplest" (whatever that may mean) computing machine could be. The answer is quite surprising, we do not need wires, electricity, silicon, etc; we only need a very simple machine that can write and read to a tape following a simple set of rules.

Turing Machines: The idea

- ▷ Idea: Simulate a machine by a person executing a well-defined procedure!
- ▷ Setup: Person changes the contents of an infinite amount of ordered paper sheets that can contain one of a finite set of symbols.
- ▷ Memory: The person needs to remember one of a finite set of states
- ▷ Procedure: "If your state is 42 and the symbol you see is a '0' then replace this with a '1', remember the state 17, and go to the following sheet."

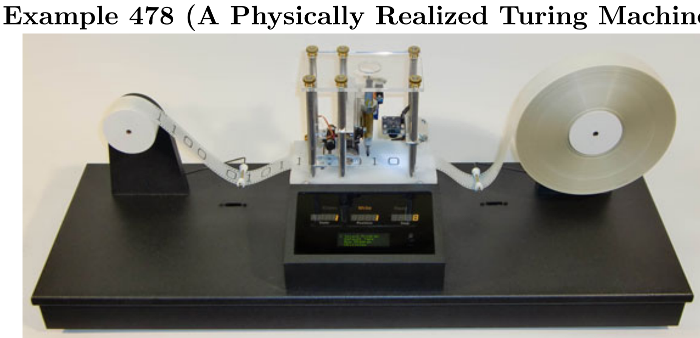


Note that the physical realization of the machine as a box with a (paper) tape is immaterial, it is inspired by the technology at the time of its inception (in the late 1940ies; the age of ticker-tape commuincation).

A Physical Realization of a Turing Machine

- ▷ Note: Turing machine can be built, but that is not the important aspect

▷ **Example 478 (A Physically Realized Turing Machine)**



For more information see `http://aturingmachine.com`.

- ▷ Turing machines are mainly used for thought experiments, where we simulate them in our heads. (or via programs)

To use (i.e. simulate) Turing machines, we have to make the notion a bit more precise.

Turing Machine: The Definition

▷ **Definition 479** A Turing Machine consists of

- ▷ An infinite tape which is divided into cells, one next to the other (each cell contains a symbol from a finite alphabet $\mathcal{L}$ with $\#(\mathcal{L}) \geq 2$ and $0 \in \mathcal{L}$)
- ▷ A head that can read/write symbols on the tape and move left/right.
- ▷ A state register that stores the state of the Turing machine. (finite set of states, register initialized with a special start state)
- ▷ An action table that tells the machine what symbol to write, how to move the head and what its new state will be, given the symbol it has just read on the tape and the state it is currently in. (If no entry applicable the machine will halt)

▷ and now again, mathematically:

▷ **Definition 480** A Turing machine specification is a quintuple $\langle \mathcal{A}, \mathcal{S}, s_0, \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{A}$ is an alphabet, $\mathcal{S}$ is a set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{F} \subseteq \mathcal{S}$ is the set of final states, and $\mathcal{R}$ is a function $\mathcal{R} \colon \mathcal{S} \backslash \mathcal{F} \times \mathcal{A} \to \mathcal{S} \times \mathcal{A} \times \{R, L\}$ called the transition function.

▷ Note: every part of the machine is finite, but it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

To fortify our intuition about the way a Turing machine works, let us consider a concrete example of a machine and look at its computation.

The only variable parts in Definition 479 are the alphabet used for data representation on the tape, the set of states, the initial state, and the actiontable; so they are what we have to give to specify a Turing machine.

Turing Machine

Example 481 with Alphabet $\{0, 1\}$

▷ Given: a series of 1s on the tape (with head initially on the leftmost)

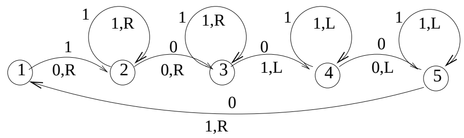
▷ Computation: doubles the 1's with a 0 in between, i.e., "111" becomes "1110111".

▷ The set of states is $\{s_1, s_2, s_3, s_4, s_5, f\}$ (s_1 initial, f final)

▷ Action Table:

Old	Read	Wr.	Mv.	New	Old	Read	Wr.	Mv.	New
s_1	1	0	R	s_2	s_4	1	1	L	s_4
s_2	1	1	R	s_2	s_4	0	0	L	s_5
s_2	0	0	R	s_3	s_5	1	1	L	s_5
s_3	1	1	R	s_3	s_5	0	1	R	s_1
s_3	0	1	L	s_4	s_1	2			f

▷ State Machine:



The computation of the turing machine is driven by the transition funciton: It starts in the initial state, reads the character on the tape, and determines the next action, the character to write, and the next state via the transition function.

Example Computation

▷ $\mathcal{T}$ starts out in s_1, replaces the first 1 with a 0, then

▷ uses s_2 to move to the right, skipping over 1's and the first 0 encountered.

▷ s_3 then skips over the next sequence of 1's (initially there are none) and replaces the first 0 it finds with a 1.

▷ s_4 moves back left, skipping over 1's until it finds a 0 and switches to s_5.

Step	State	Tape	Step	State	Tape
1	s_1	$\boxed{1}1$	9	s_2	$10\boxed{0}1$
2	s_2	$0\boxed{1}$	10	s_3	$100\boxed{1}$
3	s_2	$01\boxed{0}$	11	s_3	$1001\boxed{0}$
4	s_3	$010\boxed{0}$	12	s_4	$100\boxed{1}1$
5	s_4	$01\boxed{0}1$	13	s_4	$10\boxed{0}11$
6	s_5	$0\boxed{1}01$	14	s_5	$1\boxed{0}011$
7	s_5	$\boxed{0}101$	15	s_1	$11\boxed{0}11$
8	s_1	$1\boxed{1}01$		— halt —	

▷ s_5 then moves to the left, skipping over 1's until it finds the 0 that was originally written by s_1.

▷ It replaces that 0 with a 1, moves one position to the right and enters s1 again for another round of the loop.

▷ This continues until s_1 finds a 0 (this is the 0 right in the middle between the two strings of 1's) at which time the machine halts

We have seen that a Turing machine can perform computational tasks that we could do in other programming languages as well. The computation in the example above could equally be expressed in a while loop (while the input string is non-empty) in SW, and with some imagination we could even conceive of a way of automatically building action tables for arbitrary while loops using the ideas above.

What can Turing Machines compute?

▷ Empirically: anything any other program can also compute

- ▷ Memory is not a problem (tape is infinite)
- ▷ Efficiency is not a problem (purely theoretical question)
- ▷ Data representation is not a problem (we can use binary, or whatever symbols we like)

▷ All attempts to characterize computation have turned out to be equivalent

- ▷ primitive recursive functions ([Gödel, Kleene])
- ▷ lambda calculus ([Church])

- ▷ Post production systems ([Post])
- ▷ Turing machines ([Turing])
- ▷ Random-access machine

▷ **Conjecture 482 ([Church/Turing])** *(unprovable, but accepted)*
Anything that can be computed at all, can be computed by a Turing Machine

▷ **Definition 483** We will call a computational system Turing complete, iff it can compute what a Turing machine can.

Note that the Church/Turing hypothesis is a very strong assumption, but it has been born out by experience so far and is generally accepted among computer scientists.

The Church/Turing hypothesis is strengthened by another concept that Alan Turing introduced in [Tur36]: the universal turing machine – a Turing machine that can simulate arbitrary Turing machine on arbitrary input. The universal Turing machine achieves this by reading both the Turing machine specification $\mathcal{T}$ as well as the $\mathcal{I}$ input from its tape and simulates $\mathcal{T}$ on $\mathcal{I}$, constructing the output that $\mathcal{T}$ would have given on $\mathcal{I}$ on the tape. The construction itself is quite tricky (and lengthy), so we restrict ourselves to the concepts involved.

Some researchers consider the universal Turing machine idea to be the origin of von Neumann's architecture of a stored-program computer, which we explored in Subsection 3.4.0.

Universal Turing machines

▷ Note: A Turing machine computes a fixed partial string function.

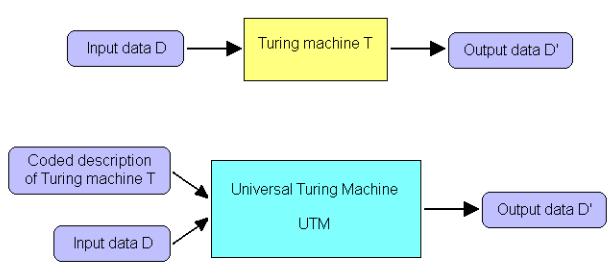
▷ In that sense it behaves like a computer with a fixed program.

▷ Idea: we can encode the action table of any Turing machine in a string.

- ▷ try to construct a Turing machine that expects on its tape
- ▷ a string describing an action table followed by
- ▷ a string describing the input tape, and then
- ▷ computes the tape that the encoded Turing machine would have computed.

▷ **Theorem 484** *Such a Turing machine is indeed possible* *(e.g. with 2 states, 18 symbols)*

▷ **Definition 485** Call it a universal Turing machine (UTM). (it can simulate any TM)



- ▷ UTM accepts a coded description of a Turing machine and simulates the behavior of the machine on the input data.

▷ The coded description acts as a program that the UTM executes, the UTM's own internal program is fixed.

The existence of the UTM is what makes computers fundamentally different from other machines such as telephones, CD players, VCRs, refrigerators, toaster-ovens, or cars.

Indeed the existence of UTMs is one of the distinguishing feature of computing. Whereas other tools are single purpose (or multi-purpose at best; e.g. in the sense of a Swiss army knife, which integrates multiple tools), computing devices can be configured to assume any behavior simply by supplying a program. This makes them universal tools.

Note: that there are very few disciplines that study such universal tools, this makes Computer Science special. The only other discipline with "universal tools" that comes to mind is Biology, where ribosomes read RNA codes and synthesize arbitrary proteins. But for all we know at the moment, RNA codes is linear and therefore Turing completeness of the RNA code is still hotly debated (I am skeptical).

Even in our limited experience from this course, we have seen that we can compile μML to $\mathcal{L}(\texttt{VMP})$ and `SW` to $\mathcal{L}(\texttt{VM})$ both of which we can interpret in `ASM`. And we can write an SML simulator of the `REMA` that closes the circle. So all these languages are equivalent and inter-simulatable. Thus, if we can simulate any of them in Turing machines, then we can simulate any of them.

Of course, not all programming languages are inter-simulatable, for instance, if we had forgotten the jump instructions in $\mathcal{L}(\texttt{VM})$, then we could not compile the control structures of `SW` or μML into $\mathcal{L}(\texttt{VM})$ or $\mathcal{L}(\texttt{VMP})$. So we should read the Church/Turing hypothesis as a statement of equivalence of all non-trivial programming languages.

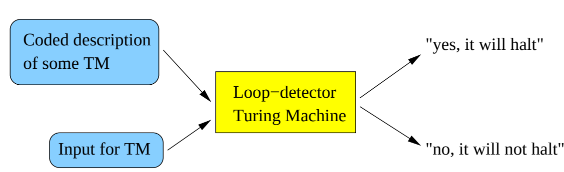
Question: So, if all non-trivial programming languages can compute the same, are there things that none of them can compute? This is what we will have a look at next.

Is there anything that cannot be computed by a TM

▷ ▷ **Theorem 486 (Halting Problem [Tur36])** *No Turing machine can infallibly tell if another Turing machine will get stuck in an infinite loop on some given input.*

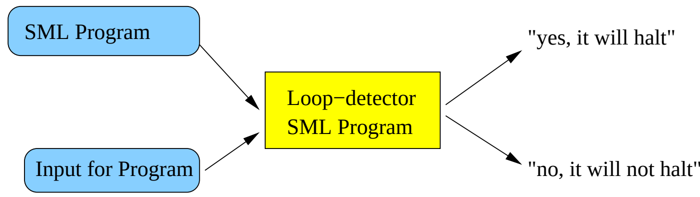
▷ The problem of determining whether a Turing machine will halt on an input is called the halting problem.

▷



▷ Proof:

P.1 let's do the argument with SML instead of a TM
assume that there is a loop detector program written in SML



Using SML for the argument does not really make a difference for the argument, since we believe that Turing machines are inter-simulatable with SML programs. But it makes the argument clearer at the conceptual level. We also simplify the types involved, but taking the argument to be a function of type `string -> string` and its input to be of type `string`, but of course, we only have to exhibit one counter-example to prove the halting problem.

Testing the Loop Detector Program

Proof:

P.1 The general shape of the Loop detector program

```
fun will_halt(program,data) =
    ... lots of complicated code ...
    if ( ... more code ...) then true else false;
will_halt : (string -> string) -> string -> bool
```

test programs	behave exactly as anticipated

`fun halter (s) = "";`
`halter : string -> string`
`fun looper (s) = looper(s);`
`looper : string -> string`
	`will_halt(halter,"");`
	`val true : bool`
	`will_halt(looper,"");`
	`val false : bool`

Consider the following program

```
fun turing (prog) =
   if will_halt(eval(prog),prog) then looper(1) else 1;
turing : string -> string
```

Yeah, so what? what happens, if we feed the `turing` function to itself? □

Observant readers may already see what is going to happen here, we are going for a diagonalization argument, where we apply the function `turing` to itself.

Note that to get the types to work out, we are assuming a function `eval : string -> string -> string` that takes (program) string and compiles it into a function of type `string -> string`. This can be written, since the SML compiler exports access to its internals in the SML runtime.

But given this trick, we can apply `turing` to itself, and get into the well-known paradoxical situation we have already seen for the "set of all sets that do not contain themselves" in Russell's paradox.

What happens indeed?

Proof:

P.2 P.3 P.1
```
fun turing (prog) =
     if will\_halt(eval(prog),prog) then looper(1) else 1;
```

the `turing` function uses `will_halt` to analyze the function given to it.

- ▷ If the function halts when fed itself as data, the `turing` function goes into an infinite loop.
- ▷ If the function goes into an infinite loop when fed itself as data, the `turing` function immediately halts.

But if the function happens to be the `turing` function itself, then

P.2 ▷ the `turing` function goes into an infinite loop if the `turing` function halts (when fed itself as input)

▷ the `turing` function halts if the `turing` function goes into an infinite loop (when fed itself as input)

This is a blatant logical contradiction! Thus there cannot be a `will_halt` function □

The halting problem is historically important, since it is one of the first problems that was shown to be undecidable – in fact Alonzo Church's proof of the undecidability of the λ-calculus was published one month earlier as [Chu36].

Just as the existence of an UTM is a defining characteristic of computing science, the existence of undecidable problems is a (less happy) defining fact that we need to accept about the fundamental nature of computation.

In a way, the halting problem only shows that computation is inherently non-trivial — just in the way sets are; we can play the same diagonalization trick on them and end up in Russell's paradox. So the halting problems should not be seen as a reason to despair on computation, but to rejoice that we are tackling non-trivial problems in Computer Science. Note that there are a lot of problems that are decidable, and there are algorithms that tackle undecidable problems, and perform well in many cases (just not in all). So there is a lot to do; let's get to work.

3.5 The Information and Software Architecture of the Internet and World Wide Web

In the last chapters we have seen how to build computing devices, and how to program them with high-level programming languages. But this is only part of the computing infrastructure we have gotten used to in the last two decades: computers are nowadays globally networked on the Internet, and we use computation on remote computers and information services on the World Wide Web on a day-to-day basis.

In this sectionwe will look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up.

3.5.1 Overview

We start off with a disambiguation of the concepts of Internet and World Wide Web that are often used interchangeably (and thus imprecisely) in the popular discussion. In fact, the form quite different pieces in the general networking infrastructure, with the World Wide Web building on the Internet as one of many services. We will give an overview over the devices and protocols driving the Internet in Subsection 3.5.1 and on the central concepts of the World Wide Web in Subsection 3.5.2.

The Internet and the Web

P.3 ▷ **Definition 487** The Internet is a worldwide computer network that connects hundreds of thousands of smaller networks. (The mother of all networks)

▷ **Definition 488** The World Wide Web (WWW) is the interconnected system of servers that support multimedia documents, i.e. the multimedia part of the Internet.

▷ The Internet and WWWeb form critical infrastructure for modern society and commerce.

▷ The Internet/WWW is huge:

Year	Web	Deep Web	eMail
1999	21 TB	100 TB	11TB
2003	167 TB	92 PB	447 PB
2010	????	?????	?????

▷ We want to understand how it works (services and scalability issues)

.

 336 JACOBS UNIVERSITY

One of the central things to understand about the Internet and the WWWeb is that they have been growing exponentially over the last decades in terms of traffic and available content. In fact, we do not really know how big the Internet/WWWeb are, its distributed, and increasingly commercial nature and global scale make that increasingly difficult to measure.

Of course, we also want to understand the units used in the measurement of the size of the Internet, this is next.

Units of Information

Bit (b)	*binary digit 0/1*
Byte (B)	*8 bit*
2 Bytes	A Unicode character.
10 Bytes	your name.
Kilobyte (KB)	*1,000 bytes OR* 10^3 *bytes*
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (MB)	*1,000,000 bytes OR* 10^6 *bytes*
1 Megabyte	A small novel OR a 3.5 inch floppy disk.
2 Megabytes	A high-resolution photograph.
5 Megabytes	The complete works of Shakespeare.
10 Megabytes	A minute of high-fidelity sound.
100 Megabytes	1 meter of shelved books.
500 Megabytes	A CD-ROM.
Gigabyte (GB)	*1,000,000,000 bytes or* 10^9 *bytes*
1 Gigabyte	a pickup truck filled with books.
20 Gigabytes	A good collection of the works of Beethoven.
100 Gigabytes	A library floor of academic journals.

Terabyte (TB)	*1,000,000,000,000 bytes or* 10^{12} *bytes*
1 Terabyte	50000 trees made into paper and printed.
2 Terabytes	An academic research library.
10 Terabytes	The print collections of the U.S. Library of Congress.
400 Terabytes	National Climactic Data Center (NOAA) database.
Petabyte (PB)	*1,000,000,000,000,000 bytes or* 10^{15} *bytes*
1 Petabyte	3 years of EOS data (2001).
2 Petabytes	All U.S. academic research libraries.
20 Petabytes	Production of hard-disk drives in 1995.
200 Petabytes	All printed material (ever).
Exabyte (EB)	*1,000,000,000,000,000,000 bytes or* 10^{18} *bytes*
2 Exabytes	Total volume of information generated in 1999.
5 Exabytes	All words ever spoken by human beings ever.
300 Exabytes	All data stored digitally in 2007.
Zettabyte (EB)	*1,000,000,000,000,000,000,000 bytes or* 10^{21} *bytes*
2 Zettabytes	Total volume digital data transmitted in 2011
100 Zettabytes	Data equivalent to the human Genome in one body.

 337 JACOBS UNIVERSITY

The information in this table is compiled from various studies, most recently [HL11].

A Timeline of the Internet and the Web

- ▷ Early 1960s: introduction of the network concept
- ▷ 1970: ARPANET, scholarly-aimed networks
- ▷ 62 computers in 1974
- ▷ 1975: Ethernet developed by Robert Metcalfe
- ▷ 1980: TCP/IP
- ▷ 1982: The first computer virus, Elk Cloner, spread via Apple II floppy disks
- ▷ 500 computers in 1983
- ▷ 28,000 computers in 1987
- ▷ 1989: Web invented by Tim Berners-Lee
- ▷ 1990: First Web browser based on HTML developed by Berners-Lee
- ▷ Early 1990s: Andreessen developed the first graphical browser (Mosaic)
- ▷ 1993: The US White House launches its Web site
- ▷ 1993 –: commercial/public web explodes

 338 JACOBS UNIVERSITY

We will now look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up.

3.5.2 Internet Basics

We will show aspects of how the Internet can cope with this enormous growth of numbers of computers, connections and services.

The growth of the Internet rests on three design decisions taken very early on. The Internet

1. is a packet-switched network rather than a network, where computers communicate via dedicated physical communication lines.
2. is a network, where control and administration are decentralized as much as possible.
3. is an infrastructure that only concentrates on transporting packets/datagrams between computers. It does not provide special treatment to any packets, or try to control the content of the packets.

The first design decision is a purely technical one that allows the existing communication lines to be shared by multiple users, and thus save on hardware resources. The second decision allows the administrative aspects of the Internet to scale up. Both of these are crucial for the scalability of the Internet. The third decision (often called "net neutrality") is hotly debated. The defenders cite that net neutrality keeps the Internet an open market that fosters innovation, where as the attackers say that some uses of the network (illegal file sharing) disproportionately consume resources.

Package-Switched Networks

▷ **Definition 489** A packet-switched network divides messages into small network packets that are transported separately and re-assembled at the target.

▷ Advantages:

- ▷ many users can share the same physical communication lines.
- ▷ packets can be routed via different paths. (bandwidth utilization)
- ▷ bad packets can be re-sent, while good ones are sent on. (network reliability)
- ▷ packets can contain information about their sender, destination.
- ▷ no central management instance necessary (scalability, resilience)

 339 JACOBS UNIVERSITY

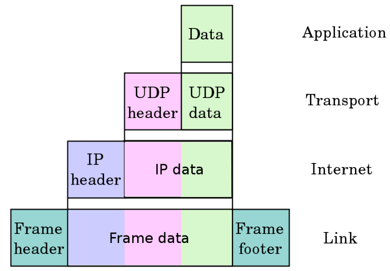
These ideas are implemented in the Internet Protocol Suite, which we will present in the rest of the section. A main idea of this set of protocols is its layered design that allows to separate concerns and implement functionality separately.

The Intenet Protocol Suite

▷ **Definition 490** The Internet Protocol Suite (commonly known as TCP/IP) is the set of communications protocols used for the Internet and other similar networks. It structured into 4 layers.

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	UDP,TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

▷ Layers in TCP/IP: TCP/IP uses encapsulation to provide abstraction of protocols and services.
An application (the highest level of the model) uses a set of protocols to send its data down the layers, being further encapsulated at each level.

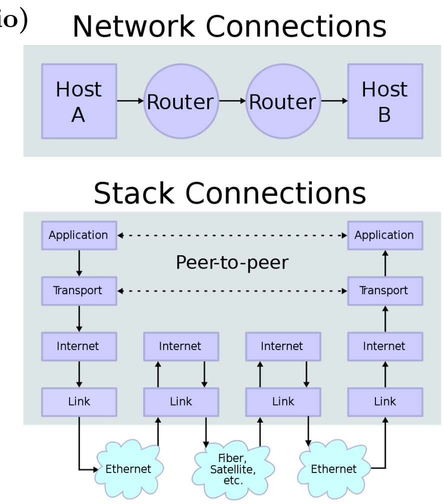


▷ **Example 491 (TCP/IP Scenario)** Consider a situation with two Internet host computers communicate across local network boundaries.

▷ network boundaries are constituted by internetworking gateways (routers).

▷ **Definition 492** A router is a purposely customized computer used to forward data among computer networks beyond directly connected devices.

▷ A router implements the link and internet layers only and has two network connections.



 340

We will now take a closer look at each of the layers shown above, starting with the lowest one.

Instead of going into network topologies, protocols, and their implementation into physical signals that make up the link layer, we only discuss the devices that deal with them. Network Interface controllers are specialized hardware that encapsulate all aspects of link-level communication, and we take them as black boxes for the purposes of this course.

Network Interfaces

▷ The nodes in the Internet are computers, the edges communication channels

▷ **Definition 493** A network interface controller (NIC) is a hardware device that handles an interface to a computer network and thus allows a network-capable device to access that network.

▷ **Definition 494** Each NIC contains a unique number, the media access control address (MAC address), identifies the device uniquely on the network.

▷ MAC addresses are usually 48-bit numbers issued by the manufacturer, they are usually displayed to humans as six groups of two hexadecimal digits, separated by hyphens (-) or colons (:), in transmission order, e.g. 01-23-45-67-89-AB, 01:23:45:67:89:AB.

▷ **Definition 495** A network interface is a software component in the operating system that implements the higher levels of the network protocol (the NIC handles the lower ones).

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

▷ A computer can have more than one network interface. (e.g. a router)

The next layer ist he Internet Layer, it performs two parts: addressing and packing packets.

Internet Protocol and IP Addresses

▷ **Definition 496** The Internet Protocol (IP) is a protocol used for communicating data across a packet-switched internetwork. The Internet Protocol defines addressing methods and structures for datagram encapsulation. The Internet Protocol also routes data packets between networks

▷ **Definition 497** An Internet Protocol (IP) address is a numerical label that is assigned to devices participating in a computer network, that uses the Internet Protocol for communication between its nodes.

▷ An IP address serves two principal functions: host or network interface identification and location addressing.

▷ **Definition 498** The global IP address space allocations are managed by the Internet Assigned Numbers Authority (IANA), delegating allocate IP address blocks to five Regional Internet Registries (RIRs) and further to Internet service providers (ISPs).

▷ **Definition 499** The Internet mainly uses Internet Protocol Version 4 (IPv4) [RFC80], which uses 32-bit numbers (IPv4 addresses) for identification of network interfaces of Computers.

▷ IPv4 was standardized in 1980, it provides 4,294,967,296 (2^{32}) possible unique addresses. With the enormous growth of the Internet, we are fast running out of IPv4 addresses

▷ **Definition 500** Internet Protocol Version 6 (IPv6) [DH98], which uses 128-bit numbers (IPv6 addresses) for identification.

▷ Although IP addresses are stored as binary numbers, they are usually displayed in human-readable notations, such as 208.77.188.166 (for IPv4), and $2001 : db8 : 0 : 1234 : 0 : 567 : 1 : 1$ (for IPv6).

The Internet infrastructure is currently undergoing a dramatic retooling, because we are moving from IPv4 to IPv6 to counter the depletion of IP addresses. Note that this means that all routers and switches in the Internet have to be upgraded. At first glance, it would seem that that this problem could have been avoided if we had only anticipated the need for more the 4 million computers. But remember that TCP/IP was developed at a time, where the Internet did not exist yet, and it's precursor had about 100 computers. Also note that the IP addresses are part of every packet, and thus reserving more space for them would have wasted bandwidth in a time when it was scarce.

We will now go into the detailed structure of the IP packets as an example of how a low-level protocol is structured. Basically, an IP packet has two parts: the "header", whose sequence of bytes is strictly standardized, and the "payload", a segment of bytes about which we only know the length, which is specified in the header.

The Structure of IP Packets

▷ **Definition 501** IP packets are composed of a $160b$ header and a payload. The IPv4 packet header consists of:

b	name	comment
4	version	IPv4 or IPv6 packet
4	Header Length	in multiples 4 bytes (e.g., 5 means 20 bytes)
8	QoS	Quality of Service, i.e. priority
16	length	of the packet in bytes
16	fragid	to help reconstruct the packet from fragments,
3	fragmented	DF $\hat{=}$ "Don't fragment"/MF $\hat{=}$ "More Fragments"
13	fragment offset	to identify fragment position within packet
8	TTL	Time to live (router hops until discarded)
8	protocol	TCP, UDP, ICMP, etc.
16	Header Checksum	used in error detection,
32	Source IP	
32	target IP	
...	optional flags	according to header length

▷ Note that delivery of IP packets is not guaranteed by the IP protocol.

As the internet protocol only supports addressing, routing, and packaging of packets, we need another layer to get services like the transporting of files between specific computers. Note that the IP protocol does not guarantee that packets arrive in the right order or indeed arrive at all, so the transport layer protocols have to take the necessary measures, like packet re-sending or handshakes, ...

The Transport Layer

▷ **Definition 502** The transport layer is responsible for delivering data to the appropriate application process on the host computers by forming data packets, and adding source and destination port numbers in the header.

▷ **Definition 503** The internet protocol mainly suite uses the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols at the transport layer.

▷ TCP is used for communication, UDP for multicasting and broadcasting.

▷ TCP supports virtual circuits, i.e. provide connection oriented communication over an underlying packet oriented datagram network. (hide/reorder packets)

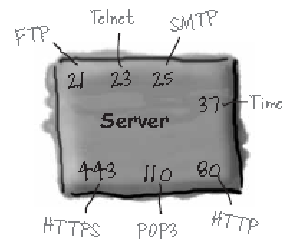
▷ TCP provides end-to-end reliable communication (error detection & automatic repeat)

We will see that there are quite a lot of services at the network application level. And indeed, many web-connected computers run a significant subset of them at any given time, which could lead to problems of determining which packets should be handled by which service. The answer to this problem is a system of "ports" (think pigeon holes) that support finer-grained addressing to the various services.

Ports

▷ **Definition 504** To separate the services and protocols of the network application layer, network interfaces assign them specific port, referenced by a number.

▷ **Example 505** We have the following ports in common use on the Internet



Port	use	comment
22	SSH	remote shell
53	DNS	Domain Name System
80	HTTP	World Wide Web
443	HTTPS	HTTP over SSL

On top of the transport-layer services, we can define even more specific services. From the perspective of the internet protocol suite this layer is unregulated, and application-specific. From a user perspective, many useful services are just "applications" and live at the application layer.

The Application Layer

▷ **Definition 506** The application layer of the internet protocol suite contains all protocols and methods that fall into the realm of process-to-process communications via an Internet Protocol (IP) network using the Transport Layer protocols to establish underlying host-to-host connections.

▷ **Example 507 (Some Application Layer Protocols and Services)**

BitTorrent	Peer-to-peer	Atom	Syndication
DHCP	Dynamic Host Configuration	DNS	Domain Name System
FTP	File Transfer Protocol	HTTP	HyperText Transfer
IMAP	Internet Message Access	IRCP	Internet Relay Chat
NFS	Network File System	NNTP	Network News Transfer
NTP	Network Time Protocol	POP	Post Office Protocol
RPC	Remote Procedure Call	SMB	Server Message Block
SMTP	Simple Mail Transfer	SSH	Secure Shell
TELNET	Terminal Emulation	WebDAV	Write-enabled Web

We will now go into the some of the most salient services on the network application layer.

The domain name system is a sort of telephone book of the Internet that allows us to use symbolic names for hosts like `kwarc.info` instead of the IP number 212.201.49.189.

Domain Names

▷ **Definition 508** The DNS (Domain Name System) is a distributed set of servers that provides the mapping between (static) IP addresses and domain names.

▷ **Example 509** e.g. `www.kwarc.info` stands for the IP address 212.201.49.189.

▷ **Definition 510** Domain names are hierarchically organized, with the most significant part (the top-level domain TLD) last.

▷ networked computers can have more than one DNS name. (virtual servers)

▷ Domain names must be registered to ensure uniqueness (registration fees vary, cybersquatting)

▷ **Definition 511** ICANN is a non-profit organization was established to regulate human-friendly domain names. It approves top-level domains, and corresponding domain name registrars and delegates the actual registration to them.

 347 JACOBS UNIVERSITY

Let us have a look at a selection of the top-level domains in use today.

Domain Name Top-Level Domains

▷ `.com` ("commercial") is a generic top-level domain. It was one of the original top-level domains, and has grown to be the largest in use.

▷ `.org` ("organization") is a generic top-level domain, and is mostly associated with non-profit organizations. It is also used in the charitable field, and used by the open-source movement. Government sites and Political parties in the US have domain names ending in .org

▷ `.net` ("network") is a generic top-level domain and is one of the original top-level domains. Initially intended to be used only for network providers (such as Internet service providers). It is still popular with network operators, it is often treated as a second `.com`. It is currently the third most popular top-level domain.

▷ `.edu` ("education") is the generic top-level domain for educational institutions, primarily those in the United States. One of the first top-level domains, .edu was originally intended for educational institutions anywhere in the world. Only post-secondary institutions that are accredited by an agency on the U.S. Department of Education's list of nationally recognized accrediting agencies are eligible to apply for a `.edu` domain.

▷ `.info` ("information") is a generic top-level domain intended for informative website's, although its use is not restricted. It is an unrestricted domain, meaning that anyone can obtain a second-level domain under .info. The .info was one of many extension(s) that was meant to take the pressure off the overcrowded .com domain.

▷ `.gov` ("government") a generic top-level domain used by government entities in the United States. Other countries typically use a second-level domain for this purpose, e.g., `.gov.uk` for the United Kingdom. Since the United States controls the `.gov` Top Level Domain, it would be impossible for another country to create a domain ending in `.gov`.

▷ `.biz` ("business") the name is a phonetic spelling of the first syllable of "business". A generic top-level domain to be used by businesses. It was created due to the demand for good domain names available in the `.com` top-level domain, and to provide an alternative to businesses whose preferred `.com` domain name which had already been registered by another.

▷ `.xxx` ("porn") the name is a play on the verdict "X-rated" for movies. A generic top-level domain to be used for sexually explicit material. It was created in 2011 in the hope to move sexually explicit material from the "normal web". But there is no mandate for porn to be restricted to the `.xxx` domain, this would be difficult due to problems of definition, different jurisdictions, and free speech issues.

 348 JACOBS UNIVERSITY

Note: Anybody can register a domain name from a registrar against a small yearly fee. Domain names are given out on a first-come-first-serve basis by the domain name registrars, which usually also offer services like domain name parking, DNS management, URL forwarding, etc.

The next application-level service is the SMTP protocol used for sending e-mail. It is based on the telnet protocol for remote terminal emulation which we do not discuss here.

telnet is one of the oldest protocols, which uses TCP directly to send text-based messages between a terminal client (on the local host) and a terminal server (on the remote host). The operation of a remote terminal is the following: the terminal server on the remote host receives commands from the terminal client on the local host, executes them on the remote host and sends back the results to the client on the local host.

A Protocol Example: SMTP over telnet

- ▷ We call up the telnet service on the Jacobs mail server

```
telnet exchange.jacobs-university.de 25
```

- ▷ it identifies itself (have some patience, it is very busy)

```
Trying 10.70.0.128...
Connected to exchange.jacobs-university.de.
Escape character is '^]'.
220 SHUBCAS01.jacobs.jacobs-university.de
Microsoft ESMTP MAIL Service ready at Tue, 3 May 2011 13:51:23 +0200
```

- ▷ We introduce ourselves politely (but we lie about our identity)

```
helo mailhost.domain.tld
```

- ▷ It is really very polite.

```
250 SHUBCAS04.jacobs.jacobs-university.de Hello [10.222.1.5]
```

- ▷ We start addressing an e-mail (again, we lie about our identity)

```
mail from: user@domain.tld
```

- ▷ this is acknowledged

```
250 2.1.0 Sender OK
```

- ▷ We set the recipient (the real one, so that we really get the e-mail)

```
rcpt to: m.kohlhase@jacobs-university.de
```

- ▷ this is acknowledged

```
250 2.1.0 Recipient OK
```

- ▷ we tell the mail server that the mail data comes next

```
data
```

- ▷ this is acknowledged

```
354 Start mail input; end with <CRLF>.<CRLF>
```

- ▷ Now we can just type the a-mail, optionally with Subject, date,...

```
Subject: Test via SMTP

and now the mail body itself
.
```

- ▷ And a dot on a line by itself sends the e-mail off

```
250 2.6.0 <ed73c3f3-f876-4d03-98f2-e5ad5bbb6255@SHUBCAS04.jacobs.jacobs-university.de>
[InternalId=965770] Queued mail for delivery
```

▷ That was almost all, but we close the connection (this is a telnet command)

```
quit
```

▷ our terminal server (the telnet program) tells us

```
221 2.0.0 Service closing transmission channel
Connection closed by foreign host.
```

 349

Essentially, the SMTP protocol mimics a conversation of polite computers that exchange messages by reading them out loud to each other (including the addressing information).

We could go on for quite a while with understanding one Internet protocol after each other, but this is beyond the scope of this course (indeed there are specific courses that do just that). Here we only answer the question where these protocols come from, and where we can find out more about them.

Internet Standardization

▷ Question: Where do all the protocols come from? (someone has to manage that)

▷ **Definition 512** The Internet Engineering Task Force (IETF) is an open standards organization that develops and standardizes Internet standards, in particular the TCP/IP and Internet protocol suite.

▷ All participants in the IETF are volunteers (usually paid by their employers)

▷ Rough Consensus and Running Code: Standards are determined by the "rough consensus method" (consensus preferred, but not all members need agree) IETF is interested in practical, working systems that can be quickly implemented.

▷ Idea: running code leads to rough consensus or vice versa.

▷ **Definition 513** The standards documents of the IETF are called Request for Comments (RFC). (more than 6300 so far; see `http://www.rfc-editor.org/`)

 350

This concludes our very brief exposition of the Internet. The essential idea is that it consists of a decentrally managed, packet-switched network whose function and value is defined in terms of the Internet protocol suite.

3.5.3 Basic Concepts of the World Wide Web

The World Wide Web (WWWeb) is the hypertext/multimedia part of the Internet. It is implemented as a service on top of the Internet (at the aplication level) based on specific protocols and markup formats for documents.

Concepts of the World Wide Web

▷ **Definition 514** A web page is a document on the WWWeb that can include multimedia data and hyperlinks.

▷ **Definition 515** A web site is a collection of related Web pages usually designed or controlled by the same individual or company.

▷ a web site generally shares a common domain name.

▷ **Definition 516** A hyperlink is a reference to data that can immediately be followed by the user or that is followed automatically by a user agent.

▷ **Definition 517** A collection text documents with hyperlinks that point to text fragments within the collection is called a hypertext. The action of following hyperlinks in a hypertext is called browsing or navigating the hypertext.

▷ In this sense, the WWWeb is a multimedia hypertext.

Addressing on the World Wide Web

The essential idea is that the World Wide Web consists of a set of resources (documents, images, movies, etc.) that are connected by links (like a spider-web). In the WWWeb, the the links consist of pointers to addresses of resources. To realize them, we only need addresses of resources (much as we have IP numbers as addresses to hosts on the Internet).

Uniform Resource Identifier (URI), Plumbing of the Web

▷ **Definition 518** A uniform resource identifier (URI) is a global identifiers of network-retrievable documents (web resources). URIs adhere a uniform syntax (grammar) defined in RFC-3986 [BLFM05]. Grammar Rules contain:
URI :== scheme, ′ :′, hierPart, [′?′ query], [′#′ fragment] hier − part :== ′//′ (pathAbempty | pathAbsolute | pathRootless | pathEmpty)

▷ **Example 519** The following are two example URIs and their component parts:

```
  http://example.com:8042/over/there?name=ferret#nose
  \__/   \______________/\_________/ \_________/ \__/
   |           |              |           |        |
scheme     authority        path        query   fragment
   |___   _________________|_________
  /    \ /                           \
 mailto:m.kohlhase@jacobs-university.de
```

Note: URIs only identify documents, they do not have to be provide access to them (e.g. in a browser).

The definition above only specifies the structure of a URI and its functional parts. It is designed to cover and unify a lot of existing addressing schemes, including URLs (which we cover next), ISBN numbers (book identifiers), and mail addresses.

In many situations URIs still have to be entered by hand, so they can become quite unwieldy. Therefore there is a way to abbreviate them.

Relative URIs

▷ ▷ **Definition 520** URIs can be abbreviated to relative URIs; missing parts are filled in from the context

▷ **Example 521** Relative URIs are more convenient to write

relative URI	abbreviates	in context
`#foo`	⟪`current-file`⟫`#foo`	curent file
`../bar.txt`	`file:///home/kohlhase/foo/bar.txt`	file system
`../bar.html`	`http://example.org/foo/bar.html`	on the web

Note that some forms of URIs can be used for actually locating (or accessing) the identified resources, e.g. for retrieval, if the resource is a document or sending to, if the resource is a mailbox. Such URIs are called "uniform resource *locators*", all others "uniform resource *locators*".

Uniform Resource Names and Locators

- **Definition 522** A uniform resource locator (URL) is a URI that that gives access to a web resource, by specifying an access method or location. All other URIs are called uniform resource names (URN).
- Idea: A URN defines the identity of a resource, a URL provides a method for finding it.
- **Example 523** The following URI is a URL (try it in your browser)

  `http://kwarc.info/kohlhase/index.html`

- **Example 524** `urn:isbn:978-3-540-37897-6` only identifies [Koh06] (it is in the library)
- **Example 525** URNs can be turned into URL via a catalog service, e.g. `http://wm-urn.org/urn:isbn:978-3-540-37897-6`
- Note: URI/URLs are one of the core features of the web infrastructure, they are considered to be the plumbing of the WWWeb. (direct the flow of data)

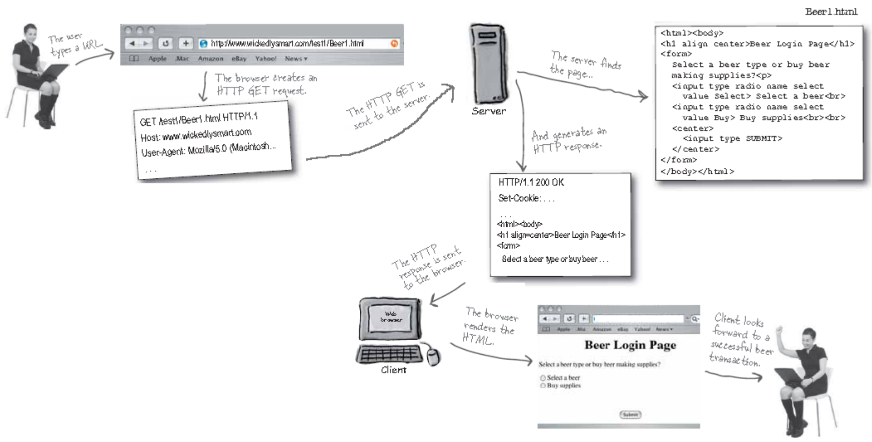
Historically, started out as URLs as short strings used for locating documents on the Internet. The generalization to identifiers (and the addition of URNs) as a concept only came about when the concepts evolved and the application layer of the Internet grew and needed more structure.

Note that there are two ways in URIs can fail to be resource locators: first, the scheme does not support direct access (as the ISBN scheme in our example), or the scheme specifies an access method, but address does not point to an actual resource that could be accessed. Of course, the problem of "dangling links" occurs everywhere we have addressing (and change), and so we will neglect it from our discussion. In practice, the URL/URN distinction is mainly driven by the scheme part of a URI, which specifies the access/identification scheme.

Running the World Wide Web

The infrastructure of the WWWeb relies on a client-server architecture, where the servers (called web servers) provide documents and the clients (usually web browsers) present the documents to the (human) users. Clients and servers communicate via the http protocol. We give an overview via a concrete example before we go into details.

The World Wide Web as a Client/Server System



 355 JACOBS UNIVERSITY

We will now go through and introduce the infrastructure components of the WWWeb in the order we encounter them. We start with the user agent; in our example the web browser used by the user to request the web page by entering its URL into the URL bar.

Web Browsers

▷ **Definition 526** A web Browser is a software application for retrieving, presenting, and traversing information resources on the World Wide Web, enabling users to view Web pages and to jump from one page to another.

▷ Practical Browser Tools:

- ▷ Status Bar: security info, page load progress
- ▷ Favorites (bookmarks)
- ▷ View Source: view the code of a Web page
- ▷ Tools/Internet Options, history, temporary Internet files, home page, auto complete, security settings, programs, etc.

▷ **Example 527 (Common Browsers)** ▷ `MSInternetExplorer` is provided by Microsoft for `Windows` (very common)

- ▷ `FireFox` is an open source browser for all platforms, it is known for its standards compliance.
- ▷ `Safari` is provided by Apple for `MacOSX` and `Windows`
- ▷ `Chrome` is a lean and mean browser provided by Google
- ▷ `WebKit` is a library that forms the open source basis for `Safari` and `Chrome`.

 356 JACOBS UNIVERSITY

The web browser communicates with the web server through a specialized protocol, the hypertext transfer protocol, which we cover now.

HTTP: Hypertext Transfer Protocol

▷ **Definition 528** The Hypertext Transfer Protocol (HTTP) is an application layer protocol for distributed, collaborative, hypermedia information systems.

▷ June 1999: HTTP/1.1 is defined in RFC 2616 [FGM+99].

Definition 529 HTTP is used by a client (called user agent) to access web resources (addressed by Uniform Resource Locators (URLs)) via a http request. The web server answers by supplying the resource

▷ Most important HTTP requests (5 more less prominent)

GET	Requests a representation of the specified resource.	safe
PUT	Uploads a representation of the specified resource.	idempotent
DELETE	Deletes the specified resource.	idempotent
POST	Submits data to be processed (e.g., from a web form) to the identified resource.	

▷ **Definition 530** We call a HTTP request safe, iff it does not change the state in the web server. (except for server logs, counters,...; no side effects)

▷ **Definition 531** We call a HTTP request idempotent, iff executing it twice has the same effect as executing it once.

▷ HTTP is a stateless protocol (very memory-efficient for the server.)

 357

Finally, we come to the last component, the web server, which is responsible for providing the web page requested by the user.

Web Servers

▷ **Definition 532** A web server is a network program that delivers web pages and supplementary resources to and receives content from user agents via the hypertext transfer protocol.

▷ **Example 533 (Common Web Servers)**
- ▷ apache is an open source web server that serves about 60% of the WWWeb.
- ▷ IIS is a proprietary server provided by Microsoft.
- ▷ nginx is a lightweight open source web server.

▷ Even though web servers are very complex software systems, they come preinstalled on most UNIX systems and can be downloaded for Windows [XAM].

 358

Now that we have seen all the components we fortify our intuition of what actually goes down the net by tracing the http messages.

Example: An http request in real life

▷ Connect to the web server (port 80) (so that we can see what is happening)

```
telnet www.kwarc.info 80
```

▷ Send off the GET request

```
GET /teaching/GenCS2.html http/1.1
Host: www.kwarc.info
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.4)
  Gecko/20100413 Firefox/3.6.4
```

▷ Response from the server

```
HTTP/1.1 200 OK
Date: Mon, 03 May 2010 06:48:36 GMT
```

```
Server: Apache/2.2.9 (Debian) DAV/2 SVN/1.5.1 mod_fastcgi/2.4.6 PHP/5.2.6-1+lenny8 with
    Suhosin-Patch mod_python/3.3.1 Python/2.5.2 mod_ssl/2.2.9 OpenSSL/0.9.8g
Last-Modified: Sun, 02 May 2010 13:09:19 GMT
ETag: "1c78b-db1-4859c2f221dc0"
Accept-Ranges: bytes
Content-Length: 3505
Content-Type: text/html

<!--This file was generated by ws2html.xsl. Do NOT edit manually! -->
<html xmlns="http://www.w3.org/1999/xhtml"><head>...</head></html>
```


Multimedia Documents on the World Wide Web

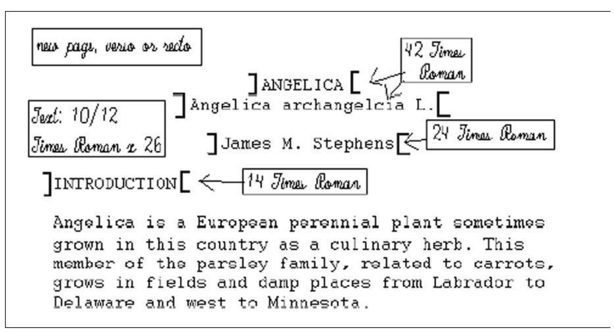
We have seen the client-server infrastructure of the WWWeb, which essentially specifies how hypertext documents are retrieved. Now we look into the documents themselves.

In Subsection 2.4.2 have already discussed how texts can be encoded in files. But for the rich docments we see on the WWWeb, we have to realize that documents are more than just sequences of characters. This is traditionally captured in the notion of document markup.

Document Markup

▷ **Definition 534 (Document Markup)** Document markupmarkup is the process of adding codes (special, standardized character sequences) to a document to control the structure, formatting, or the relationship among its parts.

▷ **Example 535** A text with markup codes (for printing)



There are many systems for document markup ranging from informal ones as in Definition 534 that specify the intended document appearance to humans – in this case the printer – to technical ones which can be understood by machines but serving the same purpose.

WWWeb documents have a specialized markup language that mixes markup for document structure with layout markup, hyper-references, and interaction. The HTML markup elements always concern text fragments, they can be nested but may not otherwise overlap. This essentially turns a text into a document tree.

HTML: Hypertext Markup Language

▷ **Definition 536** The HyperText Markup Language (HTML), is a representation format for web pages. Current version 4.01 is defined in [RHJ98].

▷ **Definition 537 (Main markup elements of HTML)** HTML marks up the structure and appearance of text with tags of the form `<el>` (begin) and `</el>` (end), where `el` is one of the following

structure	`html`,`head`, `body`	metadata	`title`, `link`, `meta`
headings	`h1`, `h2`, ..., `h6`	paragraphs	`p`, `br`
lists	`ul`, `ol`, `dl`, ..., `li`	hyperlinks	`a`
images	`img`	tables	`table`, `th`, `tr`, `td`, ...
styling	`style`, `div`, `span`	old style	`b`, `u`, `tt`, `i`, ...
interaction	`script`	forms	`form`, `input`, `button`

▷ **Example 538** A (very simple) HTML file with a single paragraph.

```
<html>
  <body>
    <p>Hello GenCS students!</p>
  </body>
</html>
```

 361 JACOBS UNIVERSITY

HTML was created in 1990 and standardized in version 4 in 1997. Since then there has HTML has been basically stable, even though the WWWeb has evolved considerably from a web of static web pages to a Web in which highly dynamic web pages become user interfaces for web-based applications and even mobile applets. Acknowledging the growing discrepancy, the W3C has started the standardization of version 5 of HTML.

HTML5: The Next Generation HTML

▷ **Definition 539** The HyperText Markup Language (HTML5), is believed to be the next generation of HTML. It is defined by the W3C and the WhatWG.

▷ HTML5 includes support for

- ▹ audio/video without plugins,
- ▹ a canvas element for scriptable, 2D, bitmapped graphics
- ▹ *SVG* for Scalable Vector Graphics
- ▹ MathML inline and display-style mathematical formulae

▷ The W3C is expected to issue a "recommendation" that standardizes HTML5 in 2014.

▷ Even though HTML5 is not formally standardized yet, almost all major web browsers already implement almost all of HTML5.

 362 JACOBS UNIVERSITY

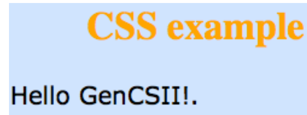
As the WWWeb evolved from a hypertext system purely aimed at human readers to an Web of multimedia documents, where machines perform added-value services like searching or aggregating, it became more important that machines could understand critical aspects web pages. One way to facilitate this is to separate markup that specifies the content and functionality from markup that specifies human-oriented layout and presentation (together called "styling"). This is what "cascading style sheets" set out to do. Another motivation for CSS is that we often want the styling of a web page to be customizable (e.g. for vision-impaired readers).

CSS: Cascading Style Sheets

▷ Idea: Separate structure/function from appearance.

Definition 540 The Cascading Style Sheets (CSS), is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents. Current version 2.1 is defined in [BCHL09].

▷ **Example 541** Our text file from Example 538 with embedded CSS




```
<html>
  <head>
    <style type="text/css">
       body {background-color:#d0e4fe;}
       h1 {color:orange;
                 text-align:center;}
       p {font-family:"Verdana";
                 font-size:20px;}
   </style>
  </head>
  <body>
    <h1>CSS example</h1>
    <p>Hello GenCSII!.</p>
  </body>
</html>
```


With the technology described so far, we can basically build static web pages: hypertexts, where interaction is limited to navigation via hyperlinks: they are highlighted in the layout, and when we select them (usually by clicking), we navigate to the link target (to a new web page or a text fragment in the same page). With a few additions to the technology, web pages can be made much more interactive and versatile, up to a point, where they can function as interfaces to applications which run on a web server.

3.5.4 Web Applications

In this subsectionwe show how with a few additions to the basic WWWeb infrastructure introduced in Subsection 3.5.2, we can turn web pages into web-based applications that can be used without having to install additional software.

The first thing we need is a means to send information back to the web server, which can be used as input for the web application. Fortunately, this is already forseen by the HTML format.

HTML Forms: Submitting Information to the Web Server

▷ **Example 542** Forms contain input fields and explanations.

```
<form name="input" action="html_form_submit.asp" method="get">
  Username: <input type="text" name="user" />
  <input type="submit" value="Submit" />
</form>
```

The result is a form with three elements: a text, an input field, and a submit button, that will trigger a HTTP GET request to the URL specified in the `action` attribute.

As the WWWeb is based on a client-server architecture, computation in web applications can be executed either on the client (the web browser) or the server (the web server). For both we have a special technology; we start with computation on the web server.

Server-Side Scripting: Programming Web Pages

- ▷ Idea: Why write HTML pages if we can also program them! (easy to do)
- ▷ **Definition 543** A server-side scripting framework is a web server extension that generates web pages upon HTTP GET requests.
- ▷ **Example 544** `perl` is a scripting language with good string manipulation facilities. `perl` CGI is an early server-side scripting framework based on this.
- ▷ Server-side scripting frameworks allow to make use of external resources (e.g. databases or data feeds) and computational services during web page generation.
- ▷ Problem: Most web page content is static (page head, text blocks, etc.) (and no HTML editing support in program editors)
- ▷ Idea: Embed program snippets into HTML pages. (only execute these, copy rest)
- ▷ **Definition 545** A server-side scripting language is a server side scripting framework where web pages are generated from HTML documents with embedded program fragments that are executed in context during web page generation.
- ▷ Note: No program code is left in the resulting web page after generation (important security concern)

 365

To get a concrete intuition on the possibilities of server-side scripting frameworks, we will present PHP, a commonly used open source scripting framework. There are many other examples, but they mainly differ on syntax and advanced features.

PHP, a Server-Side Scripting Language

- ▷ **Definition 546** PHP (originally "Programmable Home Page Tools", later "PHP: Hypertext Processor") is a server-side scripting language with a C-like syntax. PHP code is embedded into HTML via special "tags" `<?php` and `?>`
- ▷ **Example 547** The following PHP program uses `echo` for string output

```
<html>
  <body><?php echo 'Hello␣world';?></body>
</html>
```

- ▷ **Example 548** We can access the server clock in PHP (and manipulate it)

```
<?php
$tomorrow = mktime(0,0,0,date("m"),date("d")+1,date("Y"));
echo "Tomorrow␣is␣".date("d.␣m.␣Y", $tomorrow);
?>
This fragment inserts tomorrow's␣date␣into␣a␣web␣page
```

- ▷ **Example 549** We can generate pages from a database (here `MySQL`)

```
<?php
$con = mysql_connect("localhost","peter","abc123");
if (!$con)
  {
  die('Could␣not␣connect:␣' . mysql_error());
  }
```

```
mysql_select_db("my_db", $con);

$result = mysql_query("SELECT␣*␣FROM␣Persons");

while($row = mysql_fetch_array($result))
  {
  echo $row['FirstName'] . "␣" . $row['LastName'];
  echo "<br␣/>";
  }

mysql_close($con);
?>
```

▷ **Example 550** We can even send e-mail via this e-mail form.

```
<html><body>
<?php
if (isset($_REQUEST['email']))//if "email" is filled out, send email
  {//send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail("someone@example.com", $subject,
  $message, "From:" . $email);
  echo "Thank␣you␣for␣using␣our␣mail␣form";}
else //if "email" is not filled out, display the form
  {echo "<form␣method='post'␣action='mailform.php'>
␣␣␣Email:␣<input␣name='email'␣type='text'␣/><br␣/>
␣␣␣Subject:␣<input␣name='subject'␣type='text'␣/><br␣/>
␣␣␣Message:<br␣/>
␣␣␣<textarea␣name='message'␣rows='15'␣cols='40'>
␣␣␣</textarea><br␣/>
␣␣␣<input␣type='submit'␣/>
␣␣␣</form>";}
?>
</body></html>
```


With server-side scripting frameworks like PHP, we can already build web applications, which we now define.

Web Applications: Using Applications without Installing

▷ **Definition 551** A web application is a website that serves as a user interface for a server-based application using a web browser as the client.

▷ **Example 552** Commonly used web applications include

- ▷ `http://ebay.com`; auction pages are generated from databases
- ▷ `http://www.weather.com`; weather information generated weather feeds
- ▷ `http://slashdot.org`; aggregation of news feeds/discussions
- ▷ `http://sf.net`; source code hosting and project management

Common Traits: pages generated from databases and external feeds, content submission via HTML forms, file upload

▷▷ **Definition 553** A web application framework is a software framework for creating web applications.

▷ **Example 554** The LAMP stack is a web application framework based on `linux`, `apache`, `MySQL`, and PHP.

▷ **Example 555** A variant of the LAMP stack is available for `Windows` as XAMPP [XAM].

Indeed, the first web applications were essentially built in this way. Note however, that as we remarked above, no PHP code remains in the generated web pages, which thus "look like" static web pages to the client, even though they were generated dynamically on the server.

There is one problem however with web applications that is difficult to solve with the technologies so far. We want web applications to give the user a consistent user experience even though they are made up of multiple web pages. In a regular application we we only want to login once and expect the application to remember e.g. our username and password over the course of the various interactions with the system. For web applications this poses a technical problem which we now discuss.

State in Web Applications and Cookies

- Recall: Web applications contain multiple pages, HTTP is a stateless protocol.
- Problem: how do we pass state between pages? (e.g. username, password)
- Simple Solution: Pass information along in query part of page URLs.
- **Example 556 (HTTP GET for Single Login)** Since we are generating pages we can generated augmented links

```
<a href="http://example.org/more.html?user=joe,pass=hideme">... more</a>
```

  Problem: only works for limited amounts of information and for a single session
- Other Solution: Store state persistently on the client hard disk
- **Definition 557** A cookie is a text file stored on the client hard disk by the web browser. Web servers can request the browser to store and send cookies.
- cookies are data not programs, they do not generate pop-ups or behave like viruses, but they can include your log-in name and browser preferences
- cookies can be convenient, but they can be used to gather information about you and your browsing habits
- **Definition 558** third party cookies are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)

Note that that both solutions to the state problem are not ideal, for usernames and passwords the URL-based solution is particularly problematic, since HTTP transmits URLs in GET requests without encryption, and in our example passwords would be visible to anybody with a packet sniffer. Here cookies are little better as cookies, since they can be requested by any website you visit.

We now turn to client-side computation

One of the main advantages of moving documents from their traditional ink-on-paper form into an electronic form is that we can interact with them more directly. But there are many more interactions than just browsing hyperlinks we can think of: adding margin notes, looking up definitions or translations of particular words, or copy-and-pasting mathematical formulae into a computer algebra system. All of them (and many more) can be made, if we make documents programmable. For that we need three ingredients: *i)* a machine-accessible representation of the document structure, and *ii)* a program interpreter in the web browser, and *iii)* a way to send programs to the browser together with the documents. We will sketch the WWWeb solution to this in the following.

Dynamic HTML

- Observation: The nested, markup codes turn HTML documents into trees.
- **Definition 559** The document object model (DOM) is a data structure for the HTML document tree together with a standardized set of access methods.
- Note: All browsers implement the DOM and parse HTML documents into it; only then is the DOM rendered for the user.
- Idea: generate parts of the web page dynamically by manipulating the DOM.
- **Definition 560** JavaScript is an object-oriented scripting language mostly used to enable programmatic access to the DOM in a web browser.
- JavaScript is standardized by ECMA in [ECM09].
- **Example 561** We write the some text into a HTML document object (the document API)

```
<html>
 <head>
  <script type="text/javascript">document.write("Dynamic␣HTML!");</script>
 </head>
 <body><!-- nothing here; will be added by the script later --></body>
</html>
```

 369

Let us fortify our intuition about dynamic HTML by going into a more involved example.

Applications and useful tricks in Dynamic HTML

- **Example 562** hide document parts by setting CSS `style` attributes to `display:none`

```
<html>
  <head>
     <style type="text/css">#dropper { display: none; }</style>
     <script language="JavaScript" type="text/javascript">
        function toggleDiv(element){
            if(document.getElementById(element).style.display = 'none')
                 {document.getElementById(element).style.display = 'block'}
            else if(document.getElementById(element).style.display = 'block')
                 {document.getElementById(element).style.display = 'none'}}
      </script>
  </head>
  <body>
     <div onClick="toggleDiv('dropper');">...more </div>
     <div id="dropper">
        <p>Now you see it!</p>
    </div>
  </body>
</html>
```

  Application: write "gmail" or "google docs" as JavaScript enhanced web applications. (client-side computation for immediate reaction)

- Current Megatrend: Computation in the "cloud", browsers (or "apps") as user interfaces

 370

Current web applications include simple office software (word processors, online spreadsheets, and presentation tools), but can also include more advanced applications such as project management, computer-aided design, video editing and point-of-sale. These are only possible if we carefully balance the effects of server-side and client-side computation. The former is needed for computational resources and data persistence (data can be stored on the server) and the latter to keep personal information near the user and react to local context (e.g. screen size).

We have now seen the basic architecture and protocols of the World Wide Web. This covers basic interaction with web pages via browsing of links, as has been prevalent until around 1995. But this is not now we interact with the web nowadays; instead of browsing we use web search engines like Google or Yahoo, we will cover next how they work.

3.5.5 Introduction to Web Search

In this subsectionwe will present an overview over the basic mechanisms of web search engines. They are important to understanding the WWWeb, since they have replaced web portals as the entry points of the WWWeb.

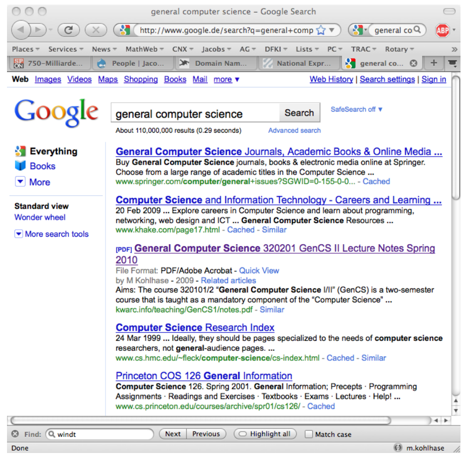
Web Search Engines

- ▷ **Definition 563** A web search engine is a web application designed to search for information on the World Wide Web.
- ▷ **Definition 564 (Types of Search Engines)** We call a web search engine
  - ▷ human-organized, if documents are categorized by subject-area experts, (e.g. Open Directory, About)
  - ▷ computer-created Software spiders crawl the web for documents and categorize pages, (e.g. Google)
  - ▷ hybrid if it combines the two categories above
  - ▷ metasearch if ti sends direct queries to multiple search engines and aggregates/clusters results, (e.g. Copernic, Vivisimo, Mamma)

 371 JACOBS UNIVERSITY

We wil concentrate on computer-created search engines, since they are prevalent on the WWWeb nowadays. Let us see how they work.

Functional Schema of Web Search Engines



▷ Web search engines usually operate in four phases/components

1. Data Acquisition: a web crawler finds and retrieves (changed) web pages
2. Search in Index: write an index and search there.
3. Sort the hits: e.g. by importance
4. Answer composition: present the hits (and add advertisement)

We will now go through the four phases in turn and discuss specifics

Data Acquisition for Web Search Engines: Web Crawlers

▷ **Definition 565** A web crawler or spider is a computer probram that browses the WWWeb in an automated, orderly fashion for the purpose of information gathering.

▷ Web crawlers are mostly used for data acquisition of web search engines, but can also automate web maintenance jobs (e.g. link checking).

▷ The WWWeb changes: 20% daily, 30% monthly, 50% never

▷ A Web crawler cycles over the following actions

1. reads web page
2. reports it home
3. finds hyperlinks
4. follows them

▷ Note: you can exclude web crawlers from your web site by configuring `robots.txt`.

Even though the image of a crawling insect suggests that, a web crawler is a program that lives on a host and stays there, downloading selected portions of the WWWeb. Actually, the picture we paint above is quite simplified, a web crawler has to be very careful not to download web pages multiple times to make progress. Recall that – seen as directed graphs – hypertexts may very well be cyclic. Additionally, much of the WWWeb content is only generated by web applications on user request, therefore modern web crawlers will try to generate queries that generate pages they can crawl.

The input for a web search engine is a query, i.e. a string that describes the set of documents to be referenced in the answer set. There are various types of query languages for information retrieval; we will only go into the most common ones here

Web Search: Queries

▷ **Definition 566** A web search query is a string that describes a set of document (fragments).

▷ **Example 567** Most web search engines accept multiword queries, i.e. multisets of strings (words).

▷ **Example 568** Many web search engines also accept advanced query operators and wild cards

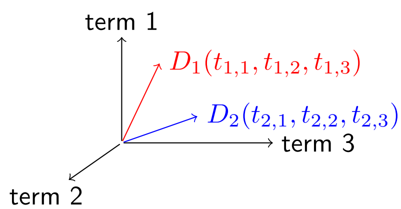
?	(e.g. `science?` means search for the keyword "science" but I am not sure of the spelling)
*	(wildcard, e.g. `comput*` searches for keywords starting with `comput` combined with any word ending)
`AND`	(both terms must be present)
`OR`	(at least one of the terms must be present)

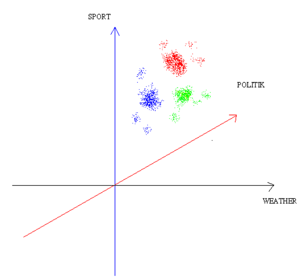
Also: Searches for various information formats & types, e.g. image search, scholarly search (require specialized query languages)

We now come to the central component of a web search engine: the indexing component. The main realization here is that with the size of the current web it is impossible to search the web linearly by comparing the query to the crawled documents one-by-one. So instead, the web search engine extracts salient features from the documents and stores them in a special data structure (usually tree-like) that can be queried instead of the documents themselves. In the prevalent information retrieval algorithms, the salient feature is a word frequency vector.

Searching for Documents Efficiently: Indexing

- ▷ Problem: We cannot search the WWWeb linearly (even with 10^6 computers: $\geq 10^{15}B$)
- ▷ Idea: Write an "index" and search that instead. (like the index in a book)
- ▷ **Definition 569** Search engine indexing analyzes data and stores key/data pairs in a special data structure (the search index to facilitate efficient and accurate information retrieval.
- ▷ Idea: Use the words of a document as index (multiword index) The key for a document is the vector of word frequencies.



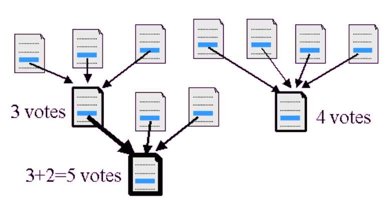


Note: The word frequency vectors used in the "vector space model" for information retrieval are very high-dimensional; the dimension is the number of words in the document corpus. Millions of dimensions are usual. However, linguistic methods like "stemming" (reducing words to word stems) are used to bring down the number of words in practice.

Once an answer set has been determined, the results have to be sorted, so that they can be presented to the user. As the user has a limited attention span – users will look at most at three to eight results before refining a query, it is important to rank the results, so that the hits that contain information relevant to the user's information need early. This is a very difficult problem, as it involves guessing the intentions and information context of users, to which the search engine has no access.

Ranking Search Hits: e.g. Google's Pagerank

- ▷ Problem: There are many hits, need to sort them by some criterion (e.g. importance)
- ▷ Idea: A web site is important, . . . if many other hyperlink to it.



- ▷ Refinement: . . . , if many important web pages hyperlink to it.

▷ **Definition 570** Let A be a web page that is hyperlinked from web pages $S_1, \ldots, S_n$, then

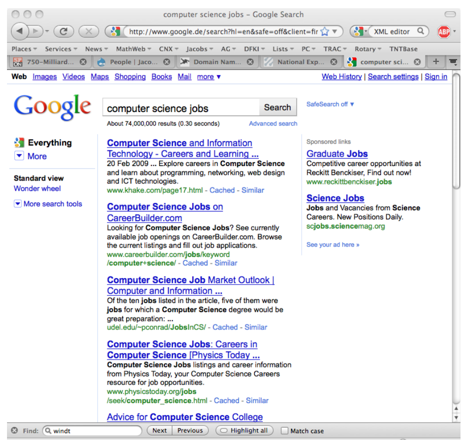
$$\mathrm{PR}(A) = 1 - d + d\left(\frac{\mathrm{PR}(S_1)}{C(S_1)} + \cdots \frac{\mathrm{PR}(S_n)}{C(S_n)}\right)$$

where $C(W)$ is the number of links in a page W and $d = 0.85$.

Getting the ranking right is a determining factor for success of a search engine. In fact, the early of Google was based on the pagerank algorithm discussed above (and the fact that they figured out a revenue stream using text ads to monetize searches).

The final step for a web search engine is answer composition; at least, if the answer is addressed at a human user. The main task here is to assemble those information fragments that the user needs to determine whether the hit described contains information relevant to the respective information need.

Answer Composition in Search Engines



▷ Answers: To present the search results we need to address:

- ▷ Hits and their context
- ▷ format conversion
- ▷ caching

▷ Advertising: to finance the service

- ▷ advertiser can buy search terms
- ▷ ads correspond to search interest
- ▷ advertiser pays by click.

Due to the gigantic size of the Internet, search engines are extremely resource-hungry web applications. The precise figures about the computational resources of the large internet companies are well-kept trade secrets, but the following figure should give an intuition of the scales involved.

How to run Google

- ▷ Google Hardware: estimated 2003
  - ▹ 79,112 Computers (158,224 CPUs)
  - ▹ 316,448 Ghz computation power
  - ▹ 158,224 GB RAM
  - ▹ 6,180 TB Hard disk space
- ▷ 2010 Estimate: $\sim$ 2 MegaCPU
- ▷ Google Software: Custom Linux Distribution

So far we have completely neglected security issues on the Internet. They have become very important since the commercialization of the Internet in 1993 as we on the one hand nowadays use the internet for banking, trade, and confidential communication and on the other hand the Internet has become infested with malware, spyware, e-mail spam, and denial-of-service attacks. Fortunately, many of the underlying problems: authentication, authorization, and confidential communication can be solved by a joint technology: encryption.

3.5.6 Security by Encryption

There are various ways to ensure security on networks: one is just to cease all traffic (not a very attractive one), another is to make the information on the network inaccessible by physical means (e.g. shielding the wires electrically and guarding them by a large police force). Here we want to look into "security by encryption", which makes the content of Internet packets unreadable by unauthorized parties. We will start by reviewing the basics, and work our way towards a secure network infrastructure via the mathematical foundations and special protocols.

Security by Encryption

- ▷ Problem: In open packet-switched networks like the Internet, anyone
  - ▹ can inspect the packets (and see their contents via packet sniffers)
  - ▹ create arbitrary packets (and forge their metadata)

▷ can combine both to falsify communication (man-in-the-middle attack)

In "dedicated line networks" (e.g. old telephone) you needed switch room access.

▷ But there are situations where we want our communication to be confidential,

▷ Internet Banking (obviously, other criminals would like access to your account)

▷ Whistle-blowing (your employer should not know what you sent to WikiLeaks)

▷ Login to Campus.net (wouldn't you like to know my password to "correct" grades?)

▷ Idea: Encrypt packet content (so that only the recipients can decrypt)
an build this into the fabric of the Internet (so that users don't have to know)

▷ **Definition 571** Encryption is the process of transforming information (referred to as plaintext) using an algorithm to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key. The result of encryption is called cyphertext, and the reverse process that transforms cyphertext to plaintext: decryption.

 379 JACOBS UNIVERSITY

g The simplest form of encryption (and the way we know from spy stories) uses uses the same key for encryption and decryption.

Symmetric Key Encryption

▷ **Definition 572** Symmetric-key algorithms are a class of cryptographic algorithms that use essentially identical keys for both decryption and encryption.

▷ **Example 573** Permute the ASCII table by a bijective function $\varphi\colon \{0,\ldots,127\} \to \{0,\ldots,127\}$ (φ is the shared key)

▷ **Example 574** The AES algorithm (Advanced Encryption Standard) [AES01] is a widely used symmetric-key algorithm that is approved by US government organs for transmitting top-secret information.

▷ Note: For trusted communication sender and recipient need access to shared key.

▷ Problem: How to initiate safe communication over the internet? (far, far apart) Need to exchange shared key (chicken and egg problem)

▷ Pipe dream: Wouldn't it be nice if I could just publish a key publicly and use that?

▷ Actually: this works, just (obviously) not with symmetric-key encryption.

 380 JACOBS UNIVERSITY

To get around the chicken-and-egg problem of secure communication we identified above, we will introduce a more general way of encryption: one where we allow the keys for encryption and decryption to be different. This liberalization allows us to enter into a whole new realm of applications.

Public Key Encryption

▷ **Definition 575** In an asymmetric-key encryption method, the key needed to encrypt a message is different from the key for decryption. Such a method is called a public-key encryption if the the decryption key (the private key) is very difficult to reconstruct from encryption key (called the public key).

▷ Preparation: The person who anticipates receiving messages first creates both a public key and an associated private key, and publishes the public key.

▷ Application: Confidential Messaging: To send a confidential message the sender encrypts it using the intended recipient's public key; to decrypt the message, the recipient uses the private key.

▷ Application: Digital Signatures: A message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender had access to the private key (and therefore is likely to be the person associated with the public key used), and the part of the message that has not been tampered with.

 381 JACOBS UNIVERSITY

The confidential messaging is analogous to a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender.

Note: For both applications (confidential messaging and digitally signed documents) we have only stated the basic idea. Technical realizations are more elaborate to be more efficient. One measure for instance is not to encrypt the whole message and compare the result of decrypting it, but only a well-chosen excerpt.

Let us now look at the mathematical foundations of encryption. It is all about the existence of natural-number functions with specific properties. Indeed cryptography has been a big and somewhat unexpected application of mathematical methods from number theory (which was perviously thought to be the ultimate pinnacle of "pure math".)

Encryption by Trapdoor Functions

▷ Idea: Mathematically, encryption can be seen as an injective function. Use functions for which the inverse (decryption) is difficult to compute.

▷ **Definition 576** A one-way function is a function that is "easy" to compute on every input, but "hard" to invert given the image of a random input.

▷ In theory: "easy" and "hard" are understood wrt. computational complexity theory, specifically the theory of polynomial time problems. E.g. "easy" $\hat{=} O(n)$ and "hard" $\hat{=} \Omega(2^n)$

▷ Remark: It is open whether one-way functions exist (≘ to $P = NP$ conjecture)

▷ In practice: "easy" is typically interpreted as "cheap enough for the legitimate users" and "prohibitively expensive for any malicious agents".

▷ **Definition 577** A trapdoor function is a one-way function that is easy to invert given a piece of information called the trapdoor.

▷ **Example 578** Consider a padlock, it is easy to change from "open" to closed, but very difficult to change from "closed" to open unless you have a key (trapdoor).

 382 JACOBS UNIVERSITY

Of course, we need to have one-way or trapdoor functions to get public key encryption to work. Fortunately, there are multiple candidates we can choose from. Which one eventually makes it into the algorithms depends on various details; any of them would work in principle.

Candidates for one-way/trapdoor functions

- ▷ Multiplication and Factoring: The function f takes as inputs two prime numbers p and q in binary notation and returns their product. This function can be computed in $O(n^2)$ time where n is the total length (number of digits) of the inputs. Inverting this function requires finding the factors of a given integer N. The best factoring algorithms known for this problem run in time $2^{O(\log(N)^{\frac{1}{3}}\log(\log(N))^{\frac{2}{3}})}$.
- ▷ Modular squaring and square roots: The function f takes two positive integers x and N, where N is the product of two primes p and q, and outputs x^2 div N. Inverting this function requires computing square roots modulo N; that is, given y and N, find some x such that $x^2 \bmod N = y$. It can be shown that the latter problem is computationally equivalent to factoring N (in the sense of polynomial-time reduction) (used in RSA encryption)
- ▷ Discrete exponential and logarithm: The function f takes a prime number p and an integer x between 0 and $p-1$; and returns the 2^x div p. This discrete exponential function can be easily computed in time $O(n^3)$ where n is the number of bits in p. Inverting this function requires computing the discrete logarithm modulo p; namely, given a prime p and an integer y between 0 and $p-1$, find x such that $2^x = y$.

To see whether these trapdoor function candidates really behave as expected, RSA laboratories, one of the first security companies specializing in public key encryption has established a series of prime factorization challenges to test the assumptions underlying public key cryptography.

Example: RSA-129 problem

- ▷ **Definition 579** We call a number semi-prime, iff it has exactly two prime factors.
- ▷ These are exactly the numbers involved in RSA encryption.
- ▷ RSA laboratories initiated the RSA challenge, to see whether multiplication is indeed a "practical" trapdoor function
- ▷ **Example 580 (The RSA129 Challenge)** is to factor the semi-prime number on the right

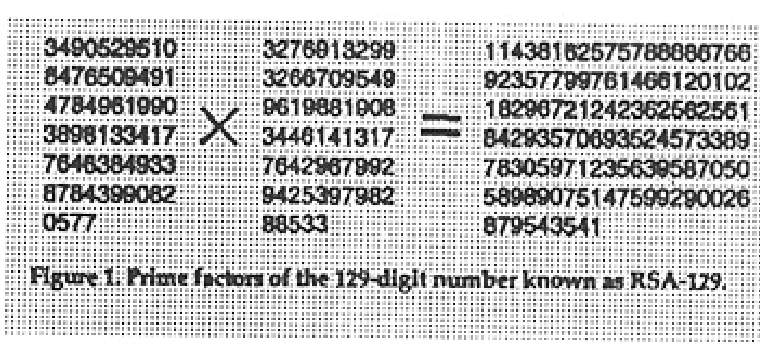


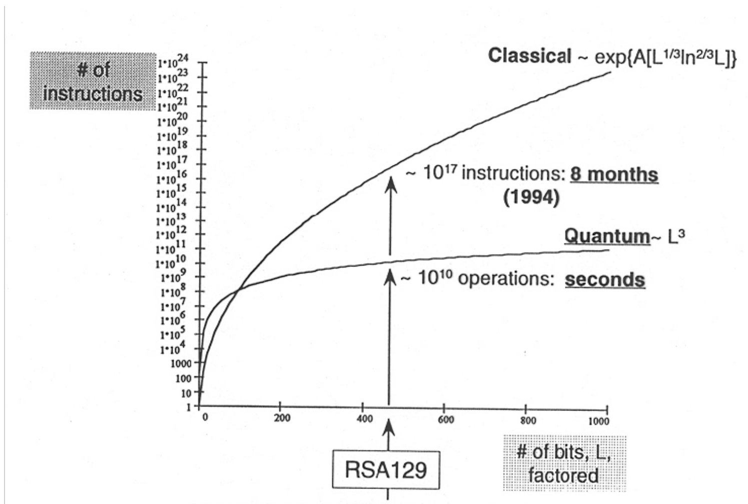
Figure 1. Prime factors of the 129-digit number known as RSA-129.

- ▷ So far, the challenges up to ca 200 decimal digits have been factored, but all within the expected complexity bounds.

▷ but: would you report an algorithm that factors numbers in low complexity?

Note that all of these test are run on conventional hardware (von Neumann architectures); there have been claims that other computing hardware; most notably quantum computing or DNA computing might have completely different complexity theories, which might render these factorization problems tractable. Up to now, nobody has been able to actually build alternative computation hardware that can actually even attempt to solve such factorization problems (or they are not telling).

Classical- and Quantum Computers for RSA-129



This concludes our excursion into theoretical aspects of encryption, we will now turn to the task of building these ideas into existing infrastructure of the Internet and the WWWeb. The most obvious thing we need to do is to publish public keys in a way that it can be verified to whom they belong.

Public Key Certificates

▷ **Definition 581** A public key certificate is an electronic document which uses a digital signature to bind a public key with an identity, e.g. the name of a person or an organization.

▷ Idea: If we trust the signatory's signature, then we can use the certificate to verify that a public key belongs to an individual. Otherwise we verify the signature using the signatory's public key certificate.

▷ Problem: We can ascend the ladder of trust, but in the end we have to trust someone!

▷ In a typical public key infrastructure scheme, the signature will be of a certificate authority, an organization chartered to verify identity and issue public key certificates.

▷ In a "web of trust" scheme, the signature is of either the user (a self-signed certificate) or other users ("endorsements").

▷ on a UNIX system, you can create a certificate (and associated private key) e.g. with

```
openssl ca -in req.pem -out newcert.pem
```


Building on the notion of a public key certificate, we can build secure variants of the application-level protocols. Of course, we could do this individually for every protocol, but this would duplicate efforts. A better way is to leverage the layered infrastructure of the Internet and build a generic secure transport-layer protocol, that can be utilized by all protocols that normally build on TCP or UDP.

Building Security in to the WWWeb Infrastructure

▷ Idea: Build Encryption into the WWWeb infrastructure (make it easy to use)
⇝ Secure variants of the application-level protocols that encrypt contents

▷ **Definition 582** Transport layer security (TLS) is a cryptographic protocol that encrypts the segments of network connections at the transport layer, using asymmetric cryptography for key exchange, symmetric encryption for privacy, and message authentication codes for message integrity.

▷ TLS can be used to make application-level protocols secure.

Let us now look at bit closer into the structure of the TLS handshake, the part of the TLS protocol that initiates encrypted communication.

A TLS Handshake between Client and Server

▷ **Definition 583** A TLS handshake authenticates a server and provides a shared key for symmetric-key encryption. It has the following steps

1. Client presents a list of supported encryption methods
2. Server picks the strongest and tells client (C/S agree on method)
3. Server sends back its public key certificate (name and public key)
4. Client confirms certificate with CA (authenticates Server if successful)
5. Client picks a random number, encrypts that (with servers public key) and sends it to server.
6. Only server can decrypt it (using its private key)
7. Now they both have a shared secret (the random number)
8. From the random number, both parties generate key material

▷ **Definition 584** A TLS connection is a transport-layer connection secured by symmetric-key encryption. Authentication and keys are established by a TLS handshake and the connection is encrypted until it closes.

The reason we switch from public key to symmetric encryption after communication has been initiated and keys have been exchanged is that symmetric encryption is computationally more efficient without being intrinsically less secure.

But there is more to the integration of encryption into the WWWeb, than just enabling secure transport protocols. We need to extend the web servers and web browsers to implement the secure protocols (of course), and we need to set up a system of certification agencies, whose public keys are baked into web servers (so that they can check the signatures on public keys in server certificates). Moreover, we need user interfaces that allow users to inspect certificates, and grant exceptions, if needed.

Building Security in to the WWWeb Infrastructure

- ▷ **Definition 585** HTTP Secure (HTTPS) is a variant of HTTP that uses TLS for transport. HTTPS URIs start with `https://`

- ▷ Server Integration: All common web servers support HTTPS on port 443 (default), but need a public key certificate. (self-sign one or buy one from a CA)
- ▷ Browser Integration: All common web browsers support HTTPS and give access to certificates

 389 JACOBS UNIVERSITY

The next topic is an abstraction of the markup frameworks we have looked at so far. We introduce XML, the eXtensible Markup Language, which is used as a basis of many Internet and WWWeb technologies nowadays.

3.5.7 An Overview over XML Technologies

We have seen that many of the technologies that deal with marked-up documents utilize the tree-like structure of (the DOM) of HTML documents. Indeed, it is possible to abstract from the concrete vocabulary of HTML that the intended layout of hypertexts and the function of its fragments, and build a generic framework for document trees. This is what we will study in this *subsection*.

Excursion: XML (EXtensible Markup Language)

- ▷ XML is language family for the Web
  - ▹ tree representation language (begin/end brackets)
  - ▹ restrict instances by *Doc. Type Def. (DTD)* or *Schema* (Grammar)
  - ▹ Presentation markup by *style files* (XSL: XML Style Language)
- ▷ XML is extensible HTML & simplified SGML
- ▷ logic annotation (*markup*) instead of presentation!
- ▷ many tools available: parsers, compression, data bases, ...

▷ conceptually: transfer of directed graphs instead of strings.

▷ details at http://www.w3c.org

The idea of XML being an “extensible” markup language may be a bit of a misnomer. It is made “extensible” by giving language designers ways of specifying their own vocabularies. As such XML does not have a vocabulary of its own, so we could have also it an “empty” markup language that can be filled with a vocabulary.

XML is Everywhere (E.g. document metadata)

▷ **Example 586** Open a PDF file in AcrobatReader, then cklick on *File* ↘ *DocumentProperties* ↘ *DocumentMetadata* ↘ *ViewSource*, you get the following text: (showing only a small part)

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:iX='http://ns.adobe.com/iX/1.0/'>
  <rdf:Description xmlns:pdf='http://ns.adobe.com/pdf/1.3/'>
    <pdf:CreationDate>2004-09-08T16:14:07Z</pdf:CreationDate>
    <pdf:ModDate>2004-09-08T16:14:07Z</pdf:ModDate>
    <pdf:Producer>Acrobat Distiller 5.0 (Windows)</pdf:Producer>
    <pdf:Author>Herbert Jaeger</pdf:Author>
    <pdf:Creator>Acrobat PDFMaker 5.0 for Word</pdf:Creator>
    <pdf:Title>Exercises for ACS 1, Fall 2003</pdf:Title>
  </rdf:Description>
  ...
  <rdf:Description xmlns:dc='http://purl.org/dc/elements/1.1/'>
    <dc:creator>Herbert Jaeger</dc:creator>
    <dc:title>Exercises for ACS 1, Fall 2003</dc:title>
  </rdf:Description>
</rdf:RDF>
```


This is an excerpt from the document metadata which AcrobatDistiller saves along with each PDF document it creates. It contains various kinds of information about the creator of the document, its title, the software version used in creating it and much more. Document metadata is useful for libraries, bookselling companies, all kind of text databases, book search engines, and generally all institutions or persons or programs that wish to get an overview of some set of books, documents, texts. The important thing about this document metadata text is that it is not written in an arbitrary, PDF-proprietary format. Document metadata only make sense if these metadata are independent of the specific format of the text. The metadata that MSWord saves with each Word document should be in the same format as the metadata that Amazon saves with each of its book records, and again the same that the British library uses, etc.

XML is Everywhere (E.g. Web Pages)

▷ **Example 587** Open web page file in FireFox, then click on *View* ↘ *PageSource*, you get the following text: (showing only a small part and reformatting)

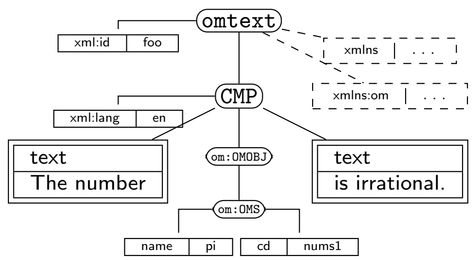
```
<!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Michael Kohlhase</title>
    <meta name="generator"
         content="Page␣generated␣from␣XML␣sources␣with␣the␣WSML␣package"/>
  </head>
  <body>...
    <p>
      <i>Professor of Computer Science</i><br/>
      Jacobs University<br/><br/>
      <strong>Mailing address - Jacobs (except Thursdays)</strong><br/>
```

```
      <a href="http://www.jacobs-university.de/schools/ses">
        School of Engineering &amp; Science
       </a><br/>...
    </p>...
  </body>
</html>
```


XML Documents as Trees

▷ Idea: An XML Document is a Tree

```
<omtext xml:id="foo"
   xmlns="..."
   xmlns:om="...">
  <CMP xml:lang='en'>
   The number
   <om:OMOBJ>
     <om:OMS cd="nums1"
             name="pi"/>
   <om:OMOBJ>
   is irrational.
  </CMP>
</omtext>
```



▷ **Definition 588** The XML document tree is made up of element nodes, attribute nodes, text nodes (and namespace declarations, comments,...)

▷ **Definition 589** For communication this tree is serialized into a balanced bracketing structure, where

- ▷ an element el is represented by the brackets <el> (called the opening tag) and </el> (called the closing tag).
- ▷ The leaves of the tree are represented by empty elements (serialized as <el></el>, which can be abbreviated as <el/>
- ▷ and text nodes (serialized as a sequence of UniCode characters).
- ▷ An element node can be annotated by further information using attribute nodes — serialized as an attribute in its opening tag

Note: As a document is a tree, the XML specification mandates that there must be a unique document root.

The Document Object Model

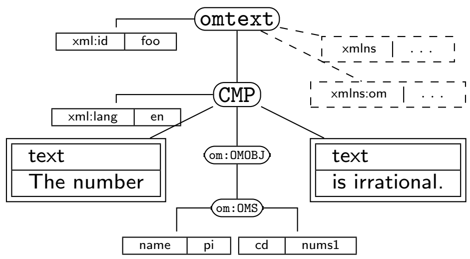
▷ ▷ **Definition 590** The document object model (DOM) is a data structure for storing documents as marked-up documents as document trees together with a standardized set of access methods for manipulating them.

One of the great advantages of viewing marked-up documents as trees is that we can describe subsets of its nodes.

XPath, A Language for talking about XML Tree Fragments

▷ **Definition 591** The XML path language (`XPath`) is a language framework for specifying fragments of XML trees.

▷ **Example 592**



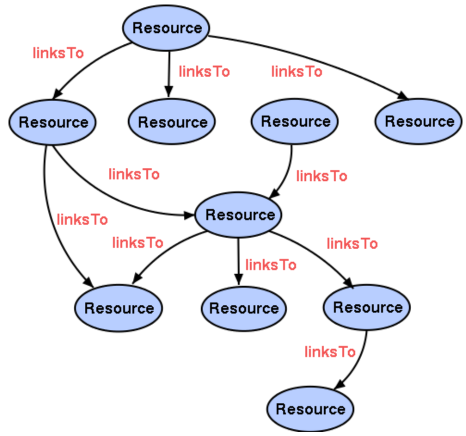
XPath exp.	fragment
/	root
`omtext/CMP/*`	all `CMP` children
//name@	the `name` attribute on the `om:OMS` element
`//CMP/*[1]`	the first child of all `OMS` elements
//*[cd='nums1']@	all elements whose `cd` has value `nums1`

An `XPath` processor is an application or library that reads an XML file into a DOM and given an `XPath` expression returns (pointers to) the set of nodes in the DOM that satisfy the expression.

The final topic for our introduction to the information architecture of the Internet and the WWWeb is an excursion into the “Semantic Web”, a set of technologies that to make the WWWeb more machine-understandable.

3.5.8 The Semantic Web

The Current Web



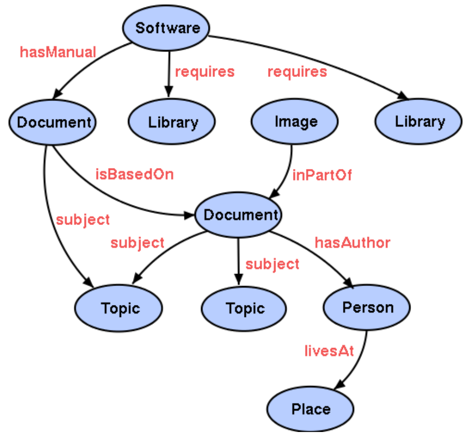
▷ Resources: identified by URI's, untyped

▷ Links: `href`, `src`, . . . limited, non-descriptive

▷ User: Exciting world - semantics of the resource, however, gleaned from content

▷ Machine: Very little information available - significance of the links only evident from the context around the anchor.

The Semantic Web



- ▷ Resources: Globally Identified by URI's or Locally scoped (Blank), Extensible, Relational
- ▷ Links: Identified by URI's, Extensible, Relational
- ▷ User: Even more exciting world, richer user experience
- ▷ Machine: More processable information is available (Data Web)
- ▷ Computers and people: Work, learn and exchange knowledge effectively

What is the Information a User sees?

WWW2002
The eleventh international world wide web conference
Sheraton waikiki hotel
Honolulu, hawaii, USA
7-11 may 2002
1 location 5 days learn interact

Registered participants coming from
australia, canada, chile denmark, france, germany, ghana, hong kong, india, ireland, italy, japan, malta, new zealand, the netherlands, norway, singapore, switzerland, the united kingdom, the united states, vietnam, zaire

On the 7th May Honolulu will provide the backdrop of the eleventh international world wide web conference. This prestigious event ?
Speakers confirmed
Tim Berners-Lee: Tim is the well known inventor of the Web, ?
Ian Foster: Ian is the pioneer of the Grid, the next generation internet ?

What the machine sees

𝒲𝒲𝒲∈′′∈
𝒯⟨]]↕]⊑]\⊔⟨⟩\⊔]∇\⊣⊔⟩≀\⊣↕⊒≀∇↕⌈⊒⟩⊓]⊒]⌊⌋≀\{]∇]\⌋]
𝒮⟨]∇⊣⊔≀\⊒⊣⟩∥⟩∥⟩⟨≀⊔]↕
ℋ≀\⊔↕⊓↕⊓⇔⟨⊣⊒⊣⟩⟩⇔𝒰𝒮𝒜
↖∞∞↕⊣†∈′′∈

ℛ]}⟩∫⊔]∇]⌈√⊣∇⊔⟩⌋⟩√⊣\⊔∫⌋≀↕⟩\}{∇≀↕
⊣⊓∫⊔∇⊣↕⟩⊣⇔⌋⊣\⊣⌈⊣⇔⌋⟨⟩↕]⊓]\↕⊣∇∥⇔{∇⊣\⌋]⇔}]∇↕⊣\†⇔}⟨⊣\⊣⇔⟨≀\}∥≀\}⇔⟩\⌈⟩⊣⇔
⟩∇]↕⊣\⌈⇔⟩⊔⊣↕†⇔⌊⊣√⊣\⇔↕⊣↕⊔⊣⇔\]⊒‡]⊣↕⊣\⌈⇔⊔⟨]\]⊔⟨]∇↕⊣\⌈∫⇔\≀∇⊒⊣†⇔
∫⟩\}⊣√≀∇]⇔∫⊒⟩⊔‡]∇↕⊣\⌈⇔⊔⟨]⊓\⟩⊔]⌈∥⟩\}⌈≀↕⇔⊔⟨]⊓\⟩⊔]⌈∫⊔⊣⊔]∫⇔⊑⟩]⊔\⊣↕⇔‡⊣⟩∇]

𝒪\⊔⟨]⊔⟨ℳ⊣†ℋ≀\⊔↕⊓↕⊓⊒⟩↕↕√∇≀⊑⟩⊓⊔⟨]⌊⊣⌋∥⌈∇≀√≀{⊔⟨]]↕]⊑]\⊔⟨
⟩\⊔]∇\⊣⊔⟩≀\⊣↕⊒≀∇↕⌈⊒⟩⊓]⊒]⌊⌋≀\{]∇]\⌋]↙𝒯⟩∫√∇]∫⊔⟩}}≀⊓∫]⊑]\⊔⊥

𝒮√]⊣∥]∇∫⌋≀\{⟩∇↕]⌈
𝒯⟩↕⌊]∇\]∇∫↖↕]]¬𝒯⟩↕⟩∫⊔⟨]⊒]↕↕∥\≀⊒\⟩\⊑]\⊔≀∇≀{⊔⟨]𝒲]⌊⇔⊥
ℐ⊣\ℱ≀∫⊔]∇¬ℐ⊣\⟩∫⊔⟨]√≀\]]∇≀{⊔⟨]𝒢∇⟩⌈⇔⊔⟨]\]§⊔}]\]∇⊣⊔⟩≀\⟩\⊔]∇\]⊔⊥

Solution: XML markup with "meaningful" Tags

<title>𝒲𝒲𝒲∈′′∈𝒯⟨]]↕]⊑]\⊔⟨⟩\⊔]∇\⊣⊔⟩≀\⊣↕⊒≀∇↕⌈⊒⟩⊓]⊒]⌊⌋≀\{]∇]\⌋]</title>
<place>𝒮⟨]∇⊣⊔≀\𝒲⊣⟩∥⟩∥⟩⟨≀⊔]↕ℋ≀\⊔↕⊓↕⊓⇔⟨⊣⊒⊣⟩⟩⇔𝒰𝒮𝒜</place>
<date>↖∞∞↕⊣†∈′′∈</date>
<participants>ℛ]}⟩∫⊔]∇]⌈√⊣∇⊔⟩⌋⟩√⊣\⊔∫⌋≀↕⟩\}{∇≀↕
⊣⊓∫⊔∇⊣↕⟩⊣⇔⌋⊣\⊣⌈⊣⇔⌋⟨⟩↕]⊓]\↕⊣∇∥⇔{∇⊣\⌋]⇔}]∇↕⊣\†⇔}⟨⊣\⊣⇔⟨≀\}∥≀\}⇔⟩\⌈⟩⊣⇔
⟩∇]↕⊣\⌈⇔⟩⊔⊣↕†⇔⌊⊣√⊣\⇔↕⊣↕⊔⊣⇔\]⊒‡]⊣↕⊣\⌈⇔⊔⟨]\]⊔⟨]∇↕⊣\⌈∫⇔\≀∇⊒⊣†⇔
∫⟩\}⊣√≀∇]⇔∫⊒⟩⊔‡]∇↕⊣\⌈⇔⊔⟨]⊓\⟩⊔]⌈∥⟩\}⌈≀↕⇔⊔⟨]⊓\⟩⊔]⌈∫⊔⊣⊔]∫⇔⊑⟩]⊔\⊣↕⇔
‡⊣⟩∇]</participants>
</introduction>𝒪\⊔⟨]⊔⟨ℳ⊣†ℋ≀\⊔↕⊓↕⊓⊒⟩↕↕√∇≀⊑⟩⊓⊔⟨]⌊⊣⌋∥⌈∇≀√≀{⊔⟨]]↕]⊑]\⊔⟨⟩\⊔]∇\⊣↖
⊔⟩≀\⊣↕⊒≀∇↕⌈⊒⟩⊓]⊒]⌊⌋≀\{]∇]\⌋]↙</introduction>
<program>𝒮√]⊣∥]∇∫⌋≀\{⟩∇↕]⌈
<speaker>𝒯⟩↕⌊]∇\]∇∫↖↕]]¬𝒯⟩↕⟩∫⊔⟨]⊒]↕↕∥\≀⊒\⟩\⊑]\⊔≀∇≀{⊔⟨]𝒲]⌊</speaker>
<speaker>ℐ⊣\ℱ≀∫⊔]∇¬ℐ⊣\⟩∫⊔⟨]√≀\]]∇≀{⊔⟨]𝒢∇⟩⌈⇔⊔⟨]\]§⊔}]\]∇⊣⊔⟩≀\⟩\⊔]∇\]⊔<speaker>
</program>

What the machine sees of the XML

<⊔⟩⊔↕]>𝒲𝒲𝒲∈′′∈𝒯⟨]]↕]⊑]\⊔⟨⟩\⊔]∇\⊣⊔⟩≀\⊣↕⊒≀∇↕⌈⊒⟩⊓]⊒]⌊⌋≀\{]∇]\⌋]</⊔⟩⊔↕]>
<√↕⊣⌋]>𝒮⟨]∇⊣⊔≀\𝒲⊣⟩∥⟩∥⟩⟨≀⊔]↕ℋ≀\⊔↕⊓↕⊓⇔⟨⊣⊒⊣⟩⟩⇔𝒰𝒮𝒜</√↕⊣⌋]>
<⌈⊣⊔]>↖∞∞↕⊣†∈′′∈</⌈⊣⊔]>

©: Michael Kohlhase 401

Need to add "Semantics"

- External agreement on meaning of annotations E.g., Dublin Core
  - Agree on the meaning of a set of annotation tags
  - Problems with this approach: Inflexible, Limited number of things can be expressed
- Use Ontologies to specify meaning of annotations
  - Ontologies provide a vocabulary of terms
  - New terms can be formed by combining existing ones
  - Meaning (semantics) of such terms is formally specified
  - Can also specify relationships between terms in multiple ontologies
- Inference with annotations and ontologies (get out more than you put in!)
  - Standardize annotations in RDF [KC04] or RDFa [BAHS] and ontologies on OWL [w3c09]
  - Harvest RDF and RDFa in to a triplestore or OWL
  -

©: Michael Kohlhase 402

Chapter 4

Search and Declarative Computation

In this chapter, we will take a look at two particular topics in computation.

The first is a class of algorithms that are generally (and thus often as a last resort, where other algorithms are missing) appliccable to a wide class of problems that can be represented in a certain form.

The second applies what we have learnt to a new programming paradigm: after we have looked at functional programming in Section 2.2 and imperative languages in Subsection 3.4.0 and Subsection 3.4.2, we will combine the representation via logic (see ??) and backtracking search into a new programming paradigm: "logic programming", or more abstratcly put "declarative programming".

`ProLog` is a simple logic programming language that exemplifies the ideas we want to discuss quite nicely. We will not introduce the language formally, but in concrete examples as we explain the theortical concepts. For a complete reference, please consult the online book by Blackburn & Bos & Striegnitz `http://www.coli.uni-sb.de/~kris/learn-prolog-now/`.

4.1 Problem Solving and Search

In this section, we will look at a class of algorithms called search algorithms. These are algorithms that help in quite general situations, where there is a precisely described problem, that needs to be solved.

4.1.1 Problem Solving

Before we come to the search algorithms themselves, we need to get a grip on the types of problems themselves and how we can represent them, and on what the variuous types entail for the problem solving process.

The first step is to classify the problem solving process by the amount of knowledge we have available. It makes a difference, whether we know all the factors involved in the problem before we actually are in the situation. In this case, we can solve the problem in the abstract, i.e. make a plan before we actually enter the situation (i.e. offline), and then when the problem arises, only execute the plan. If we do not have complete knowledge, then we can only make partial plans, and have to be in the situation to obtain new knowledge (e.g. by observing the effects of our actions or the actions of others). As this is much more difficult we will restrict ourselves to offline problem solving.

Problem solving

▷ Problem: Find algorithms that help solving problems in general

▷ Idea: If we can describe/represent problems in a standardized way, we may have a chance to find general algorithms.

We will use the following two concepts to describe problems

States A set of possible situations in in our problem domain

Actions A set of possible actions that get us from one state to another.

Using these, we can view a sequence of actions as a solution, if it brings us into a situation, where the problem is solved.

▷ **Definition 593** Offline problem solving: Acting only with complete knowledge of problem and solution

▷ **Definition 594** Online problem solving: Acting without complete knowledge

▷ Here: we are concerned with offline problem solving only.

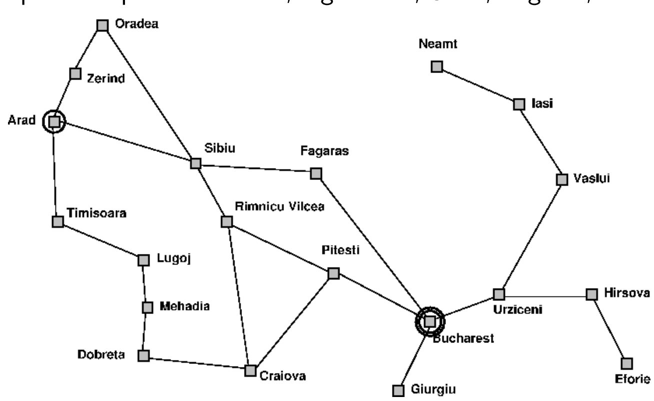
We will use the following problem as a running example. It is simple enough to fit on one slide and complex enough to show the relevant features of the problem solving algorithms we want to talk about.

Example: Traveling in Romania

▷ Scenario: On holiday in Romania; currently in Arad, Flight leaves tomorrow from Bucharest.

▷ Formulate problem: *States*: various cities *Actions*: drive between cities

▷ Solution: Appropriate sequence of cities, e.g.: Arad, Sibiu, Fagaras, Bucharest



Problem Formulation

▷ The problem formulation models the situation at an appropriate level of abstraction. (do not model things like "put on my left sock", etc.)

▷ it describes the initial state (we are in Arad)

▷ it also limits the objectives. (excludes, e.g. to stay another couple of weeks.)

▷ Finding the right level of abstraction and the required (not more!) information is often the key to success.

▷ **Definition 595** A problem (formulation) $\mathcal{P} := \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set $\mathcal{S}$ of states and a set $\mathcal{O}$ of operators that specify how states can be accessed from each other. Certain states in $\mathcal{S}$ are designated as goal states ($\mathcal{G} \subseteq \mathcal{S}$) and there is a unique initial state $\mathcal{I}$.

▷ **Definition 596** A solution for a problem $\mathcal{P}$ consists of a sequence of actions that bring us from $\mathcal{I}$ to a goal state.

 405 JACOBS UNIVERSITY

Problem types

▷ Single-state problem

- ▷ observable (at least the initial state)
- ▷ deterministic (i.e. the successor of each state is determined)
- ▷ static (states do not change other than by our own actions)
- ▷ discrete (a countable number of states)

▷ Multiple-state problem:

- ▷ initial state not/partially observable (multiple initial states?)
- ▷ deterministic, static, discrete

▷ Contingency problem:

- ▷ non-deterministic (solution can branch, depending on contingencies)
- ▷ unknown state space (like a baby, agent has to learn about states and actions)

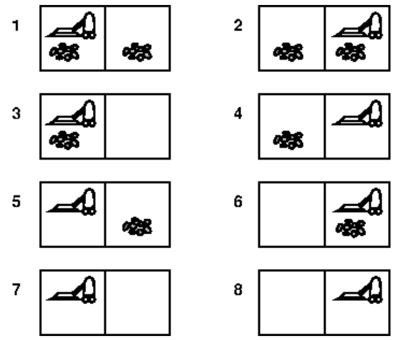
 406 JACOBS UNIVERSITY

We will explain these problem types with another example. The problem $\mathcal{P}$ is very simple: We have a vacuum cleaner and two rooms. The vacuum cleaner is in one room at a time. The floor can be dirty or clean.

The possible states are determined by the position of the vacuum cleaner and the information, whether each room is dirty or not. Obviously, there are eight states: $\mathcal{S} = \{1, 2, 3, 4, 5, 6, 7, 8\}$ for simplicity.

The goal is to have both rooms clean, the vacuum cleaner can be anywhere. So the set $\mathcal{G}$ of goal states is $\{\mathbf{7}, \mathbf{8}\}$. In the single-state version of the problem, $[right, suck]$ shortest solution, but $[suck, right, suck]$ is also one. In the multiple-state version we have $[right(\{2,4,6,8\}), suck(\{4,8\}), left(\{3,7\}), suck(\{7\})]$

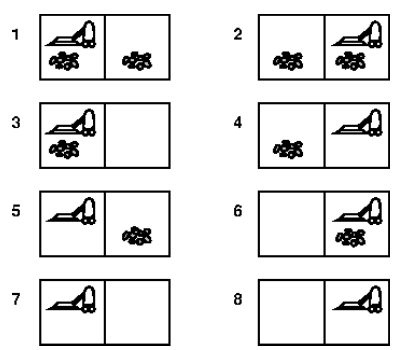
Example: vacuum-cleaner world



- Single-state Problem:
  - Start in 5
  - Solution: $[right, suck]$
- Multiple-state Problem:
  - Start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
  - Solution:
    $[right, suck, left, suck]$
    $right \rightarrow \{2, 4, 6, 8\}$
    $suck \rightarrow \{4, 8\}$
    $left \rightarrow \{3, 7\}$
    $suck \rightarrow \{7\}$

Example: vacuum-cleaner world (continued)

- Contingency Problem:



  - Murphy's Law: $suck$ can dirty a clean carpet
  - Local sensing: $dirty$ / $notdirty$ at location only
  - Start in: $\{1, 3\}$
  - Solution: $[suck, right, suck]$
    $suck \rightarrow \{5, 7\}$
    $right \rightarrow \{6, 8\}$
    $suck \rightarrow \{6, 8\}$

▷ better: $[suck, right, \textbf{if } dirt \textbf{ then } suck]$ (decide whether in 6 or 8 using local sensing)

In the contingency version of $\mathcal{P}$ a solution is the following: $[suck(\{5,7\}), right \rightarrow (\{6,8\}), suck \rightarrow (\{\mathbf{6},8\})]$, $[suck(\{\mathbf{5},7\})]$, etc. Of course, local sensing can help: narrow $\{6,8\}$ to $\{6\}$ or $\{8\}$, if we are in the first, then suck.

Single-state problem formulation

▷ Defined by the following four items

1. Initial state: (e.g. $Arad$)
2. Successor function S: (e.g. $S(Arad) = \{\langle goZer, Zerind\rangle, \langle goSib, Sibiu\rangle, \ldots\}$)
3. Goal test: (e.g. $x = Bucharest$ (explicit test) $noDirt(x)$ (implicit test))
4. Path cost (optional): (e.g. sum of distances, number of operators executed, etc.)

▷ Solution: A sequence of operators leading from the initial state to a goal state

"Path cost": There may be more than one solution and we might want to have the "best" one in a certain sense.

Selecting a state space

▷ Abstraction: Real world is absurdly complex
State space must be abstracted for problem solving

▷ (Abstract) state: Set of real states

▷ (Abstract) operator: Complex combination of real actions

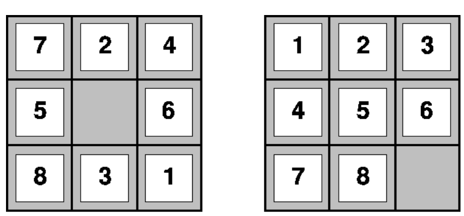
▷ Example: $Arad \rightarrow Zerind$ represents complex set of possible routes

▷ (Abstract) solution: Set of real paths that are solutions in the real world

"State": e.g., we don't care about tourist attractions found in the cities along the way. But this is problem dependent. In a different problem it may well be appropriate to include such information in the notion of state.

"Realizability": one could also say that the abstraction must be sound wrt. reality.

Example: The 8-puzzle

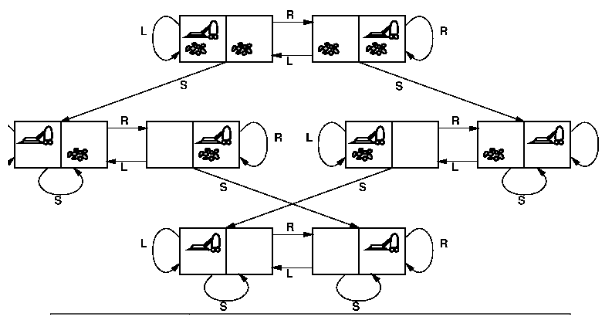


Start State Goal State

States	integer locations of tiles
Actions	$left, right, up, down$
Goal test	= goal state?
Path cost	1 per move

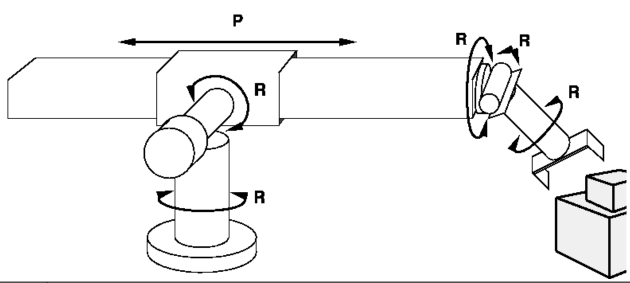
How many states are there? N factorial, so it is not obvious that the problem is in NP. One needs to show, for example, that polynomial length solutions do always exist. Can be done by combinatorial arguments on state space graph (really ?).

Example: Vacuum-cleaner



States	integer dirt and robot locations
Actions	$left, right, suck, noOp$
Goal test	$notdirty?$
Path cost	1 per operation (0 for $noOp$)

Example: Robotic assembly



States	real-valued coordinates of robot joint angles and parts of the object to be assembled
Actions	continuous motions of robot joints
Goal test	assembly complete?
Path cost	time to execute

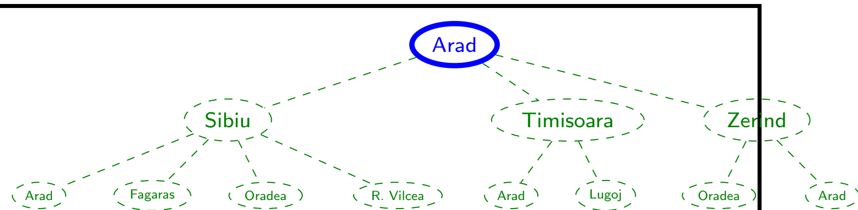
4.1.2 Search

Tree search algorithms

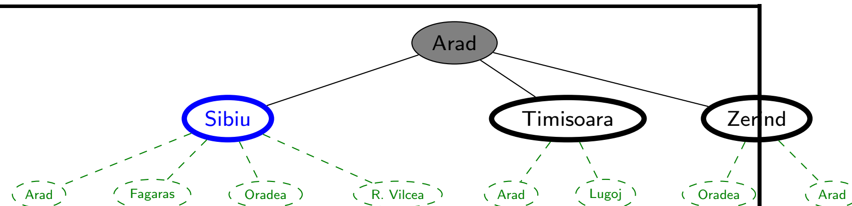
▷ Simulated exploration of state space in a search tree by generating successors of already-explored states (Offline Algorithm)

```
procedure Tree-Search (problem, strategy) : <a solution or failure>
  <initialize the search tree using the initial state of problem>
  loop
     if <there are no candidates for expansion> <return failure> end if
    <choose a leaf node for expansion according to strategy>
     if <the node contains a goal state> return <the corresponding solution>
     else <expand the node and add the resulting nodes to the search tree>
     end if
  end loop
end procedure
```

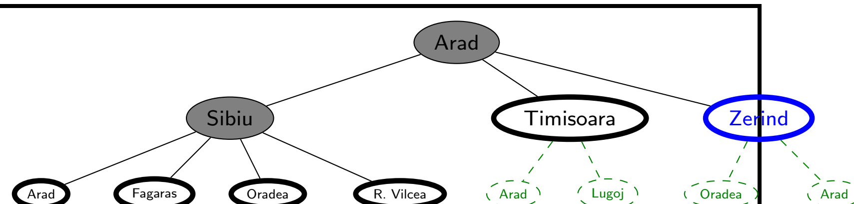

Tree Search: Example



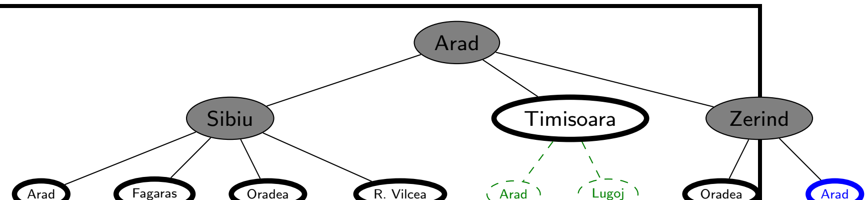
Tree Search: Example



Tree Search: Example

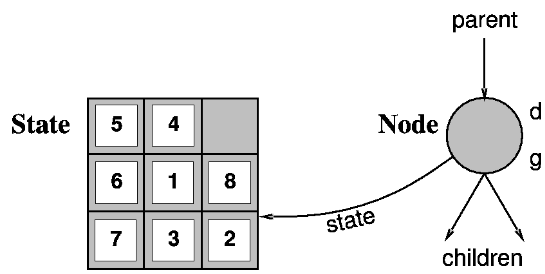


Tree Search: Example



Implementation: States vs. nodes

- ▷ A (representation of) a physical configuration
- ▷ A data structure constituting part of a search tree (includes *parent*, *children*, *depth*, *path cost*, etc.)



Implementation of search algorithms

```
procedure Tree_Search (problem,strategy)
  fringe := insert(make_node(initial_state(problem)))
    loop
     if fringe <is empty> fail end if
      node := first(fringe,strategy)
      if NodeTest(State(node)) return State(node)
      else fringe := insert_all(expand(node,problem),strategy)
      end if
   end loop
end procedure
```

- ▷ **Definition 597** The fringe is a list nodes not yet considered. It is ordered by the search strategy (see below)

STATE gives the state that is represented by *node*

EXPAND = creates new nodes by applying possible actions to *node*

A node is a data structure representing states, will be explained in a moment.

MAKE-QUEUE creates a queue with the given elements.

fringe holds the queue of nodes not yet considered.

REMOVE-FIRST returns first element of queue and as a side effect removes it from *fringe*.

STATE gives the state that is represented by *node*.

EXPAND applies all operators of the problem to the current node and yields a set of new nodes.

INSERT inserts an element into the current *fringe* queue. This can change the behavior of the search.

INSERT-ALL Perform INSERT on set of elements.

Search strategies

- ▷ Strategy: Defines the order of node expansion
- ▷ Important properties of strategies:

completeness	does it always find a solution if one exists?
time complexity	number of nodes generated/expanded
space complexity	maximum number of nodes in memory
optimality	does it always find a least-cost solution?

▷ Time and space complexity measured in terms of:

b	maximum branching factor of the search tree
d	depth of a solution with minimal distance to root
m	maximum depth of the state space (may be ∞)

Complexity means here always *worst-case* complexity.

Note that there can be infinite branches, see the search tree for Romania.

4.1.3 Uninformed Search Strategies

Uninformed search strategies

▷ **Definition 598 (Uninformed search)** Use only the information available in the problem definition

▷ Frequently used strategies:

- ▷ Breadth-first search
- ▷ Uniform-cost search
- ▷ Depth-first search
- ▷ Depth-limited search
- ▷ Iterative deepening search

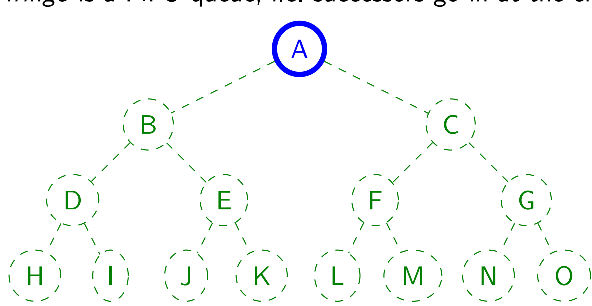
The opposite of uninformed search is informed or *heuristic* search. In the example, one could add, for instance, to prefer cities that lie in the general direction of the goal (here SE).

Uninformed search is important, because many problems do not allow to extract good heuristics.

Breadth-first search

▷ Idea: Expand shallowest unexpanded node

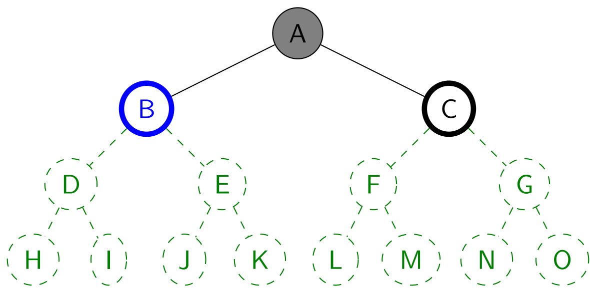
▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

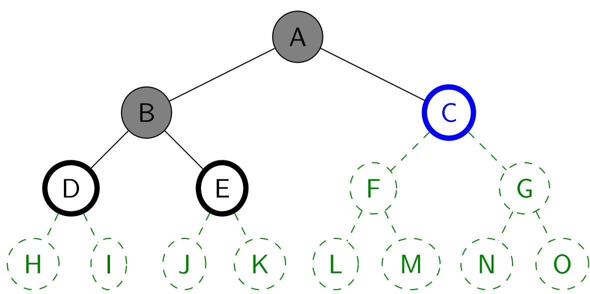
▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

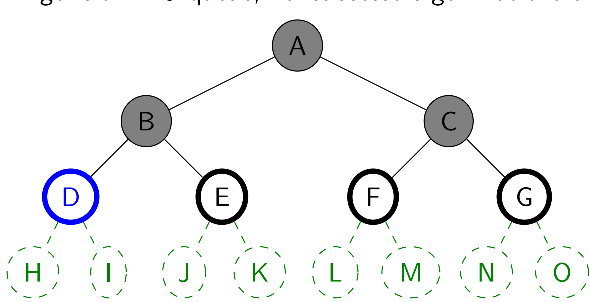
▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

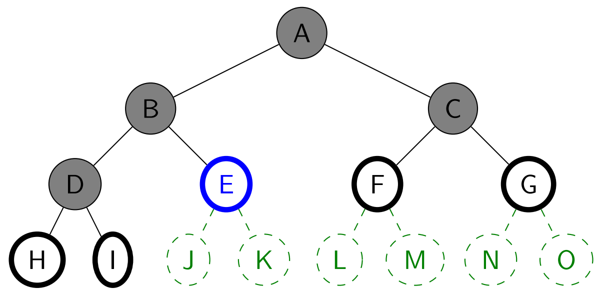
▷ Idea: Expand shallowest unexpanded node

▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



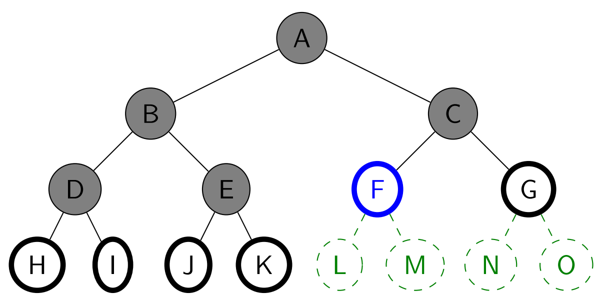
Breadth-First Search

- ▷ Idea: Expand shallowest unexpanded node
- ▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

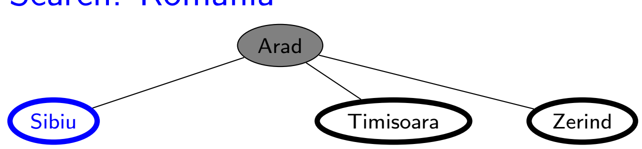
- ▷ Idea: Expand shallowest unexpanded node
- ▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



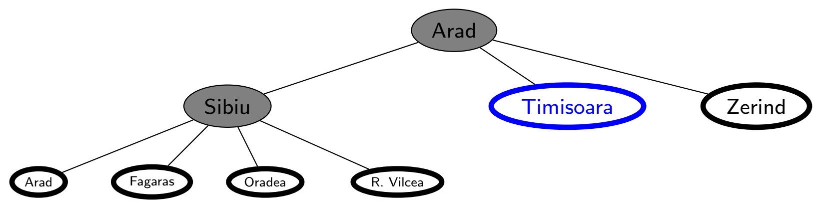
We will now apply the breadth-first search strategy to our running example: Traveling in Romania. Note that we leave out the green dashed nodes that allow us a preview over what the search tree will look like (if expanded). This gives a much

Breadth-First Search: Romania

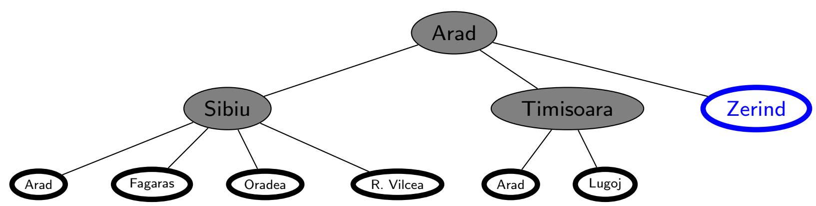
Breadth-First Search: Romania



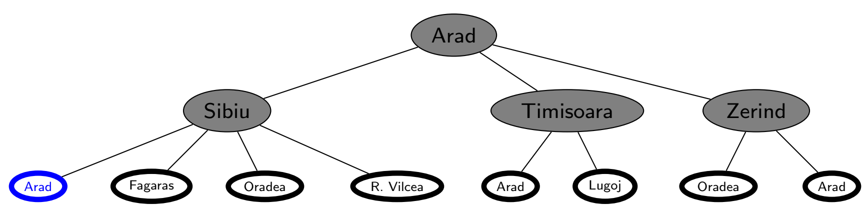
Breadth-First Search: Romania



Breadth-First Search:Romania



Breadth-First Search:Romania



Breadth-first search: Properties

Complete	Yes (if b is finite)
Time	$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1})$ i.e. exponential in d
Space	$O(b^{d+1})$ (keeps every node in memory)
Optimal	Yes (if cost $= 1$ per step), not optimal in general

▷ Disadvantage: Space is the big problem(can easily generate nodes at 5MB/sec so 24hrs = 430GB)

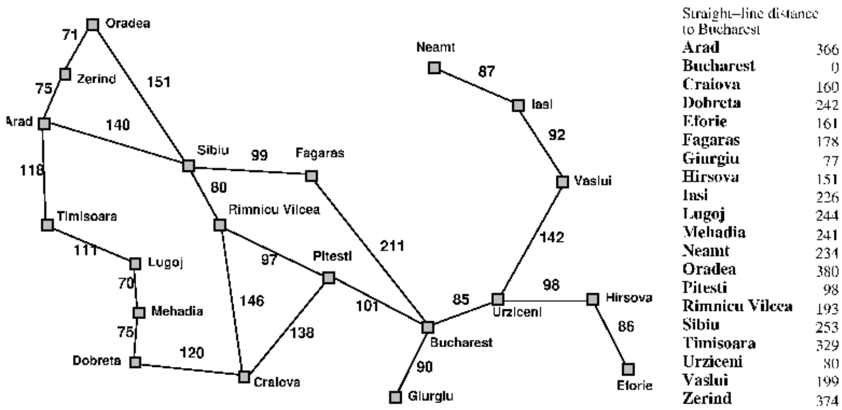
▷ Optimal?: if cost varies for different steps, there might be better solutions below the level of the first solution.

▷ An alternative is to generate *all* solutions and then pick an optimal one. This works only, if m is finite.

The next idea is to let cost drive the search. For this, we will need a non-trivial cost function: we will take the distance between cities, since this is very natural. Alternatives would be the driving time, train ticket cost, or the number of tourist attractions along the way.

Of course we need to update our problem formulation with the necessary information.

Romania with Step Costs as Distances

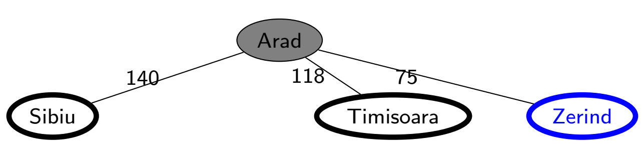


Uniform-cost search

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: `fringe` is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)

Uniform Cost Search: Romania

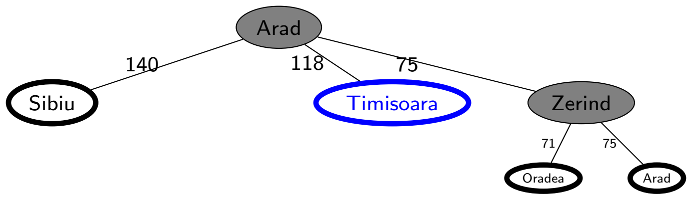
- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: `fringe` is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



Uniform Cost Search: Romania

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: `fringe` is queue ordered by increasing path cost.

▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



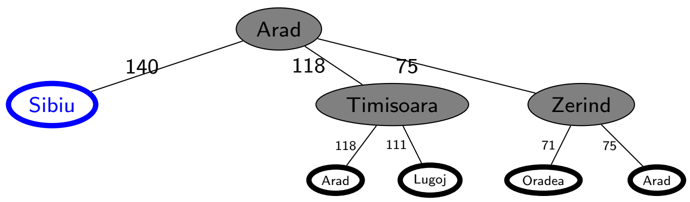
 438 JACOBS UNIVERSITY

Uniform Cost Search: Romania

▷ Idea: Expand least-cost unexpanded node

▷ Implementation: `fringe` is queue ordered by increasing path cost.

▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



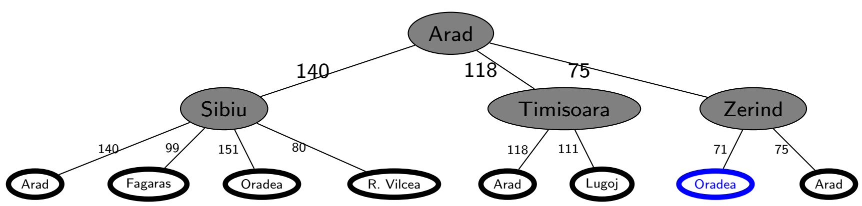
 439 JACOBS UNIVERSITY

Uniform Cost Search: Romania

▷ Idea: Expand least-cost unexpanded node

▷ Implementation: `fringe` is queue ordered by increasing path cost.

▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



 440 JACOBS UNIVERSITY

Note that we must sum the distances to each leaf. That is, we go back to the first level after step 3.

Uniform-cost search: Properties

Complete	Yes (if step costs $\geq \epsilon > 0$)
Time	number of nodes with past-cost less than that of optimal solution
Space	number of nodes with past-cost less than that of optimal solution
Optimal	Yes

If step cost is negative, the same situation as in breadth-first search can occur: later solutions may be cheaper than the current one.

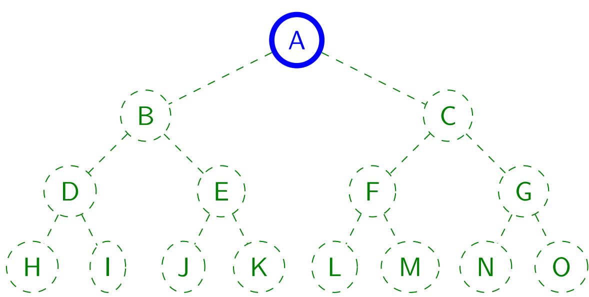
If step cost is 0, one can run into infinite branches. UC search then degenerates into depth-first search, the next kind of search algorithm. Even if we have infinite branches, where the sum of step costs converges, we can get into trouble[18] EdNote:18

Worst case is often worse than BF search, because large trees with small steps tend to be searched first. If step costs are uniform, it degenerates to BF search.

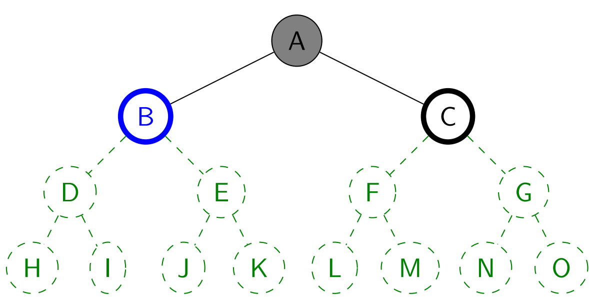
Depth-first search

- Idea: Expand deepest unexpanded node
- **Definition 599 (Implementation)** depth first search is tree search where the *fringe* is organized as a LIFO queue (a stack), i.e. successors go in at front of queue
- Note: Depth-first search can perform infinite cyclic excursions
  Need a finite, non-cyclic search space (or repeated-state checking)

Depth-First Search

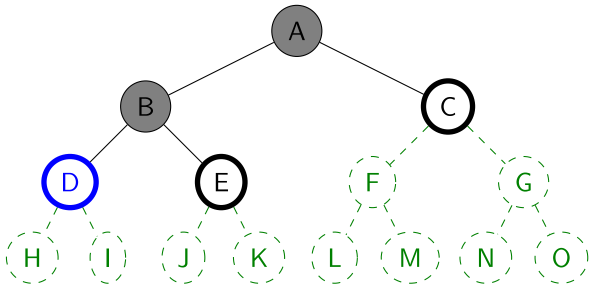


Depth-First Search



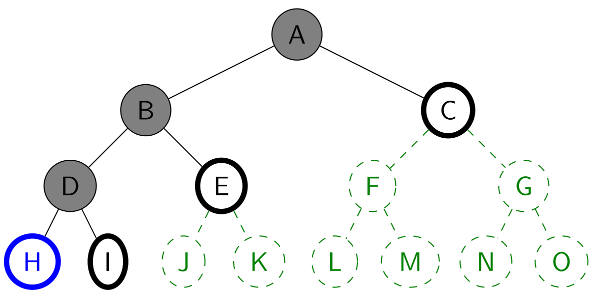
Depth-First Search

[18] EdNote: say how



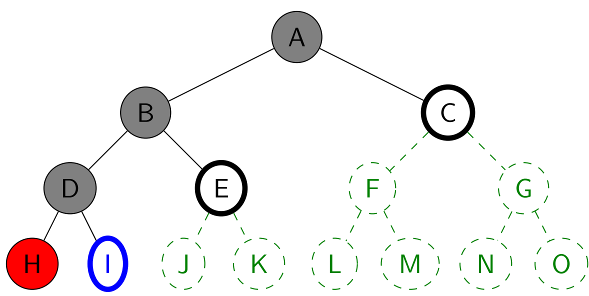
 445

Depth-First Search



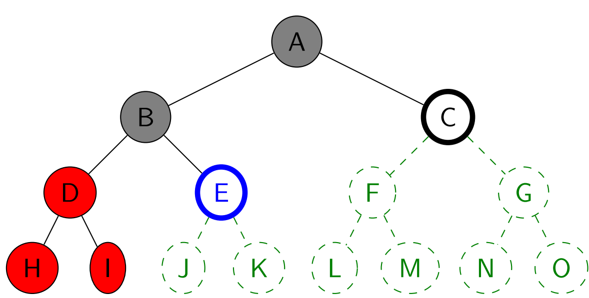
 446

Depth-First Search



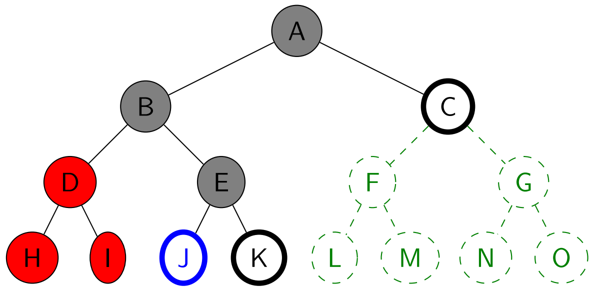
 447

Depth-First Search



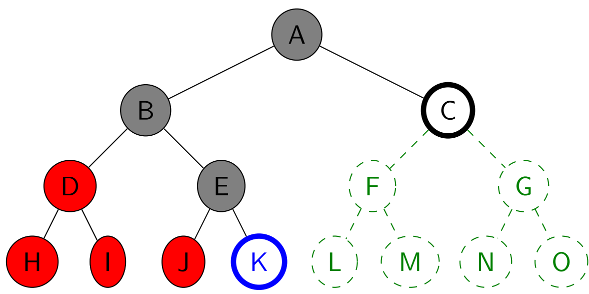
 448

Depth-First Search



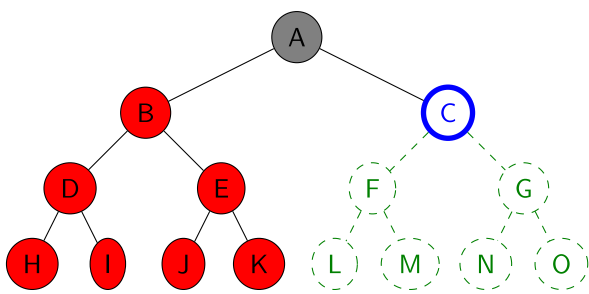
 449

Depth-First Search



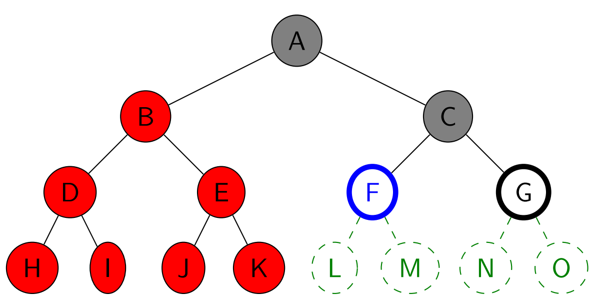
 450

Depth-First Search



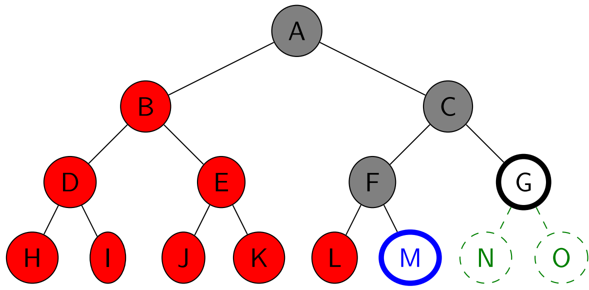
 451

Depth-First Search



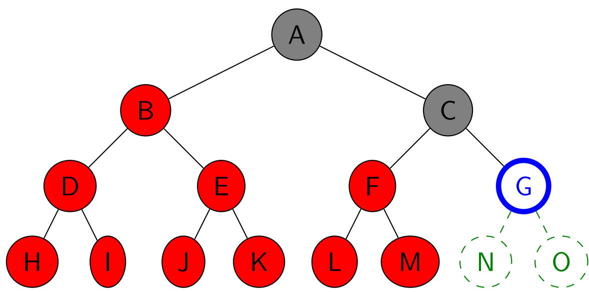
 452

Depth-First Search



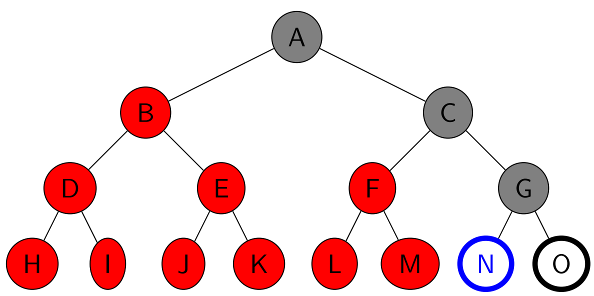
 453

Depth-First Search



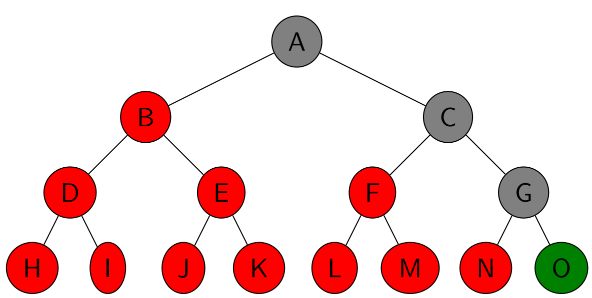
 454

Depth-First Search



 455

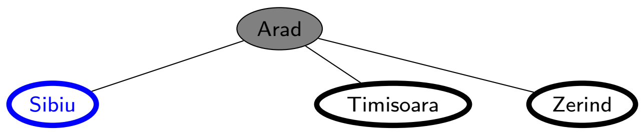
Depth-First Search



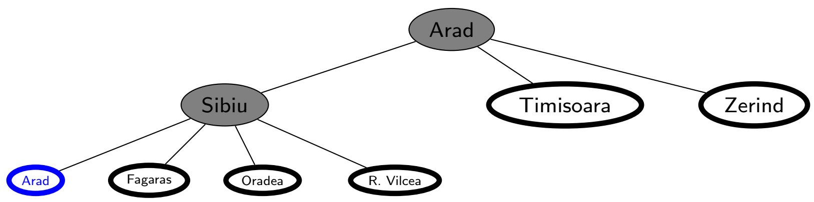
 456

Depth-First Search: Romania

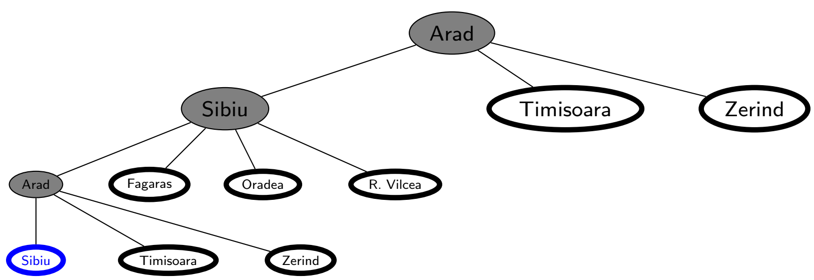
Depth-First Search: Romania



Depth-First Search: Romania



Depth-First Search: Romania



Depth-first search: Properties

Complete	Yes: if state space finite No: if state contains infinite paths or loops
Time	$O(b^m)$ (we need to explore until max depth m in any case!)
Space	$O(b \cdot m)$ (i.e. linear space) (need at most store m levels and at each level at most b nodes)
Optimal	No (there can be many better solutions in the unexplored part of the search tree)

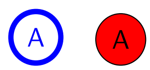
- ▷ Disadvantage: Time terrible if m much larger than d.
- ▷ Advantage: Time may be much less than breadth-first search if solutions are dense.

Iterative deepening search

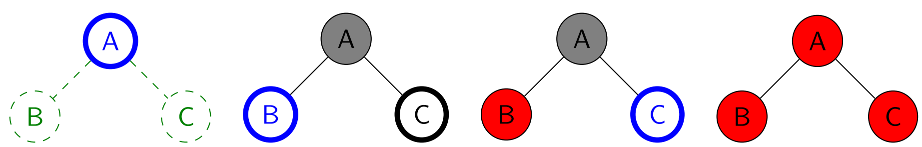
- ▷ Depth-limited search: Depth-first search with depth limit
- ▷ Iterative deepening search: Depth-limit search with ever increasing limits

```
procedure Tree_Search (problem)
   <initialize the search tree using the initial state of problem>
   for depth = 0 to ∞
     result := Depth_Limited_search(problem,depth)
     if depth ≠ cutoff return result end if
   end for
end procedure
```

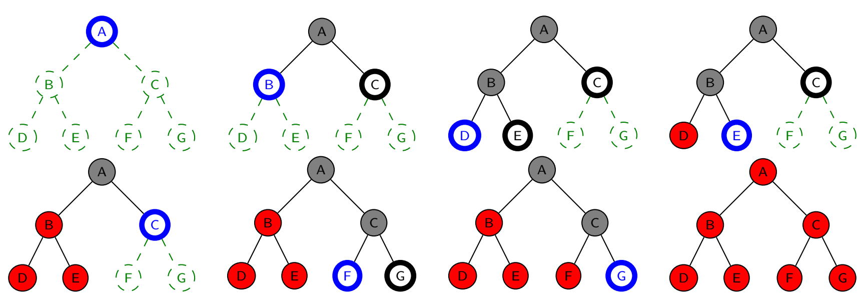

Iterative Deepening Search at Limit Depth 0



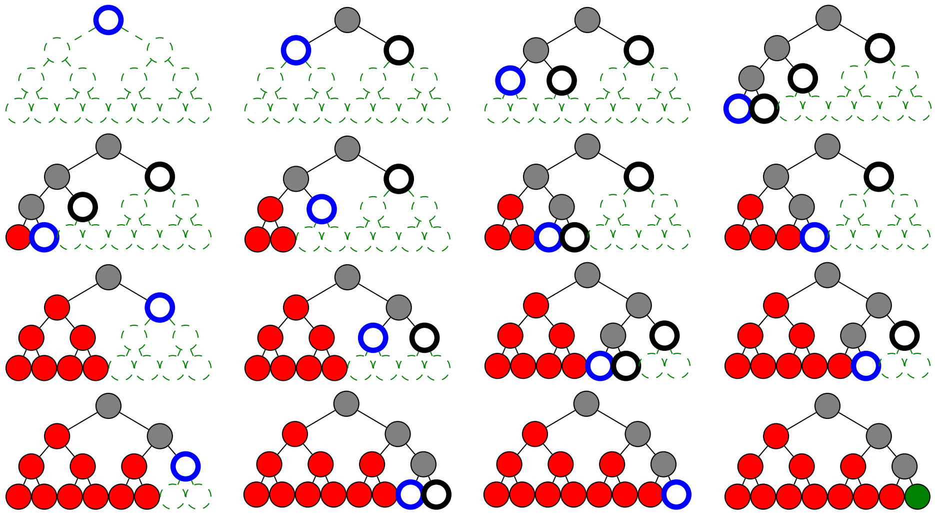
Iterative Deepening Search at Limit Depth 1



Iterative Deepening Search at Limit Depth 2



Iterative Deepening Search at Limit Depth 3



Iterative deepening search: Properties

Complete	Yes
Time	$(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d \in O(b^{d+1})$
Space	$O(bd)$
Optimal	Yes (if step cost = 1)

▷ (Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.

▷ Comparison:

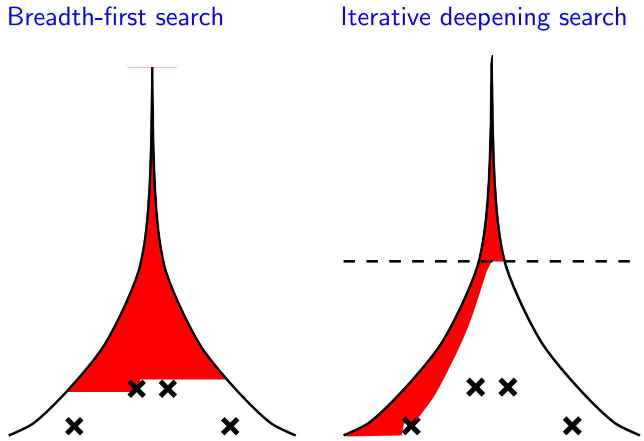
Criterion	Breadth-first	Uniform-cost	Depth-first	Iterative deepening
Complete?	Yes*	Yes*	No	Yes
Time	b^{d+1}	$\approx b^d$	b^m	b^d
Space	b^{d+1}	$\approx b^d$	bm	bd
Optimal?	Yes*	Yes	No	Yes

Note: To find a solution (at depth d) we have to search the whole tree up to d. Of course since we do not save the search state, we have to re-compute the upper part of the tree for the next level. This seems like a great waste of resources at first, however, iterative deepening search tries to be complete without the space penalties.

However, the space complexity is as good as depth-first search, since we are using depth-first search along the way. Like in breadth-first search, the whole tree on level d (of optimal solution) is explored, so optimality is inherited from there. Like breadth-first search, one can modify this to incorporate uniform cost search.

As a consequence, variants of iterative deepening search are the method of choice if we do not have additional information.

Comparison



 468 JACOBS UNIVERSITY

4.1.4 Informed Search Strategies

Summary: Uninformed Search/Informed Search

- ▷ Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- ▷ Variety of uninformed search strategies
- ▷ Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- ▷ Next Step: Introduce additional knowledge about the problem (informed search)
  - ▷ Best-first-, A*-search (guide the search by heuristics)
  - ▷ Iterative improvement algorithms

 469 JACOBS UNIVERSITY

Best-first search

- ▷ Idea: Use an evaluation function for each node (estimate of "desirability") Expand most desirable unexpanded node
- ▷ Implementation: *fringe* is a queue sorted in decreasing order of desirability
- ▷ Special cases: Greedy search, A^* search

 470 JACOBS UNIVERSITY

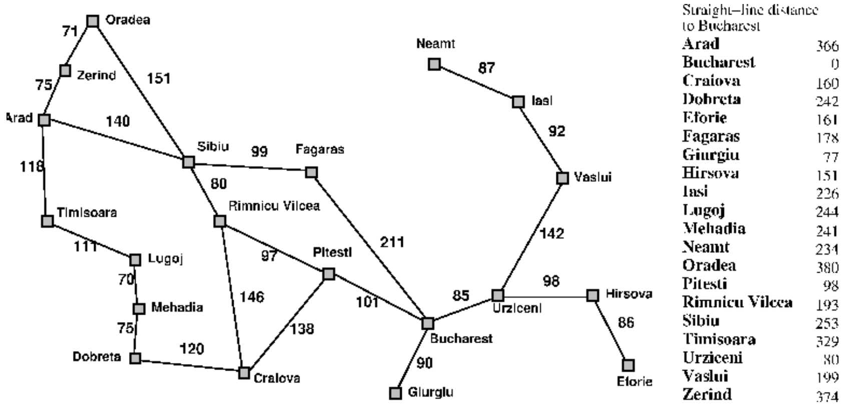
This is like UCS, but with evaluation function related to problem at hand replacing the path cost function.

If the heuristics is arbitrary, we expect incompleteness!

Depends on how we measure "desirability".

Concrete examples follow.

Romania with step costs in km



Greedy search

▷ **Definition 600** A heuristic is an evaluation function h on nodes that estimates of cost from n to the nearest goal state.

Idea: Greedy search expands the node that appears to be closest to goal

▷ **Example 601** $h_{SLD}(n)$ = straight-line distance from n to Bucharest

▷ Note: Unlike uniform-cost search the node evaluation function has nothing to do with the nodes explored so far

internal search control $\rightarrow$ external search control
partial solution cost $\rightarrow$ goal cost estimation

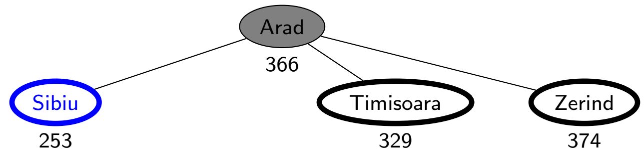
In greedy search we replace the *objective* cost to *construct* the current solution with a heuristic or *subjective* measure from which we think it gives a good idea how far we are from a *solution*. Two things have shifted:

- we went from internal (determined only by features inherent in the search space) to an external/heuristic cost
- instead of measuring the cost to build the current partial solution, we estimate how far we are from the desired goal

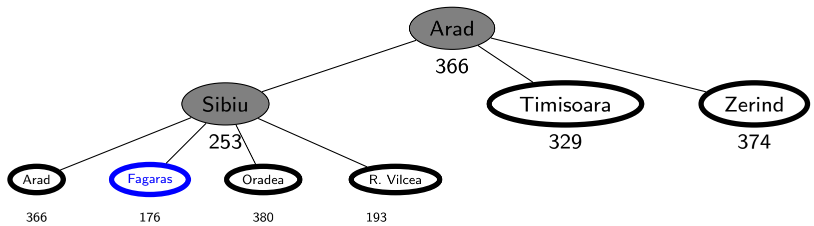
Greedy Search: Romania

Arad
366

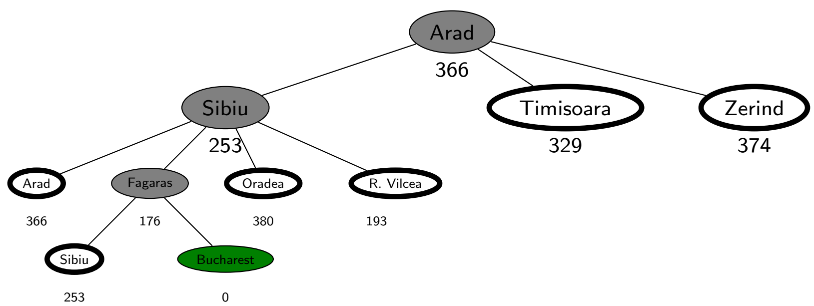
Greedy Search: Romania



Greedy Search: Romania



Greedy Search: Romania



Greedy search: Properties

Complete	No: Can get stuck in loops Complete in finite space with repeated-state checking
Time	$O(b^m)$
Space	$O(b^m)$
Optimal	No

- ▷ **Example 602** Greedy search can get stuck going from Iasi to Oradea: Iasi → Neamt → Iasi → Neamt → $\cdots$
- ▷ Worst-case time same as depth-first search,
- ▷ Worst-case space same as breadth-first
- ▷ But a good heuristic can give dramatic improvement

Greedy Search is similar to UCS. Unlike the latter, the node evaluation function has nothing to do with the nodes explored so far. This can prevent nodes from being enumerated systematically as they are in UCS and BFS.

For completeness, we need repeated state checking as the example shows. This enforces complete enumeration of state space (provided that it is finite), and thus gives us completeness.

Note that nothing prevents from *all* nodes nodes being searched in worst case; e.g. if the heuristic function gives us the same (low) estimate on all nodes except where the heuristic mis-estimates the distance to be high. So in the worst case, greedy search is even worse than BFS, where d (depth of first solution) replaces m.

The search procedure cannot be optional, since actual cost of solution is not considered.

For both, completeness and optimality, therefore, it is necessary to take the actual cost of partial solutions, i.e. the path cost, into account. This way, paths that are known to be expensive are avoided.

A^* search

- ▷ Idea: Avoid expanding paths that are already expensive (make use of actual cost)

  The simplest way to combine heuristic and path cost is to simply add them.

- ▷ **Definition 603** The evaluation function for A^*-search is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost for n and $h(n)$ is the estimated cost to goal from n.
- ▷ Thus $f(n)$ is the estimated total cost of path through n to goal
- ▷ **Definition 604** Best-First-Search with evaluation function $g + h$ is called *astarSearch* search.

This works, provided that h does not overestimate the true cost to achieve the goal. In other words, h must be *optimistic* wrt. the real cost h^*. If we are too pessimistic, then non-optimal solutions have a chance.

A^* search: Admissibility

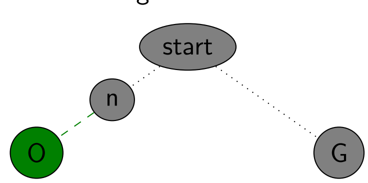
- ▷ **Definition 605 (Admissibility of heuristic)** $h(n)$ is called admissible if $(0 \leq h(n) \leq h^*(n))$ for all nodes n, where $h^*(n)$ is the true cost from n to goal. (In particular: $h(G) = 0$ for goal G)
- ▷ **Example 606** Straight-line distance never overestimates the actual road distance (triangle inequality)

  Thus $h_{SLD}(n)$ is admissible.

A^* Search: Admissibility

- ▷ **Theorem 607** *A^* search with admissible heuristic is optimal*
- ▷ Proof: We show that sub-optimal nodes are never selected by A^*

  P.1 Suppose a suboptimal goal G has been generated then we are in the following situation:

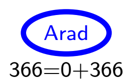


P.2 Let n be an unexpanded node on a path to an optimal goal O, then

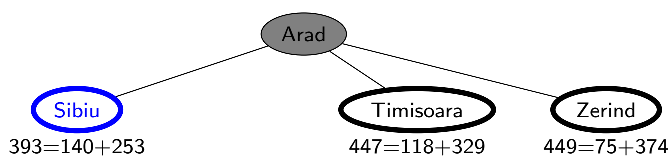
$f(G) = g(G)$	since $h(G) = 0$
$g(G) > g(O)$	since G suboptimal
$g(O) = g(n) + h^*(n)$	n on optimal path
$g(n) + h^*(n) \geq g(n) + h(n)$	since h is admissible
$g(n) + h(n) = f(n)$	

P.3 Thus, $f(G) > f(n)$ and $astarSearch$ never selects G for expansion. □

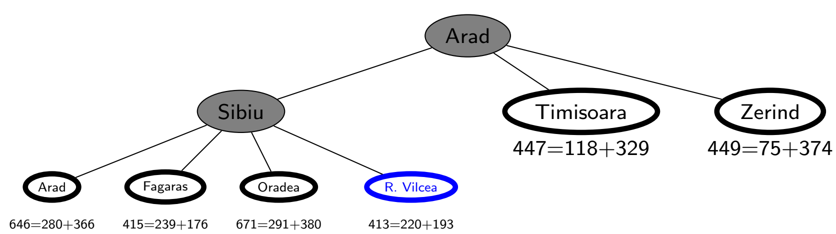
A^* Search Example



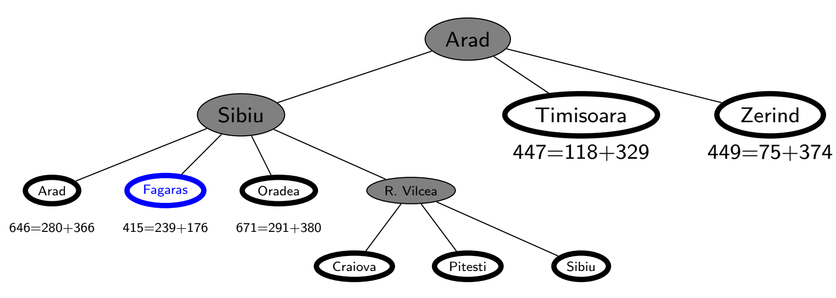
A^* Search Example



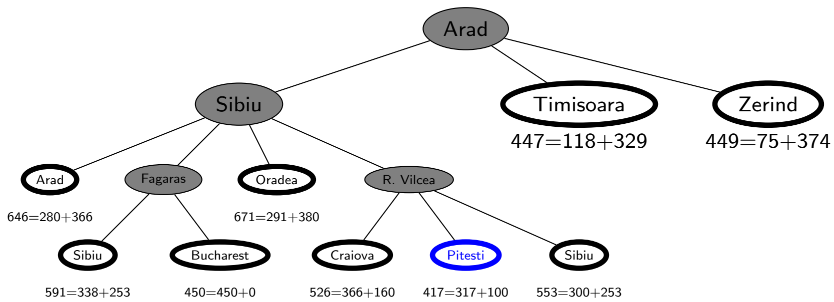
A^* Search Example



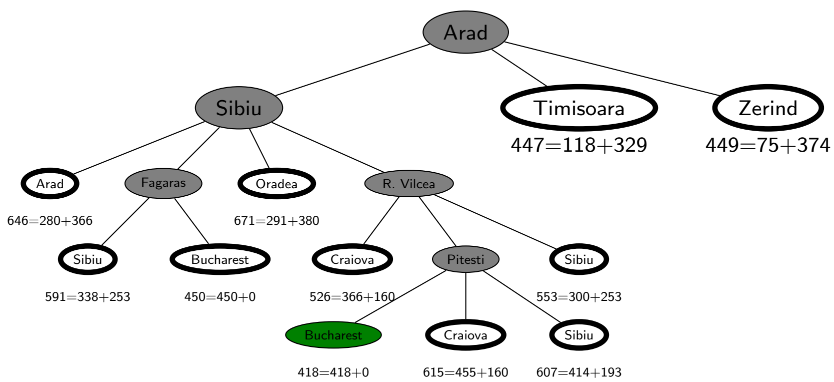
A^* Search Example



A^* Search Example

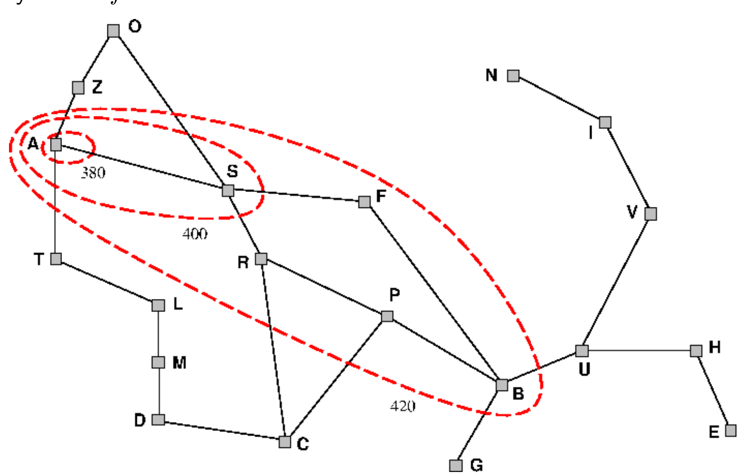


A^* Search Example



A^* search: f-contours

- ▷ A^* gradually adds "f-contours" of nodes



A^* search: Properties

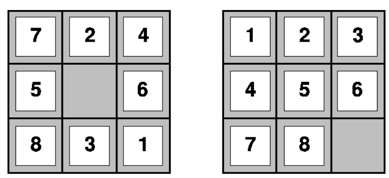
Complete	Yes (unless there are infinitely many nodes n with $f(n) \leq f(0)$)
Time	Exponential in [relative error in h $\times$ length of solution]
Space	Same as time (variant of BFS)
Optimal	Yes

▷ A^* expands all (some/no) nodes with $f(n) < h^*(n)$

▷ The run-time depends on how good we approximated the real cost h^* with h.

Since the availability of admissible heuristics is so important for informed search (particularly for A^*), let us see how such heuristics can be obtained in practice. We will look at an example, and then derive a general procedure from that.

Admissible heuristics: Example 8-puzzle



Start State Goal State

▷ **Example 608** Let $h_1(n)$ be the number of misplaced tiles in node n ($h_1(S) = 6$)

▷ **Example 609** Let $h_2(n)$ be the total manhattan distance from desired location of each tile. ($h_2(S) = 2+0+3+1+0+1+3+4 = 14$)

▷ **Observation 610 (Typical search costs)** (*IDS $\hat{=}$ iterative deepening search*)

nodes explored	*IDS*	$A^*(h_1)$	$A^*(h_2)$
$d = 14$	*3,473,941*	*539*	*113*
$d = 24$	*too many*	*39,135*	*1,641*

Dominance

▷ **Definition 611** Let h_1 and h_2 be two admissible heuristics we say that h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ or all n.

▷ **Theorem 612** *If h_2 dominates h_1, then h_2 is better for search than h_1.*

Relaxed problems

▷ Finding good admissible heuristics is an art!

▷ Idea: Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem.

▷ **Example 613** If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then we get heuristic h_1.

▷ **Example 614** If the rules are relaxed so that a tile can move to *any adjacent square*, then we get heuristic h_2.

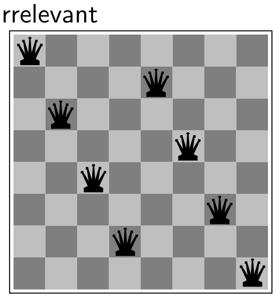
▷ Key point: The optimal solution cost of a relaxed problem is not greater than the optimal solution cost of the real problem

Relaxation means to remove some of the constraints or requirements of the original problem, so that a solution becomes easy to find. Then the cost of this easy solution can be used as an optimistic approximation of the problem.

4.1.5 Local Search

Local Search Problems

▷ Idea: Sometimes the path to the solution is irrelevant



▷ **Example 615 (8 Queens Problem)** Place 8 queens on a chess board, so that no two queens threaten each other.

▷ This problem has various solutions, e.g. the one on the right

▷ **Definition 616** A local search algorithm is a search algorithm that operates on a single state, the current state (rather than multiple paths). (advantage: constant space)

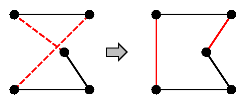
▷ Typically local search algorithms only move to successors of the current state, and do not retain search paths.

▷ Applications include: integrated circuit design, factory-floor layout, job-shop scheduling, portfolio management, fleet deployment,. . .

Local Search: Iterative improvement algorithms

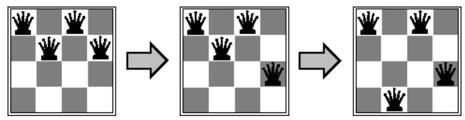
▷ **Definition 617 (Traveling Salesman Problem)** Find shortest trip through set of cities such that each city is visited exactly once.

▷ Idea: Start with any complete tour, perform pairwise exchanges



▷ **Definition 618 (n-queens problem)** Put n queens on $n \times n$ board such that no two queens in the same row, columns, or diagonal.

▷ Idea: Move a queen to reduce number of conflicts



Hill-climbing (gradient ascent/descent)

▷ Idea: Start anywhere and go in the direction of the steepest ascent.

▷ Depth-first search with heuristic and w/o memory

```
procedure Hill-Climbing (problem) (* a state that is a local minimum *)
  local current, neighbor (* nodes *)
  current := Make-Node(Initial-State[problem])
  loop
    neighbor := <a highest-valued successor of current>
    if Value[neighbor] < Value[current]
      return [current]
      current := neighbor
    end if
  end loop
end procedure
```

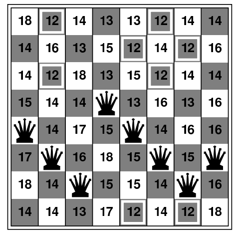
▷ Like starting anywhere in search tree and making a heuristically guided DFS.

▷ Works, if solutions are dense and local maxima can be escaped.

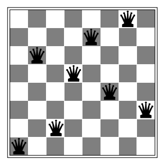
In order to understand the procedure on a more intuitive level, let us consider the following scenario: We are in a dark landscape (or we are blind), and we want to find the highest hill. The search procedure above tells us to start our search anywhere, and for every step first feel around, and then take a step into the direction with the steepest ascent. If we reach a place, where the next step would take us down, we are finished.

Of course, this will only get us into local maxima, and has no guarantee of getting us into global ones (remember, we are blind). The solution to this problem is to re-start the search at random (we do not have any information) places, and hope that one of the random jumps will get us to a slope that leads to a global maximum.

Example Hill-Climbing with 8 Queens

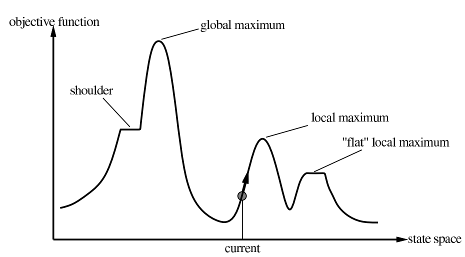


- ▷ Idea: Heuristic function h is number of queens that threaten each other.
- ▷ **Example 619** An 8-queens state with heuristic cost estimate $h = 17$ showing h-values for moving a queen within its column



- ▷ Problem: The state space has local minima. e.g. the board on the right has $h = 1$ but every successor has $h > 1$.

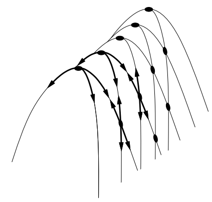
Hill-climbing



- ▷ Problem: Depending on initial state, can get stuck on local maxima/minima and plateaux
- ▷ "Hill-climbing search is like climbing Everest in thick fog with amnesia"
- ▷ Idea: Escape local maxima by allowing some "bad" or random moves.
- ▷ **Example 620** local search, simulated annealing...
- ▷ Properties: All are incomplete, non-optimal.
- ▷ Sometimes performs well in practice (if (optimal) solutions are dense)

Recent work on hill-climbing algorithms tries to combine complete search with randomization to escape certain odd phenomena occurring in statistical distribution of solutions.

Simulated annealing (Idea)



- **Definition 621** Ridges are ascending successions of local maxima
- Problem: They are extremely difficult to navigate for local search algorithms
- Idea: Escape local maxima by allowing some "bad" moves, but gradually decrease their size and frequency

- Annealing is the process of heating steel and let it cool gradually to give it time to grow an optimal cristal structure.
- Simulated Annealing is like shaking a ping-pong ball occasionally on a bumpy surface to free it. (so it does not get stuck)
- Devised by Metropolis et al., 1953, for physical process modelling
- Widely used in VLSI layout, airline scheduling, etc.

Simulated annealing (Implementation)

```
procedure Simulated-Annealing (problem,schedule) (* a solution state *)
  local node, next (* nodes*)
   local T (*a ‘‘temperature’’ controlling prob.~of downward steps *)
   current := Make-Node(Initial-State[problem])
   for t :=1 to ∞
    T := schedule[t]
      if T = 0 return current end if
      next := <a randomly selected successor of current>
     Δ(E) := Value[next]-Value[current]
      if Δ(E) > 0 current := next
      else
       current := next <only with probability> e^(Δ(E)/T)
     end if
  end for
end procedure
```

a problem schedule is a mapping from time to "temperature"

Properties of simulated annealing

▷ At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state x^* because $\frac{e^{\frac{E(x^*)}{kT}}}{e^{\frac{E(x)}{kT}}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T.

▷ Is this necessarily an interesting guarantee?

Local beam search

▷ Idea: Keep k states instead of 1; choose top k of all their successors

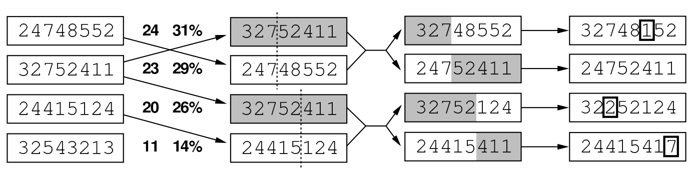
▷ Not the same as k searches run in parallel!(Searches that find good states recruit other searches to join them)

▷ Problem: quite often, all k states end up on same local hill

▷ Idea: Choose k successors randomly, biased towards good ones. (Observe the close analogy to natural selection!)

Genetic algorithms (very briefly)

▷ Idea: Use local beam search (keep a population of k) randomly modify population (mutation) generate successors from pairs of states (sexual reproduction) optimize a fitness function (survival of the fittest)



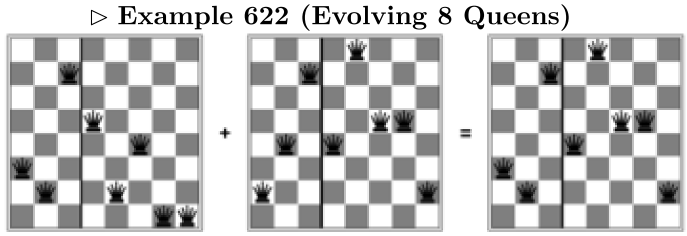
▷ (a) Initial Population (b) Fitness Function (c) Selection (d) Cross–Over (e) Mutation

Genetic algorithms (continued)

▷ Problem: Genetic Algorithms require states encoded as strings (GPs use programs)

▷ Crossover helps iff substrings are meaningful components

▷ **Example 622 (Evolving 8 Queens)**



▷ GAs $\neq$ evolution: e.g., real genes encode replication machinery!

4.2 Logic Programming

We will now learn a new programming paradigm: "logic programming" (also called "Declarative Programming"), which is an application of the search techniques we looked at last, and the logic techniques. We are going to study `ProLog` (the oldest and most widely used) as a concrete example of the ideas behind logic programming.

4.2.1 Introduction to Logic Programming and PROLOG

Logic Programming is a programming style that differs from functional and imperative programming in the basic procedural intuition. Instead of transforming the state of the memory by issuing instructions (as in imperative programming), or computing the value of a function on some arguments, logic programming interprets the program as a body of knowledge about the respective situation, which can be queried for consequences. This is actually a very natural intuition; after all we only run (imperative or functional) programs if we want some question answered.

Logic Programming

▷ Idea: Use logic as a programming language!

▷ We state what we know about a problem (the program) and then ask for results (what the program would compute)

▷ **Example 623**

Program	Leibniz is human Sokrates is is human Sokrates is a greek Every human is fallible	$x + 0 = x$ If $x + y = z$ then $x + s(y) = s(z)$ 3 is prime
Query	Are there fallible greeks?	is there a z with $s(s(0)) + s(0) = z$
Answer	Yes, Sokrates!	yes $s(s(s(0)))$

▷ How to achieve this?: Restrict the logic calculus sufficiently that it can be used as computational procedure.

▷ Slogan: Computation = Logic + Control ([Kowalski '73])

▷ We will use the programming language `ProLog` as an example

Of course, this the whole point of writing down a knowledge base (a program with knowledge about the situation), if we do not have to write down *all* the knowledge, but a (small) subset, from which the rest follows. We have already seen how this can be done: with logic. For logic programming we will use a logic called "first-order logic" which we will not formally introduce here. We have already seen that we can formulate propositional logic using terms from an abstract data type instead of propositional variables – recall Definition 2.6.1. For our purposes, we will just use terms with variables instead of the ground terms used there.

Representing a Knowledge base in `ProLog`

▷ **Definition 624** Fix an abstract data type $\langle\{\mathbb{B}, \ldots\}, \{, \ldots\}\rangle$, then we call all constructor terms of sort $\mathbb{B}$ `ProLog` terms.

▷ A knowledge base is represented (symbolically) by a set of facts and rules.

▷ **Definition 625** A fact is a statement written as a term that is unconditionally true of the domain of interest. (write with a term followed by a ".")

▷ **Example 626** We can state that Mia is a woman as `woman(mia).`

▷ **Definition 627** A rule states information that is *conditionally* true in the domain.

▷ **Definition 628** Facts and rules are collectively called clauses, a `ProLog` program consists of a list of clauses.

▷ **Example 629** Write "something is a car if it has a motor and four wheels" as
`car(X) :- has_motor(X),has_wheels(X,4).` (variables are upper-case)
this is just an ASCII notation for $m(x) \wedge w(x,4) \Rightarrow car(x)$

▷ **Definition 630** The knowledge base given by `ProLog` program is that set of facts that can be derived from it by Modus Ponens (MP), $\wedge I$ and instantiation.

$$\frac{A \quad A \Rightarrow B}{B}\text{MP} \qquad \frac{A \quad B}{A \wedge B}\wedge I \qquad \frac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\text{Subst}$$

To gain an intuition for this quite abstract definition let us consider a concrete knowledge base about cars. Instead of writing down everything we know about cars, we only write down that cars are motor vehicles with four wheels and that a particular object c has a motor and four wheels. We can see that the fact that c is a car can be derived from this. Given our definition of a knowledge base as the deductive closure of the facts and rules explicitly written down, the assertion that c is a car is in the "induced knowledge base", which is what we are after.

Knowledge Base (Example)

▷ **Example 631** `car(c).` is in the knowlege base generated by

```
has_motor(c).
has_wheels(c,4).
car(X):- has_motor(X),has_wheels(X,4).
```

$$\frac{\dfrac{m(c) \quad w(c,4)}{m(c) \wedge w(c,4)}\wedge I \qquad \dfrac{m(x) \wedge w(x,4) \Rightarrow car(x)}{m(c) \wedge w(c,4) \Rightarrow car(c)}\text{Subst}}{car(c)}\text{MP}$$

In this very simple example $car(c)$ is about the only fact we can derive, but in general, knowledge bases can be infinite (we will see examples below)

As knowledge bases can be infinite, we cannot pre-compute them. Instead, logic programming languages compute fragments of the knowledge base by need; i.e. whenever a user wants to check membership; we call this approach querying: the user enters a query term and the system answers `yes` or `no`. This answer is computed in a depth-first search process.

Querying the Knowledge base

▷ Idea: We want to see whether a term is in the knowledge base.

▷ **Definition 632** A query is a list of ProLog terms called goals. Write as `?- A_1, ..., A_n.`, if A_i are terms.

▷ Problem: Knowledge bases can be big and even infinite.

▷ **Example 633** The the knowledge base induced by the program

```
nat(zero).
nat(s(X)) :- nat(X).
```

is the set $\{nat(zero), nat(s(zero)), nat(s(s(zero))), \ldots\}$.

▷ Idea: interpret this as a search problem.

- ▷ state = tuple of goals; goal state = empty list (of goals).
- ▷ $next(\langle G, R_1, \ldots R_l\rangle) := \langle\sigma(B_1), \ldots, \sigma(B_m, R_1, \ldots, R_l)\rangle$ (backchaining) if there is a rule H:-B_1,..., B_m. and a substitution σ with $\sigma(H) = \sigma(G)$.

▷ **Example 634 (Continuing Example 633)** `?- nat(s(s(zero))).`

```
?- nat(s(zero)).
?- nat(zero).
Yes
```


This search process has as action the backchaining rule, which is really just a clever combination of the inference rules from Definition 630 applied backwards.

The backchaining rule takes the current goal G (the first one) and tries to find a substitution σ and a head H in a program clause, such that $\sigma(G) = \sigma(H)$. If such a clause can be found, it replaces G with the body of the rule instantiated by σ.

But we can extend querying from simple yes/no answers to programs that return values by simply using variables in queries. In this case, the ProLog interpreter returns a substitution.

Querying the Knowledge base (continued)

▷ If a query contains variables, then ProLog will return an answer substitution.

▷

```
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,4),has_motor(X).
?- car(Y)
?- has_wheels(Y,4),has_motor(Y).
Y = mybmw
?- has_motor(mybmw).
Y = mybmw
Yes
```

▷ If no instance of the statement in a query can be derived from the knowledge base, then the ProLog interpreter reports failure.

```
?- nat(s(s(0))).
?- nat(s(0)).
?- nat(0).
FAIL
No
```


With this, we can already do the “fallible Greeks” example from the introduction.

PROLOG: Are there Fallible Greeks?

- Program:

```
human(sokrates).
human(leibniz).
greek(sokrates).
fallible(X) :- human(X).
```

- **Example 635 (Query)** `?-fallible(X),greek(X).`
- Answer substitution: [sokrates$/X$]

 508 JACOBS UNIVERSITY

4.2.2 Programming as Search

In this subsection, we want to really use `ProLog` as a programming language, so let use first get our tools set up.

We will now discuss how to use a `ProLog` interpreter to get to know the language. The SWI `ProLog` interpreter can be downloaded from `http://www.swi-prolog.org/`. To start the `ProLog` interpreter with `pl` or `prolog` or `swipl` from the shell. The SWI manual is available at `http://gollem.science.uva.nl/SWI-Prolog/Manual/`

We will introduce working with the interpreter using unary natural numbers as examples: we first add the fact [1] to the knowledge base

```
unat(zero).
```

which asserts that the predicate `unat`[2] is `true` on the term `zero`. Generally, we can add a fact to the knowledge base either by writing it into a file (e.g. `example.pl`) and then "consulting it" by writing one of the following commands into the interpreter:

```
[example]
consult('example.pl').
```

or by directly typing

```
assert(unat(zero)).
```

into the `ProLog` interpreter. Next tell `ProLog` about the following rule

```
assert(unat(suc(X)) :- unat(X)).
```

which gives the `ProLog` runtime an initial (infinite) knowledge base, which can be queried by

```
?- unat(suc(suc(zero))).
Yes
```

Running `ProLog` in an `emacs` window is incredibly nicer than at the command line, because you can see the whole history of what you have done. Its better for debugging too. If you've never used `emacs` before, it still might be nicer, since its pretty easy to get used to the little bit of `emacs` that you need. (Just type "`emacs \&`" at the `UNIX` command line to run it; if you are on a remote terminal like `putty`, you can use "`emacs -nw`".).

If you don't already have a file in your home directory called "`.emacs`" (note the dot at the front), create one and put the following lines in it. Otherwise add the following to your existing `.emacs` file:

[1] for "unary natural numbers"; we cannot use the predicate `nat` and the constructor functions here, since their meaning is predefined in `ProLog`

[2] for "unary natural numbers".

```
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
    (autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
    (setq prolog-program-name "swipl") ; or whatever the prolog executable name is
    (add-to-list 'auto-mode-alist '("\\pl$" . prolog-mode))
```

The file `prolog.el`, which provides `prolog-mode` should already be installed on your machine, otherwise download it at `http://turing.ubishops.ca/home/bruda/emacs-prolog/`

Now, once you're in `emacs`, you will need to figure out what your "meta" key is. Usually its the alt key. (Type "control" key together with "h" to get help on using `emacs`). So you'll need a "`meta-X`" command, then type "`run-prolog`". In other words, type the meta key, type "`x`", then there will be a little window at the bottom of your `emacs` window with "`M-x`", where you type `run-prolog`[3]. This will start up the `SWI ProLog` interpreter, . . . et voilà!

The best thing is you can have two windows "within" your `emacs` window, one where you're editing your program and one where you're running `ProLog`. This makes debugging easier.

Depth-First Search with Backtracking

- ▷ So far, all the examples led to direct success or to failure. (simpl. KB)
- ▷ Search Procedure: top-down, left-right depth-first search
  - ▷ Work on the queries in left-right order.
  - ▷ match first query with the head literals of the clauses in the program in top-down order.
  - ▷ if there are no matches, fail and backtrack to the (chronologically) last point.
  - ▷ otherwise backchain on the first match , keep the other matches in mind for backtracking. (backtracking points)

 509 JACOBS UNIVERSITY

Note: We have seen before that depth-first search has the problem that it can go into loops. And in fact this is a necessary feature and not a bug for a programming language: we need to be able to write non-terminating programs, since the language would not be Turing-complete otherwise. The argument can be sketched as follows: we have seen that for Turing machines the halting problem is undecidable. So if all `ProLog` programs were terminating, then `ProLog` would be weaker than Turing machines and thus not Turing complete.

Backtracking by Example

```
has_wheels(mytricycle,3).
has_wheels(myrollerblade,3).
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,3),has_motor(X). % cars sometimes have 3 wheels
car(X):-has_wheels(X,4),has_motor(X).
?- car(Y).
?- has_wheels(Y,3),has_motor(Y). % backtrack point 1
Y = mytricycle % backtrack point 2
?- has_motor(mytricycle).
FAIL % fails, backtrack to 2
Y = myrollerblade % backtrack point 2
?- has_motor(myrollerblade).
FAIL % fails, backtrack to 1
?- has_wheels(Y,4),has_motor(Y).
Y = mybmw
?- has_motor(mybmw).
Y=mybmw
Yes
```

[3]Type "control" key together with "h" then press "m" to get an exhaustive mode help.

 510

Can We Use This For Programming?

- Question: What about functions? E.g. the addition function?
- Question: We do not have (binary) functions, in ProLog
- Idea (back to math): use a three-place predicate.

Example 636 `add(X,Y,Z)` stands for `X+Y=Z`

- Now we can directly write the recursive equations $X + 0 = X$ (base case) and $X + s(Y) = s(X + Y)$ into the knowledge base.

```
add(X,zero,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

- similarly with multiplication and exponentiation.

```
mult(X,o,o).
mult(X,s(Y),Z) :- mult(X,Y,W), add(X,W,Z).
expt(X,o,s(o)).
expt(X,s(Y),Z) :- expt(X,Y,W), mult(X,W,Z).
```

 511

Note: Viewed through the right glasses logic programming is very similar to functional programming; the only difference is that we are using $n+1$-ary relations rather than n-ary functions. To see how this works let us consider the addition function/relation example above: instead of a binary function $+$ we program a ternary relation `add`, where relation $\texttt{add}(X, Y, Z)$ means $X + Y = Z$. We start with the same defining equations for addition, rewriting them to relational style.

The first equation is straight-forward via our correspondance and we get the ProLog fact `add(X,zero,X).` For the equation $X + s(Y) = s(X + Y)$ we have to work harder, the straight-forward relational translation $\texttt{add}(X, s(Y), s(X + Y))$ is impossible, since we have only partially replaced the function $+$ with the relation `add`. Here we take refuge in a very simple trick that we can always do in logic (and mathematics of course): we introduce a new name Z for the offending expression $X + Y$ (using a variable) so that we get the fact $\texttt{add}(X, s(Y), s(Z))$. Of course this is not universally true (remember that this fact would say that "$X + s(Y) = s(Z)$ for all X, Y, and Z"), so we have to extend it to a ProLog rule `add(X,s(Y),s(Z))}{add(X,Y,Z).` which relativizes to mean "$X + s(Y) = s(Z)$ for all X, Y, and Z with $X + Y = Z$".

Indeed the rule implements addition as a recursive predicate, we can see that the recursion relation is terminating, since the left hand sides are have one more constructor for the successor function. The examples for multiplication and exponentiation can be developed analogously, but we have to use the naming trick twice.

More Examples from elementary Arithmetics

- **Example 637** We can also use the add relation for subtraction without changing the implementation. We just use variables in the "input positions" and ground terms in the other two (possibly very inefficient since "generate-and-test approach")

```
?-add(s(zero),X,s(s(s(zero)))).
X = s(s(zero))
Yes
```

▷ **Example 638** Computing the the n^{th} Fibonacci Number (0,1,1,2,3,5,8,13,...; add the last two to get the next), using the addition predicate above.

```
fib(zero,zero).
fib(s(zero),s(zero)).
fib(s(s(X)),Y):-fib(s(X),Z),fib(X,W),add(Z,W,Y).
```

▷ **Example 639** using ProLog's internal arithmetic: a goal of the form `?- D\;is\;e.` where e is a ground arithmetic expression binds D to the result of evaluating e.

```
fib(0,0).
fib(1,1).
fib(X,Y):- D is X - 1, E is X - 2,fib(D,Z),fib(E,W), Y is Z + W.
```

 512 JACOBS UNIVERSITY

Note: Note that the `is` relation does not allow "generate-and-test" inversion as it insists on the right hand being ground. In our example above, this is not a problem, if we call the `fib` with the first ("input") argument a ground term. Indeed, if match the last rule with a goal `?-` g`,Y.`, where g is a ground term, then g`-1` and g`-2` are ground and thus `D` and `E` are bound to the (ground) result terms. This makes the input arguments in the two recursive calls ground, and we get ground results for `Z` and `W`, which allows the last goal to succeed with a ground result for `Y`. Note as well that re-ordering the body literals of the rule so that the recursive calls are called before the computation literals will lead to failure.

Adding Lists to ProLog

▷ Lists are represented by terms of the form `[a,b,c,...]`

▷ first/rest representation `[F|R]`, where `R` is a rest list.

▷ predicates for `member`, `append` and `reverse` of lists in default ProLog representation.

```
member(X,[X|_]).
member(X,[_|R]):-member(X,R).
append([],L,L).
append([X|R],L,[X|S]):-append(R,L,S).
reverse([],[]).
reverse([X|R],L):-reverse(R,S),append(S,[X],L).
```

 513 JACOBS UNIVERSITY

Relational Programming Techniques

▷ Parameters have no unique direction "in" or "out"

```
:- rev(L,[1,2,3]).
:- rev([1,2,3],L1).
:- rev([1,X],[2,Y]).
```

▷ Symbolic programming by structural induction

```
rev([],[]).
rev([X,Xs],Ys) :- ...
```

▷ Generate and test

```
sort(Xs,Ys) :- perm(Xs,Ys), ordered(Ys).
```

 514

4.2.3 Logic Programming as Resolution Theorem Proving

We know all this already

- Goals, goal-sets, rules, and facts are just clauses. (called "Horn clauses")
- **Observation 640 (rule)** H:-B_1,…,B_n. *corresponds to* $H \vee \neg B_1 \vee \ldots \vee \neg B_n$ (*head the only positive literal*)
- **Observation 641 (goal set)** ?- G_1,…,G_n. *corresponds to* $\neg G_1 \vee \ldots \vee \neg G_n$
- **Observation 642 (fact)** F. *corresponds to the unit clause* F.
- **Definition 643** A Horn clause is a clause with at most one positive literal.
- Note: backchaining becomes (hyper)-resolution (special case for rule with facts)

$$\frac{P^{\mathsf{T}} \vee \mathbf{A} \quad P^{\mathsf{F}} \vee \mathbf{B}}{\mathbf{A} \vee \mathbf{B}}$$ positive, unit-resulting hyperresolution (PURR)

 515

PROLOG (Horn clauses)

- **Definition 644** Each clause contains at most one positive literal
  - $B_1 \vee \ldots \vee B_n \vee \neg A$ (`A:-`B_1,…,B_n.)
  - Rule clause: `fallible(X) :- human(X).`
  - Fact clause: `human(sokrates).`
  - Program: set of rule and fact clauses
  - Query: `?- fallible(X),greek(X).`

 516

PROLOG: Our Example

- Program:

```
human(sokrates).
human(leibniz).
greek(sokrates).
fallible(X) :- human(X).
```

- **Example 645 (Query)** `?- fallible(X),greek(X).`
- Answer substitution: $[\text{sokrates}/X]$

 517

Three Principal Modes of Inference

- ▷ Deduction: knowledge extension $$\frac{rains \Rightarrow wet\_street \quad rains}{wet\_street}D$$
- ▷ Abduction explanation $$\frac{rains \Rightarrow wet\_street \quad wet\_street}{rains}A$$
- ▷ Induction learning rules $$\frac{wet\_street \quad rains}{rains \Rightarrow wet\_street}I$$

 518 JACOBS UNIVERSITY

Bibliography

[AES01] Announcing the ADVANCED ENCRYPTION STANDARD (AES), 2001.

[BAHS] Mark Birbeck, Ben Adida, Ivan Herman, and Manu Sporny. RDFa 1.1 primer. W3C Working Draft, World Wide Web Consortium (W3C).

[BCHL09] Bert Bos, Tantek Celik, Ian Hickson, and Høakon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C Candidate Recommendation, World Wide Web Consortium (W3C), 2009.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force (IETF), 2005.

[Chu36] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, pages 40–41, May 1936.

[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.

[DH98] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, Internet Engineering Task Force (IETF), 1998.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[ECM09] ECMAScript language specification, December 2009. 5^{th} Edition.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force (IETF), 1999.

[Gen11a] General Computer Science; Problems for 320101 GenCS I. Online practice problems at `http://kwarc.info/teaching/GenCS1/problems.pdf`, 2011.

[Gen11b] General Computer Science: Problems for 320201 GenCS II. Online practice problems at `http://kwarc.info/teaching/GenCS2/problems.pdf`, 2011.

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

[HL11] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 331, feb 2011.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C recommendation, World Wide Web Consortium (W3C), 2004.

[Koh06] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006.

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

[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.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly, 2008.

[Pal] Neil/Fred's gigantic list of palindromes. web page at `http://www.derf.net/palindromes/`.

[RFC80] DOD standard internet protocol, 1980.

[RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 Specification. W3C Recommendation REC-html40, World Wide Web Consortium (W3C), April 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.

[SML10] The Standard ML basis library, 2010.

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

[Smo11] Gert Smolka. *Programmierung – eine Einführung in die Informatik mit Standard ML*. Oldenbourg Wissenschaftsverlag, corrected edition, 2011. ISBN: 978-3486705171.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, June 1936.

[vN45] John von Neumann. First draft of a report on the edvac. Technical report, University of Pennsylvania, 1945.

[w3c09] OWL 2 web ontology language: Document overview. W3C recommendation, World Wide Web Consortium (W3C), October 2009.

[XAM] apache friends - xampp. `http://www.apachefriends.org/en/xampp.html`.

[Zus36] Konrad Zuse. Verfahren zur selbsttätigen durchführung von rechnungen mit hilfe von rechenmaschinen. Patent Application Z23139/GMD Nr. 005/021, 1936.

Index