General Computer Science
320201 GenCS I & II Lecture Notes

Michael Kohlhase

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

March 6, 2014

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 2003-2014.

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, Koh13], a TEX/LATEX extension for semantic markup, which allows to export the contents into the eLearning platform PantaRhei.

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

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

Course Concept

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

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

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

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

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

Course Contents

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

[1] International University Bremen until Fall 2006

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

GenCS is not a programming course: even though it covers all three major programming paradigms (imperative, functional, and declarative programming). 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.

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

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

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

Acknowledgments

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

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

Contributors: The preparation of the course notes has been greatly helped by Ioan Sucan, who has done much of the initial editing needed for semantic preloading in SᴊTEX. 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, Lukas Kohlhase, Dmitrii Cucleschin, Aleksandar Gyorev, Tyler Buchmann, Bidesh Thapaliya, Dan Daniel Erdmann-Pham, Petre Munteanu, Utkrist Adhikari, Kim Philipp Jablonski, Aleksandar Gyorev, Tom Wiesing, Sourabh Lal, Nikhil Shakya, Otar Bichiashvili, Cornel Amariei, Enxhell Luzhnica.

Recorded Syllabus for 2013/14

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

Recorded Syllabus Fall Semester 2013:

#	date	until	slide	page
1	Sep 2.	through with admin	11	9
2	Sep 3.	at end of motivation	29	19
3	Sep 9.	recap of the first week	29	19
4	Sep 10.	introduced proofs with Peano axioms	38	28
5	Sep 16.	covered mathtalk & Alphabets	44	33
6	Sep 17.	Relations	51	38
7	Sep 23.	properties of functions	57	41
8	Sep 24.	SML component selection	64	45
8	Sep 30.	higher-order functions	70	49
9	Oct 1.	datatypes	86	58
10	Oct 7.	computation on ADTs	92	63
11	Oct 8.	constructor terms & substitutions	100	68
	Oct 14	Midterm		
12	Oct 15.	final Abstract Interpreter	108	73
13	Oct 28.	recursion relation & Fibonacci	113	78
14	Oct 29.	I/O and exceptions	125	84
15	Nov 4.	Codes, up to Morse Code	131	88
16	Nov 5.	UTF encodings	139	93
17	Nov 11.	Boolean Expressions	143	97
	Nov 12.	Midterm 2		
18	Nov 18.	Boolean Functions	150	101
19	Nov 19.	Elementary Complexity Theory	158	105
20	Nov 25.	Quine McCluskey	179	114
21	Nov 26.	Intro to Propositional Logic	186	119
22	Dec 2.	The miracle of logics	??	??
23	Nov 3.	Natural Deduction for Mathtalk	210	134

Recorded Syllabus Spring Semester 2014:

#	date	until	slide	page
1	Feb 3.	Graph Paths	223	151
2	Feb 4.	Balanced Binary Trees	232	157
3	Feb 10.	Universality of NAND and NOR	240	162
4	Feb 11.	Carry Chain adder	250	168
5	Feb 17.	Two's complement numbers	259	175
6	Feb 18.	RAM	272	183
7	Feb 24.	Basic Assembler	283	191
8	Feb 25.	Virtual machine basics	289	195
9	Mar 3.	Implementing VM arithmetics	??	??
10	Mar 4.	Compiling SW into VM	301	202

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

Syllabus Fall Semester 2012:

#	date	until	slide	page
1	Sep 3.	through with admin	11	9
2	Sep 4.	at end of motivation	29	19
3	Sep 10.	introduced proofs with Peano axioms	34	26
4	Sep 11.	covered mathtalk	42	31
5	Sep 17.	Sets and Russel's paradox	50	37
6	Sep 18.	functions	59	42
7	Sep 24.	SML pattern matching	68	48
8	Sep 25.	higher-order functions	70	49
9	Oct 1.	datatypes	86	58
10	Oct 2.	abstract procedures & computation	93	64
11	Oct 8.	substitutions and terms	102	69
12	Oct 9.	mutual recursion	115	79
13	Oct 15.	I/O and exceptions	125	84
14	Oct 29.	Sufficient Conditions for Prefix Codes	134	90
15	Oct 30.	UTF Codes	139	93
16	Nov 5.	Boolean Expressions	145	98
17	Nov 6.	Costs of Boolean Expressions	154	103
18	Nov 12.	Elementary Complexity Theory	159	105
19	Nov 13.	Naive Quine McCluskey	172	111
20	Nov 19.	Intro to Propositional Logic	186	119
21	Nov 20.	The miracle of logics	??	??
22	Nov 26.	Hilbert Calculus is sound	204	130
23	Nov 27.	Natural Deduction for Mathtalk	210	134
24	Dec 3.	Tableaux & Resolution	??	??
25	Dec 4.	Intellectual Property, Copyright, & Information Privacy	??	??

Spring Semester 2013:

#	date	until	slide	page
1	Feb 4.	Graph Isomorphism	221	150
2	Feb 5.	Parse Tree	227	154
3	Feb 11.	Universality of NAND and NOR	240	162
4	Feb 12.	Full adder	247	167
5	Feb 18.	Two's complement numbers	257	174
6	Feb 19.	RAM layout	276	185
7	Feb 25.	Basic Assembler	283	191
8	Feb 26.	Virtual machine basics	289	195
9	Mar 4.	Implementing VM arithmetics	??	??
10	Mar 5.	Compiling SW into VM	299	201
11	Mar 11.	Realizing Call Frames	310	213
12	Mar 12.	Compiling μML	322	219
13	Mar 18.	Universal Turing Machine	330	223
14	Mar 19.	Halting Problem	333	226
15	Apr 2.	Internet Protocol Suite	??	??
16	Apr 9.	Application Layer	??	??
17	Apr 15.	WWW Overview	??	??
18	Apr 16.	HTML	??	??
19	Apr 22.	Server Side Scripting	??	??
20	Apr 23.	Web Search: Queries	??	??
21	Apr 29.	public key encryption	??	??
22	Apr 30.	XPath	??	??
23	May 6.	Breadth-first search	??	??
24	May 7.	A^* search	??	??

Contents

Chapter 1

Getting Started with "General Computer Science"

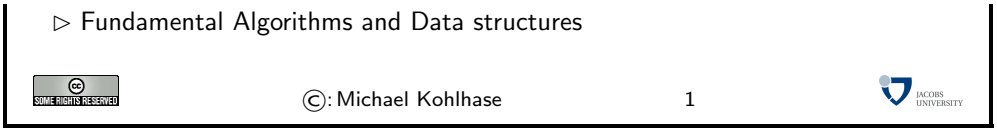
Jacobs University offers a unique CS curriculum to a special student body. Our CS curriculum is optimized to make the students successful computer scientists in only three years (as opposed to most US programs that have four years for this). In particular, we aim to enable students to pass the GRE subject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

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

1.1 Overview over the Course

Plot of "General Computer Science"

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



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

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

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

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

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

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

Algorithms are abstract representations of computation instructions

Data are representations of the objects the computations act on

Machines are representations of the devices the computations run on

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

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

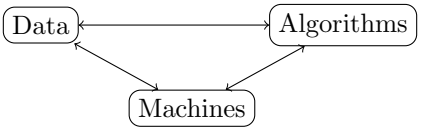


Figure 1.1: The three key ingredients of Computer Science

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

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

1.2 Administrativa

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

1.2.1 Grades, Credits, Retaking

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

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

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

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

Advanced Placement

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

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

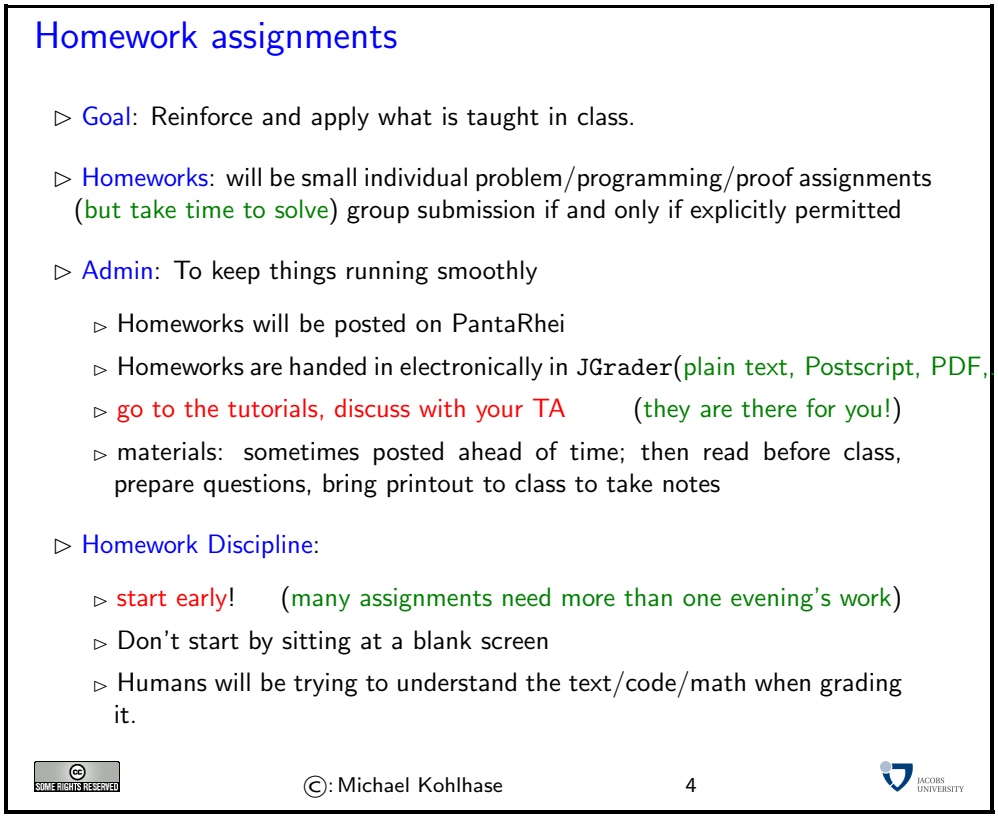
▷ Specifically: AP exams (oral) some time next week (see me for a date)

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

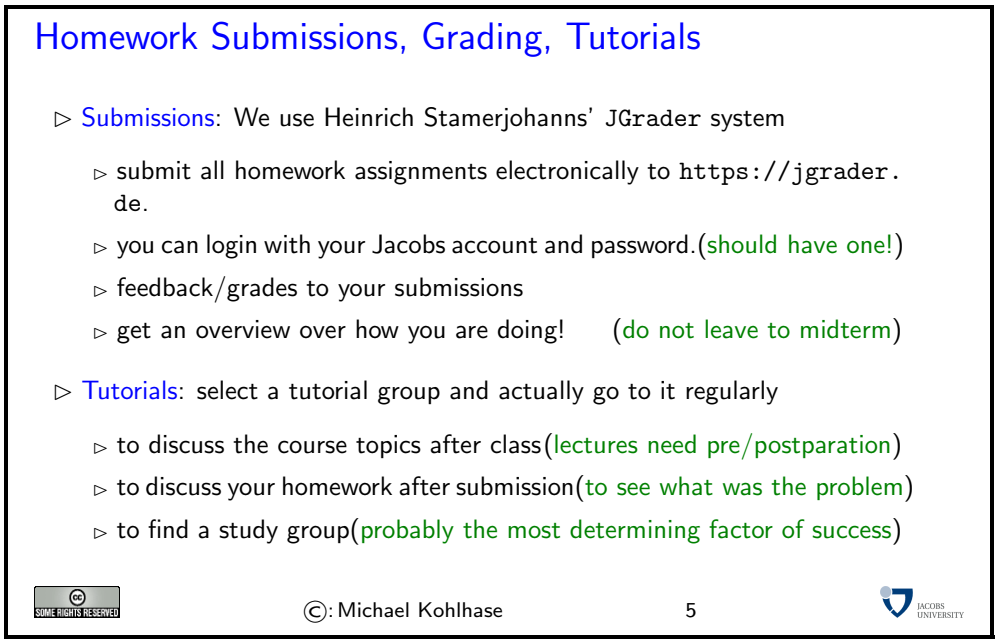
▷ Warning: you should be very sure of yourself to try (genius in C++ insufficient)

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

1.2.2 Homeworks, Submission, and Cheating



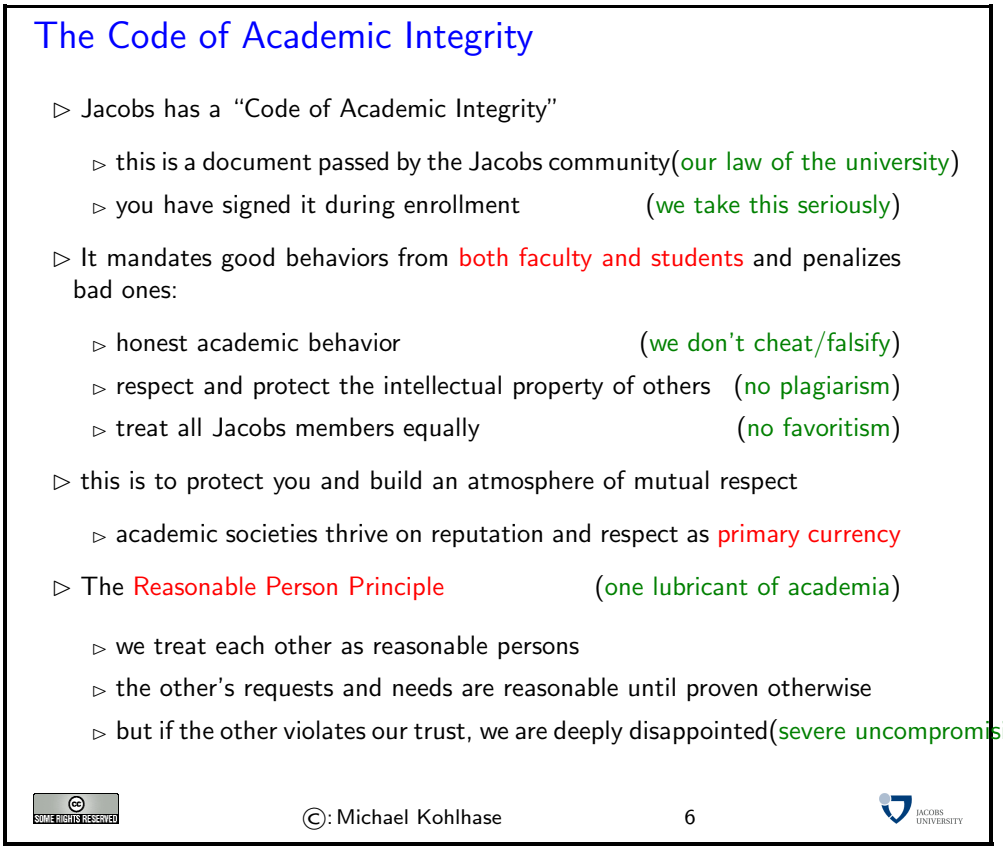
Homework assignments are a central part of the course, they allow you to review the concepts covered in class, and practice using them. They are usually directly based on concepts covered in the lecture, so reviewing the course notes often helps getting started.



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.



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.

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

The Academic Integrity Committee (AIC)

- ▷ Joint Committee by students and faculty(Not at "student honours court")
- ▷ Mandate: to hear and decide on any major or contested allegations, in particular,
  - ▷ the AIC decides based on evidence in a timely manner
  - ▷ the AIC makes recommendations that are executed by academic affairs
  - ▷ the AIC tries to keep allegations against faculty anonymous for the student
- ▷ we/you can appeal any academic integrity allegations to the AIC

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

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

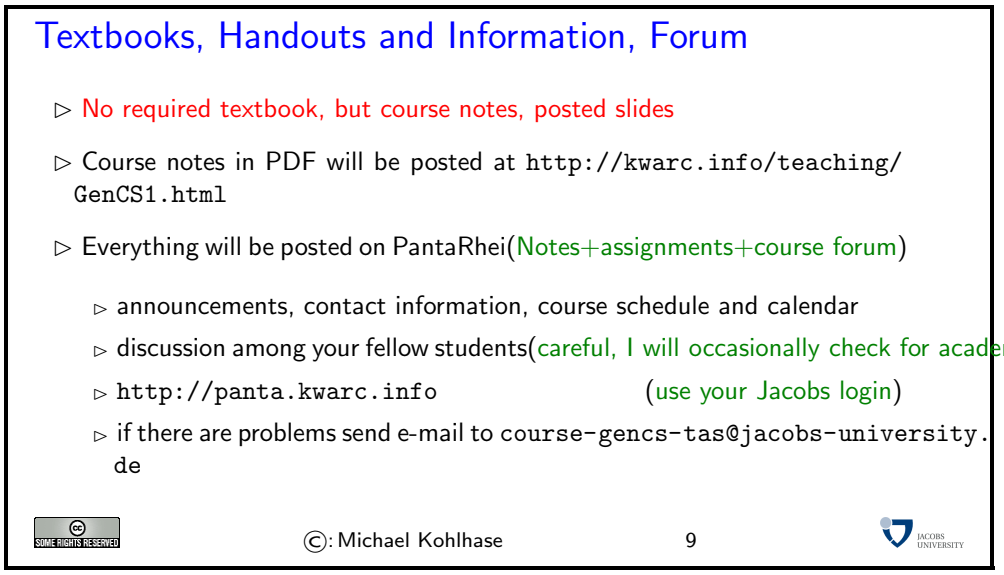
- ▷ There is no need to cheat in this course!! (hard work will do)
- ▷ cheating prevents you from learning (you are cutting your own flesh)
- ▷ if you are in trouble, come and talk to me (I am here to help you)
- ▷ We expect you to know what is useful collaboration and what is cheating
  - ▷ you will be required to hand in your own original code/text/math for all assignments
  - ▷ you may discuss your homework assignments with others, but if doing so impairs your ability to write truly original code/text/math, you will be cheating
  - ▷ copying from peers, books or the Internet is plagiarism unless properly attributed (even if you change most of the actual words)
  - ▷ more on this as the semester goes on . . .
- ▷  There are data mining tools that monitor the originality of text/code.
- ▷ Procedure: If we catch you at cheating(correction: if we suspect cheating)
  - ▷ we will confront you with the allegation (you can explain yourself)
  - ▷ if you admit or are silent, we impose a grade sanction and notify registrar
  - ▷ repeat infractions to go the AIC for deliberation (much more serious)
- ▷ Note: both active (copying from others) and passive cheating (allowing others to copy) are penalized equally

We are fully aware that the border between cheating and useful and legitimate collaboration is

difficult to find and will depend on the special case. Therefore it is very difficult to put this into firm rules. We expect you to develop a firm intuition about behavior with integrity over the course of stay at Jacobs.

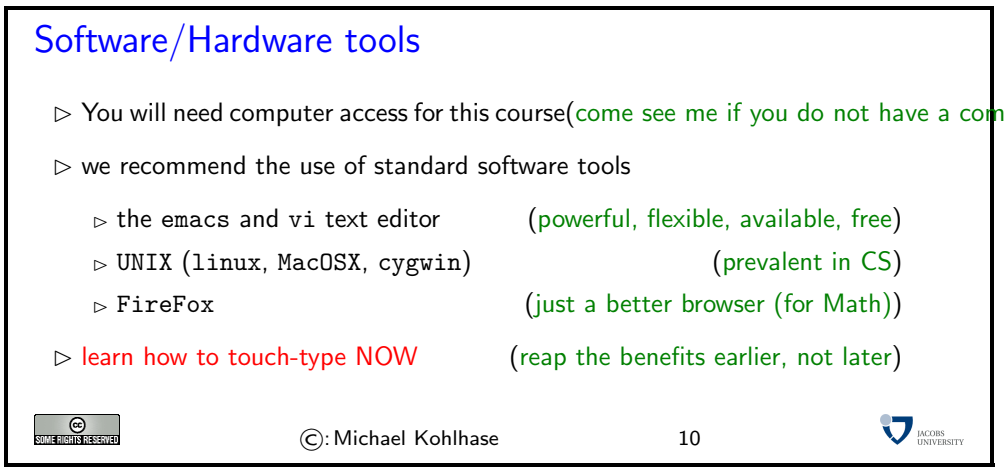
1.2.3 Resources

Even though the lecture itself will be the main source of information in the course, there are various resources from which to study the material.



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.

But of course, there is a wealth of literature on the subject of computational logic, and the references at the end of the lecture notes can serve as a starting point for further reading. We will try to point out the relevant literature throughout the notes.



Touch-typing: You should not underestimate the amount of time you will spend typing during your studies. Even if you consider yourself fluent in two-finger typing, touch-typing will give you

a factor two in speed. This ability will save you at least half an hour per day, once you master it. Which can make a crucial difference in your success.

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

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

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

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

Experiment: E-Learning with OMDoc/PantaRhei

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

Chapter 2

Motivation and Introduction

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

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

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

2.1 What is Computer Science?

What is Computer Science about?

- For instance: Software! (a hardware example would also work)
- **Example 2.1.1** 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:

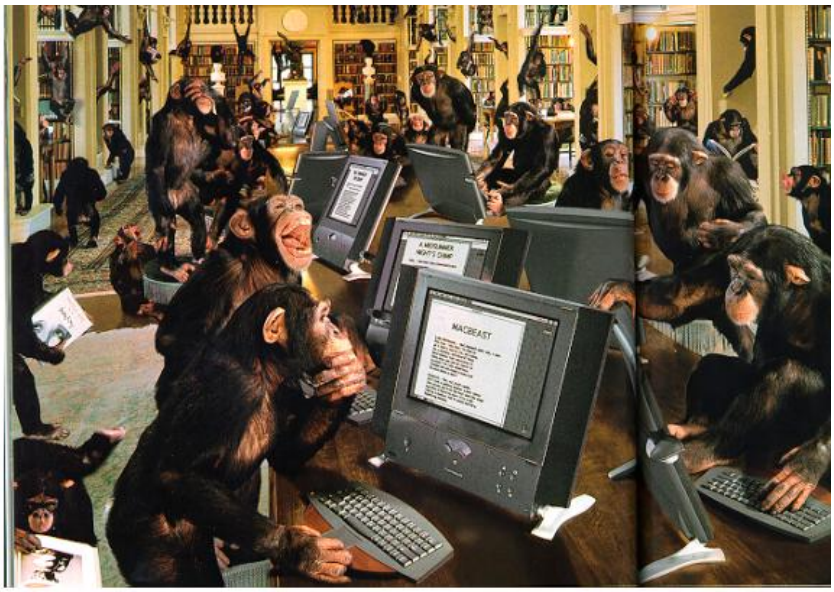
An Answer:

Let's hack

 ©: Michael Kohlhase 13

However, the result would probably be the following:

⚠ 2am in the IRC Quiet Study Area ⚠



 ©: Michael Kohlhase 14

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)

 ©: Michael Kohlhase 15

2.2 Computer Science by Example

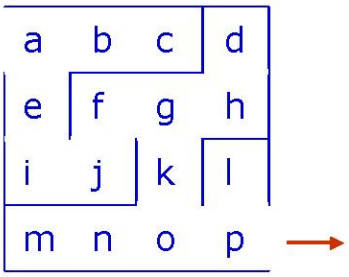
Thinking about a problem will involve thinking about the representations we want to use (after all, we want to work on the computer), which computations these representations support, and what constitutes a solutions to the problem.

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

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

Thinking about the problem

- ▷ Idea: Randomly knock out walls until we get a good maze
- ▷ Think about a grid of rooms separated by walls.
- ▷ Each room can be given a name.



- ▷ Mathematical Formulation:
  - ▷ a set of rooms: $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
  - ▷ Pairs of adjacent rooms that have an open wall between them.
- ▷ **Example 2.2.1** 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 2.2.2 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".

 ©: Michael Kohlhase 17

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.

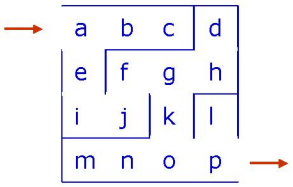
We will study graphs in more detail in this course, and cover them at an informal, intuitive level here to make our points.

Mazes as Graphs

▷ **Definition 2.2.3** Informally, a graph consists of a set of nodes and a set of edges. (a good part of CS is about graph algorithms)

▷ **Definition 2.2.4** 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

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

 ©: Michael Kohlhase 18

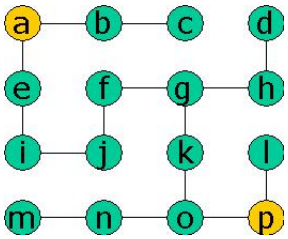
So now, we have identified the mathematical object, we will use to think about our algorithm, and indeed it is very abstract, which makes it relatively difficult for humans to work with. To get around this difficulty, we will often draw pictures of graphs, and use them instead. But we will always keep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

Mazes as Graphs (Visualizing Graphs via Diagrams)

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

Our maze

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



can be visualized as

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

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

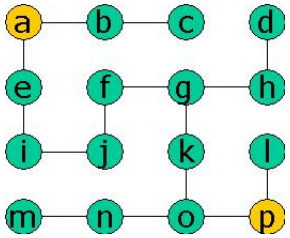
Unique solutions

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

▷ A: A path from a to p.

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

▷ A: The graph must be a tree.



Trees are special graphs, which we will now define.

Mazes as trees

▷ **Definition 2.2.5** Informally, a tree is a graph:

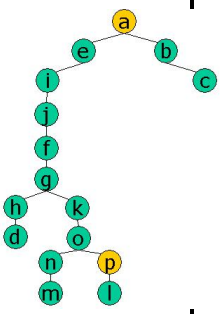
- ▷ with a unique root node, and
- ▷ each node having a unique parent.

▷ **Definition 2.2.6** 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

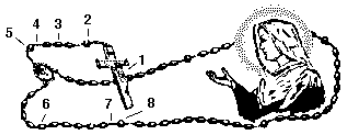
abstract (this gives us license to abstract away from many concrete details of the program and concentrate on the essentials).

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 2.2.7** An algorithm is a series of instructions to control a (computation) process



▷ **Example 2.2.8 (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 2.2.9 An algorithm is a collection of formalized rules that can be understood and executed, and that lead to a particular endpoint or result.

Example 2.2.10 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?

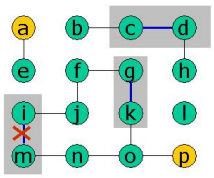
▷ When adding wall $\langle x, y\rangle$, we want to know if there is already a path from x to y in the tree.

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

Definition 2.2.11 (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 2.2.12 A partially constructed maze



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

How fast is our Algorithm?

▷ Is this a fast way to generate mazes?

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

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

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

Performance and Scaling

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

▷ Systematic analysis reveals performance characteristics.

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

size	performance		
n	$Q100n\mu$s	$Q7n^2\mu$s	$Q2^n\mu$s
1	$Q100\mu$s	$Q7\mu$s	$Q2\mu$s
5	$Q.5$ms	$Q175\mu$s	$Q32\mu$s
10	$Q1$ms	$Q.7$ms	$Q1$ms
45	$Q4.5$ms	$Q14$ms	$Q1.1Y$
100	...	...	...
1 000	...	...	...
10 000	...	...	...
1 000 000	...	...	...

What?! One year?

- ▷ $2^{10} = 1\,024$ ($Q1024\mu\text{s} \equiv Q1\text{ms}$)
- ▷ $2^{45} = 35\,184\,372\,088\,832$ ($Q3.5 \times 10^{13}\mu\text{s}{=}Q3.5 \times 10^{7}\text{s}{\equiv}Q1.1Y$)
- ▷ we denote all times that are longer than the age of the universe with $-$

size	performance		
n	$Q100n\mu$s	$Q7n^2\mu$s	$Q2^n\mu$s
1	$Q100\mu$s	$Q7\mu$s	$Q2\mu$s
5	$Q.5$ms	$Q175\mu$s	$Q32\mu$s
10	$Q1$ms1	$Q.7$ms	$Q1$ms
45	$Q4.5$ms	$Q14$ms	$Q1.1Y$
100	$Q100$ms	$Q7$s	$Q10^{16}Y$
1 000	$Q1$s	$Q12$min	$-$
10 000	$Q10$s	$Q20$h	$-$
1 000 000	$Q1.6$min	$Q2.5$mon	$-$

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

2.3 Other Topics in Computer Science

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

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

Is it correct?

- ▷ How will we know if we implemented our solution correctly?
  - ▷ What do we mean by "correct"?

▷ Will it generate the right answers?

▷ Will it terminate?

▷ Computer Science is about techniques for proving the correctness of programs

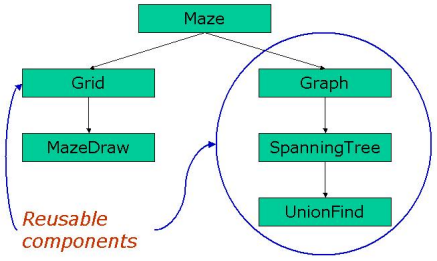
Modular design

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

▷ Grids, Graphs (with edges and nodes), Spanning trees, Union-find.

▷ With disciplined programming, we can write our program to reflect this structure.

▷ Modular designs are usually easier to get right and easier to understand.



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

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

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

2.4 Summary

The science in CS: not "hacking", but

▷ Thinking about problems abstractly.

▷ Selecting good structures and obtaining correct and fast algorithms/machines.

▷ Implementing programs/machines that are understandable and correct.

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

Part I

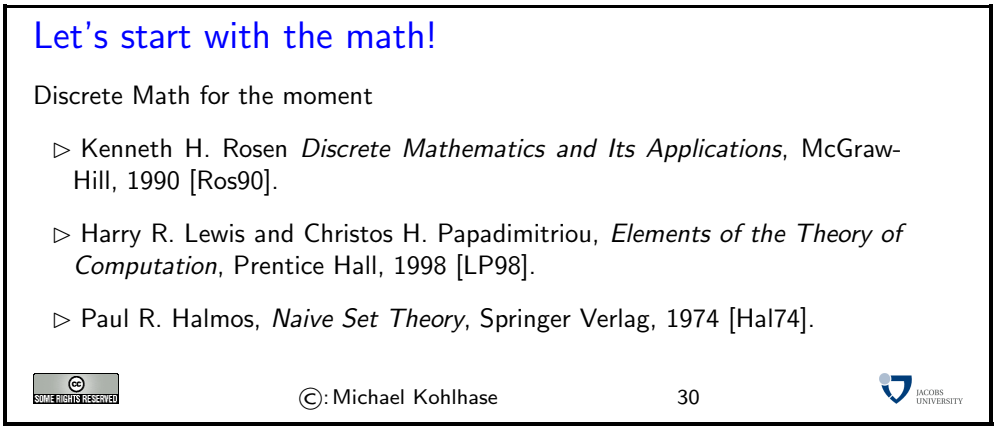
Representation and Computation

Chapter 3

Elementary Discrete Math

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

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



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

3.1 Mathematical Foundations: Natural Numbers

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

numbers in these general ways? To answer this question, we will try to find general ways to specify and manipulate arbitrary objects. Roughly speaking, this is what computation is all about.

Something very basic:

▷ Numbers are symbolic representations of numeric quantities.

▷ There are many ways to represent numbers (more on this later)

▷ let's take the simplest one (about 8,000 to 10,000 years old)

▷ we count by making marks on some surface.

▷ For instance //// stands for the number four(be it in 4 apples, or 4 worms)

▷ Let us look at the way we construct numbers a little more algorithmically,

▷ these representations are those that can be created by the following two rules.

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

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

▷ **Example 3.1.1** For ////, Apply the o-rule once and then the s-rule four times.

▷ **Definition 3.1.2** we call these representations unary natural numbers.

 31

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

meal. Instead it stands for a universal concept that can be applied to arbitrary objects.

Instead of using bones, the slash / is a more convenient symbol, but it is manipulated in the same way as in the most ancient times of mankind. The o-rule allows us to start with a blank slate or an empty container like a bowl. The s- or successor-rule allows to put an additional bone into a bowl with bones, respectively, to append a slash to a sequence of slashes. For instance //// stands for the number four — be it 4 apples, or 4 worms. This representation is constructed by applying the o-rule once and then the s-rule four times.

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

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

With the notion of a successor from Definition 3.1.3 we can formulate a set of assumptions (called axioms) about unary natural numbers. We will want to use these assumptions (statements we believe to be true) to derive other statements, which — as they have been obtained by generally accepted argumentation patterns — we will also believe true. This intuition is put more formally in Definition 3.2.4 below, which also supplies us with names for different types of statements.

A little more sophistication (math) please

▷ **Definition 3.1.3** We call a unary natural number the successor (predecessor) of another, if it can be constructing by adding (removing) a slash. (successors are created by the s-rule)

▷ **Example 3.1.4** /// is the successor of // and // the predecessor of ///.

▷ **Definition 3.1.5** The following set of axioms are called the Peano axioms (Giuseppe Peano $*1858$, $\dagger 1932$)

▷ **Axiom 3.1.6 (P1)** "" (aka. "zero") is a unary natural number.

▷ **Axiom 3.1.7 (P2)** Every unary natural number has a successor that is a unary natural number and that is different from it.

▷ **Axiom 3.1.8 (P3)** Zero is not a successor of any unary natural number.

▷ **Axiom 3.1.9 (P4)** Different unary natural numbers have different successors.

▷ **Axiom 3.1.10 (P5: Induction Axiom)** Every unary natural number possesses a property P, if

- ▷ zero has property P and (base condition)
- ▷ the successor of every unary natural number that has property P also possesses property P (step condition)

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

Note that the Peano axioms may not be the first things that come to mind when thinking about characteristic properties of natural numbers. Indeed they have been selected to to be minimal, so that we can get by with as few assumptions as possible; all other statements of properties can

be derived from them, so minimality is not a bug, but a feature: the Peano axioms form the foundation, on which all knowledge about unary natural numbers rests.

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

3.2 Reasoning about Natural Numbers

▷ Reasoning about Natural Numbers

▷ The Peano axioms can be used to reason about natural numbers.

▷ **Definition 3.2.1** An axiom is a statement about mathematical objects that we assume to be true.

▷ **Definition 3.2.2** 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 3.2.3** We call a sequence of inferences a derivation or a proof (of the last statement).

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

Definition 3.2.4 In general, a axiom or postulate is a starting point in logical reasoning with the aim to prove a mathematical statement or conjecture. A conjecture that is proven is called a theorem. In addition, there are two subtypes of theorems. The lemma is an intermediate theorem that serves as part of a proof of a larger theorem. The corollary is a theorem that follows directly from another theorem. A logical system consists of axioms and rules that allow inference, i.e. that allow to form new formal statements out of already proven ones. So, a proof of a conjecture starts from the axioms that are transformed via the rules of inference until the conjecture is derived.

We will now practice this reasoning on a couple of examples. Note that we also use them to introduce the inference system (see Definition 3.2.4) of mathematics via these example proofs.

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

Let's practice derivations and proofs

▷ **Example 3.2.5** ////////////// is a unary natural number

▷ **Theorem 3.2.6** */// is a different unary natural number than //.*

▷ **Theorem 3.2.7** *///// is a different unary natural number than //.*

▷ **Theorem 3.2.8** *There is a unary natural number of which /// is the successor*

▷ **Theorem 3.2.9** *There are at least 7 unary natural numbers.*

▷ **Theorem 3.2.10** *Every unary natural number is either zero or the successor of a unary natural number.* *(we will come back to this later)*

Induction for unary natural numbers

▷ **Theorem 3.2.11** *Every unary natural number is either zero or the successor of a unary natural number.*

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

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

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

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

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

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

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

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

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

This seems awfully clumsy, lets introduce some notation

▷ Idea: we allow ourselves to give names to unary natural numbers(we use n, m, l, k, n_1, n_2, ... as names for concr

▷ 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 3.2.12** 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 $s(s(s(s(s(s(s(s(s(s(o)))))))))))$ and $////////////$ by the symbol '12' (called "twelve")

▷ ...

▷ **Definition 3.2.13** We denote the set of all unary natural numbers with $\mathbb{N}_1$. (either representation)

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

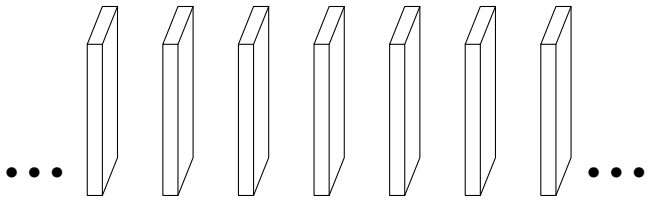
Theorem 3.2.11 is a very useful fact to know, it tells us something about the form of unary natural numbers, which lets us streamline induction proofs and bring them more into the form you may know from school: to show that some property P holds for every natural number, we analyze an arbitrary number 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 3.2.14** *Let $S_0, S_1, \ldots$ be a linear sequence of dominos, such that for any unary natural number i we know that*

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

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



The Domino Induction

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

P.1 We have to consider two cases

P.1.1 base case: i is zero:

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

□

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

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

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

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

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

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

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

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

3.3 Defining Operations on Natural Numbers

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

What can we do with unary natural numbers?

▷ So far not much (let's introduce some operations)

▷ **Definition 3.3.1 (the addition "function")** We "define" the addition operation $\oplus$ procedurally (by an algorithm)

- ▷ adding zero to a number does not change it.
  written as an equation: $n \oplus o = n$
- ▷ adding m to the successor of n yields the successor of $m \oplus n$.
  written as an equation: $m \oplus s(n) = s(m \oplus n)$

Questions: to understand this definition, we have to know

▷
- ▷ Is this "definition" well-formed? (does it characterize a mathematical object?)
- ▷ May we define "functions" by algorithms?(what is a function anyways?)

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

Definition 3.3.2 (Replacement) If we have a representation s of an object and we have an equation $l = r$, then we can obtain an object by replacing an occurrence of the sub-expression l in s by r and have $s = s'$.

In other words if we replace a sub-expression of s with an equal one, nothing changes. This is exactly what we will use the two defining equations from Definition 3.3.1 for in the following example

Example 3.3.3 (Computing the Sum Two and One) If we start with the expression $s(s(o))\oplus s(o)$, then we can use the second equation to obtain $s(s(s(o)) \oplus o)$ (replacing equals by equals), and – this time with the first equation $s(s(s(o)))$.

Observe: in the computation in Example 3.3.3 at every step there was exactly one of the two equations we could apply. This is a consequence of the fact that in the second argument of the two equations are of the form o and $s(n)$: by Theorem 3.2.11 these two cases cover all possible natural numbers and by **P**3 (see Axiom 3.1.8), the equations are mutually exclusive. As a consequence we do not really have a choice in the computation, so the two equations do form an "algorithm" (to the extend we already understand them), and the operation is indeed well-defined. The form of the arguments in the two equations in Definition 3.3.1 is the same as in the induction axiom, therefore we will consider the first equation as the base equation and second one as the step equation.

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

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

Addition on unary natural numbers is associative

▷ **Theorem 3.3.4** *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

P.2.1 base case: $n \oplus (m \oplus o) = n \oplus m = (n \oplus m) \oplus o$

P.2.2 step case:

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

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

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

□

We observe that In the proof above, the induction corresponds to the defining equations of $\oplus$; in particular base equation of $\oplus$ was used in the base case of the induction whereas the step equation

of $\oplus$ was used in the step case. Indeed computation (with operations over the unary natural numbers) and induction (over unary natural numbers) are just two sides of the same coin as we will see.

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

More Operations on Unary Natural Numbers

▷ **Definition 3.3.5** 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 3.3.6** The unary exponentiation operation can be defined by the equations $\mathsf{exp}(n, o) = s(o)$ and $\mathsf{exp}(n, s(m)) = n \odot \mathsf{exp}(n, m)$.

▷ **Definition 3.3.7** 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 3.3.8** The unary product operation can be defined by the equations $\bigodot_{i=o}^{o} n_i = s(o)$ and $\bigodot_{i=o}^{s(m)} n_i = n_{s(m)} \odot \bigodot_{i=o}^{m} n_i$.

In Definition 3.3.5, we have used the operation $\oplus$ in the right-hand side of the step-equation. This is perfectly reasonable and only means that we have eliminate more than one operator.

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

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

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

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

Definition 3.3.8 gives us the chance to elaborate on the process of definitions some more: When we define new operations such as the product over a sequence of unary natural numbers, we do have freedom of what to do with the corner cases, and for the "empty product" (the base case equation) we could have chosen

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

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

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

3.4 Talking (and writing) about Mathematics

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

Talking about Mathematics (MathTalk)

▷ **Definition 3.4.1** Mathematicians use a stylized language that

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

We call this language mathematical vernacular.

▷ **Definition 3.4.2** Abbreviations for Mathematical statements in MathTalk

- ▷ $\wedge$ 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 3.4.3** $\forall x.\exists y.x = y \Leftrightarrow \neg(x \neq y)$ reads

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

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

Peano Axioms in Mathtalk

▷ **Example 3.4.4** We can write the Peano Axioms in mathtalk: If we write $n \in \mathbb{N}_1$ for n *is a unary natural number*, and $P(n)$ for n *has property* P, then we can write

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

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

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

Definition 3.4.5 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 3.4.6 We can give the unary natural number //// the name φ. In a formula we write this as $\varphi :=$ (////) or //// $=: \varphi$.

Definition 3.4.7 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.[1] EdN:1

Example 3.4.8 The following is a pattern definition for the set intersection operator $\cap$:

$$(A \cap B) := \{x \,|\, x \in A \wedge x \in B\}$$

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

Definition 3.4.9 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 3.4.10 $\forall x.x \in \emptyset$ is an implicit definition for the empty set $\emptyset$. Indeed we can prove unique existence of $\emptyset$ by just exhibiting $\{\}$ and showing that any other set S with $\forall x.x \notin S$ we have $S \equiv \emptyset$. Indeed$S$ cannot have elements, so it has the same elements ad $\emptyset$, and thus $S \equiv \emptyset$.

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

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

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

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

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

3.5 Naive Set Theory

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

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

Understanding Sets

▷ Sets are one of the foundations of mathematics,

▷ and one of the most difficult concepts to get right axiomatically

▷ Early Definition Attempt: A set is "everything that can form a unity in the face of God". (Georg Cantor (∗1845, †1918))

▷ For this course: no definition; just intuition (naive set theory)

▷ To understand a set S, we need to determine, what is an element of S and what isn't.

▷ We can represent sets by

- ▷ listing the elements within curly brackets: e.g. $\{a, b, c\}$
- ▷ describing the elements via a property: $\{x \mid x \text{ has property } P\}$
- ▷ stating element-hood ($a \in S$) or not ($b \notin S$).

▷ **Axiom 3.5.1** Every set we can write down actually exists!(Hidden Assumption)

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

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

$$A \begin{matrix} a \\ b \\ b \end{matrix}$$

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

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

▷ Relations between Sets

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

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

Operations on Sets

- ▷ union: $(A \cup B) := \{x \mid x \in A \vee x \in B\}$
- ▷ 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: $(A_1, \ldots, A_n) := \{\langle a_1, \ldots, a_n\rangle \mid \forall i. 1 \leq i \leq n \Rightarrow a_i \in A_i\}$, 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 3.5.2** 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 $\bigcap_{i\in\{i\in\mathbb{N} \mid 1\leq i\leq n\}} S_i$.

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

Sizes of Sets

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

▷ **Definition 3.5.3** The size $\#(A)$ of a set A is the number of elements in A.

▷ **Conjecture 3.5.4** *Intuitively we should have the following identities:*

- ▷ $\#(\{a, b, c\}) = 3$
- ▷ $\#(\mathbb{N}) = \infty$ *(infinity)*
- ▷ $\#(A \cup B) \leq \#(A) + \#(B)$ *(⚠2 cases with ∞)*
- ▷ $\#(A \cap B) \leq min(\#(A), \#(B))$
- ▷ $\#(A \times B) = \#(A) \cdot \#(B)$

▷ But how do we prove any of them?(what does "number of elements" mean anyways?)

▷ Idea: We need a notion of "counting", associating every member of a set with a unary natural number.

▷ Problem: How do we "associate elements of sets with each other"?(wait for bijective functions)

Once we try to prove the identifies from Conjecture 3.5.4 we get into problems. Even though the notion of "counting the elements of a set" is intuitively clear (indeed we have been using that since we were kids), we do not have a mathematical way of talking about associating numbers with objects in a way that avoids double counting and skipping. We will have to postpone the discussion of sizes until we do.

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

Sets can be Mind-boggling

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

A less scary $\mathcal{S}$?

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

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

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

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

3.6 Relations and Functions

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

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

another (its power set).

Relations

- ▷ **Definition 3.6.1** $R \subseteq A \times B$ is a (binary) relation between A and B.
- ▷ **Definition 3.6.2** If $A = B$ then R is called a relation on A.
- ▷ **Definition 3.6.3** $R \subseteq A \times B$ is called total iff $\forall x \in A.\exists y \in B.\langle x, y\rangle \in R$.
- ▷ **Definition 3.6.4** $R^{-1} := \{\langle y, x\rangle \,|\, \langle x, y\rangle \in R\}$ 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 $(S \circ R) := \{\langle a, c\rangle \in (A \times C) \,|\, \exists b \in B.\langle a, b\rangle \in R \wedge \langle b, c\rangle \in S\}$
- ▷ **Example 3.6.5** relation $\subseteq$, $=$, has_color
- ▷ Note: we do not really need ternary, quaternary, ... relations
  - ▷ Idea: Consider $A \times B \times C$ as $A \times (B \times C)$ and $\langle a, b, c\rangle$ as $\langle a, \langle b, c\rangle\rangle$
  - ▷ we can (and often will) see $\langle a, b, c\rangle$ as $\langle a, \langle b, c\rangle\rangle$ different representations of the same object.

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

Properties of binary Relations

- ▷ **Definition 3.6.6 (Relation Properties)** A relation $R \subseteq A \times A$ is called
  - ▷ reflexive on A, iff $\forall a \in A.\langle a, a\rangle \in R$
  - ▷ irreflexive on A, iff $\forall a \in A.\langle a, a\rangle \notin R$
  - ▷ symmetric on A, iff $\forall a, b \in A.\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \in R$
  - ▷ asymmetric on A, iff $\forall a, b \in A.\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \notin R$
  - ▷ antisymmetric on A, iff $\forall a, b \in A.(\langle a, b\rangle \in R \wedge \langle b, a\rangle \in R) \Rightarrow a = b$
  - ▷ transitive on A, iff $\forall a, b, c \in A.(\langle a, b\rangle \in R \wedge \langle b, c\rangle \in R) \Rightarrow \langle a, c\rangle \in R$
  - ▷ equivalence relation on A, iff R is reflexive, symmetric, and transitive.
- ▷ **Example 3.6.7** The equality relation is an equivalence relation on any set.
- ▷ **Example 3.6.8** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.

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

Strict and Non-Strict Partial Orders

▷ **Definition 3.6.9** A relation $R \subseteq A \times A$ is called

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

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

▷ **Definition 3.6.10 (Linear order)** A partial order is called linear on A, iff all elements in A are comparable, i.e. if $\langle x, y\rangle \in R$ or $\langle y, x\rangle \in R$ for all $x, y \in A$.

▷ **Example 3.6.11** The $\leq$ relation is a linear order on $\mathbb{N}$(all elements are comparable)

▷ **Example 3.6.12** The "ancestor-of" relation is a partial order that is not linear.

▷ **Lemma 3.6.13** *Strict partial orderings are asymmetric.*

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

▷ **Lemma 3.6.14** *If $\preceq$ is a (non-strict) partial order, then $\prec := \{\langle a, b\rangle \mid (a \preceq b) \wedge a \neq b\}$ is a strict partial order. Conversely, if $\prec$ is a strict partial order, then $\preceq := \{\langle a, b\rangle \mid (a \prec b) \vee a = b\}$ is a non-strict partial order.*

Functions (as special relations)

▷ **Definition 3.6.15** $f \subseteq X \times Y$, 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 3.6.16** $f\colon X \rightharpoonup Y; x \mapsto y$ 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 3.6.17** $f(x) = y$ instead of $\langle x, y\rangle \in f$(function application)

▷ **Definition 3.6.18** 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 $f(x) = \bot$)

▷ **Definition 3.6.19** 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 3.6.20** $f\colon x \mapsto y$ if $\langle x, y\rangle \in f$ (arrow notation)

▷ **Definition 3.6.21** The identity function on a set A is defined as $\mathsf{Id}_A := \{\langle a, a\rangle \,|\, a \in A\}$.

⚠: 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 3.6.22** 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 3.6.23** 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 $\{f_a \,|\, a \in A\}$ of functions. (is a an argument or jut a "parameter"?)

▷ **Definition 3.6.24** To make the role of arguments extremely clear, we write functions in λ-notation. For $f = \{\langle x, E\rangle \,|\, x \in X\}$, where E is an expression, we write $\lambda x \in X.E$.

▷ **Example 3.6.25** The simplest function we always try everything on is the identity function:

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

▷ **Example 3.6.26** We can also to more complex expressions, here we take the square function

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

▷ **Example 3.6.27** λ-notation also works for more complicated domains. In

this case we have pairs as arguments.

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

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

Properties of functions, and their converses

▷ **Definition 3.6.28** A function $f\colon S\to T$ is called

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

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

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

▷ **Definition 3.6.29** 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 3.6.30** The function $\nu\colon\mathbb{N}_1\to\mathbb{N}$ with $\nu(o)=0$ and $\nu(s(n))=\nu(n)+1$ is a bijection between the unary natural numbers and the natural numbers from highschool.

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

Cardinality of Sets

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

▷ **Definition 3.6.31** We say that a set A is finite and has cardinality $\#(A)\in\mathbb{N}$, iff there is a bijective function $f\colon A\to\{n\in\mathbb{N}\mid n<\#(A)\}$.

▷ **Definition 3.6.32** We say that a set A is countably infinite, iff there is a bijective function $f\colon A\to\mathbb{N}$.

▷ **Theorem 3.6.33** *We have the following identities for finite sets A and B*

- ▹ $\#(\{a,b,c\})=3$ *(e.g. choose $f=\{\langle a,0\rangle,\langle b,1\rangle,\langle c,2\rangle\}$)*

- $\#(A \cup B) \leq \#(A) + \#(B)$
- $\#(A \cap B) \leq min(\#(A), \#(B))$
- $\#(A \times B) = \#(A) \cdot \#(B)$

▷ With the definition above, we can prove them (last three ⇝ 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 3.6.34** If $f \in A \to B$ and $g \in B \to C$ are functions, then we call

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

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

▷ **Definition 3.6.35** Let $f \in A \to B$ and $C \subseteq A$, then we call the function $(f|_C) := \{\langle c, b\rangle \in f \mid c \in C\}$ the restriction of f to C.

▷ **Definition 3.6.36** Let $f \colon A \to B$ 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,
- $\mathbf{Im}(f) := f(A)$ the image of 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

[Computing over Inductive Sets]Computing with Functions over Inductively Defined Sets

3.7 Standard ML: Functions as First-Class Objects

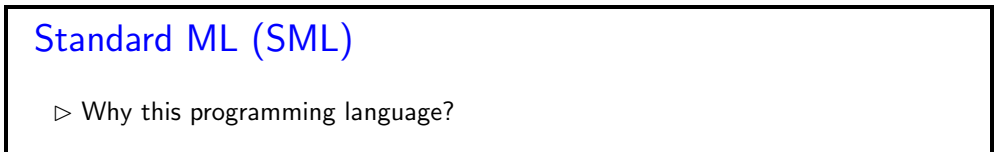
We will use the language SML for the course. This has four reasons

- 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 imperative programming languages (`C` and `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[1] — 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.



[1] The "Edinburgh LCF" system

▷ Important programming paradigm(Functional Programming (with static typing))
▷ because all of you are unfamiliar with it (level playing ground)
▷ 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 [Pau91]

⋙ Web resources: see the post on the course forum in PantaRhei.

▷ 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 3.7.1 (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 3.7.2 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 3.7.3 (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
```


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 (Component Selection)

▷ 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 3.7.4** anonymous variables(if we are not interested in one value)

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

▷ **Example 3.7.5** 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.

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.

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` ≙ empty list, `::` ≙ 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:

$$\texttt{upto(2,4)} \rightsquigarrow \texttt{2::upto(3,4)} \rightsquigarrow \texttt{2::(3::upto(4,4))} \rightsquigarrow \texttt{2::(3::(4::nil))} = \texttt{[2,3,4]}$$

▷ **Definition 3.7.6** 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

$$\texttt{diverges(1)} \rightsquigarrow \texttt{1 + diverges(2)} \rightsquigarrow \texttt{1 + (2 + diverges(3))} \rightsquigarrow \ldots$$

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 3.7.7** 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 3.7.8** Mapping a function over a list

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

▷ **Example 3.7.9** We can program the map function ourselves!

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

▷ **Example 3.7.10** declaring functions (yes, functions are normal values.)

```
- val identity = fn x => x;
val identity = fn : 'a -> 'a
- identity(5);
val it = 5 : int
```

▷ **Example 3.7.11** 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 3.7.12** $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ 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 3.7.13** $+\mathbb{Z}\mathbb{Z}, \mathbb{Z}$ with $+(3)(2)$ instead of $\langle 3, 2, (\rangle)$.

```
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 3.7.14** Call a procedure Cartesian, iff the argument type is a product type, call it cascaded, iff the result type is a function type.

▷ **Example 3.7.15** 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

▷ ▷ $e_1\ e_2\ e_3 \rightsquigarrow (e_1\ e_2)\ e_3$ (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)

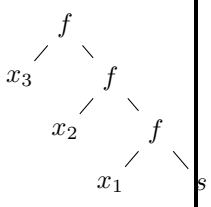
```
- fun cascaded_plus x y = x + y;
val cascaded_plus : int -> int -> int
```


We will now introduce two very useful higher-order functions. The folding operators iterate a binary function over a list given a start value. The folding operators come in two varieties: `foldl` ("fold left") nests the function in the right argument, and `foldr` ("fold right") in the left argument.

Folding Procedures

▷ **Definition 3.7.16** SML provides the left folding operator to realize a recurrent computation schema

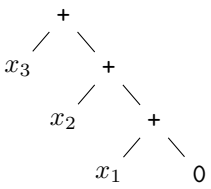
```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl f s [x1,x2,x3] = f(x3,f(x2,f(x1,s)))
```



We call the procedure f the iterator and s the startvalue

▷ **Example 3.7.17** 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.

Summing over a list is the prototypical operation that is easy to achieve. Note that a sum is just a nested addition. So we can achieve it by simply folding addition (a binary operation) over a list (left or right does not matter, since addition is commutative). For computing a sum we have to choose the start value 0, since we only want to sum up the elements in the list (and 0 is the neural element for addition).

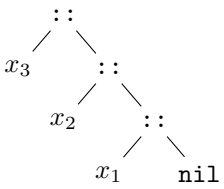
Note that we have used the binary function `op+` as the argument to the `foldl` operator instead of simply `+` in Example 3.7.17. The reason for this is that the infix notation for x `+`y is syntactic sugar for `op+(`x`,`y`)` and not for `+(`x`,`y`)` as one might think.

Note that we can use any function of suitable type as a first argument for `foldl`, including functions literally defined by `fn` x `=>`B or a n-ary function applied to $n-2$ arguments.

Folding Procedures (continued)

▷ **Example 3.7.18 (Reversing Lists)**

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



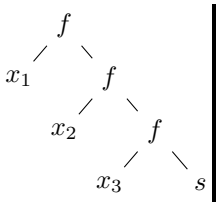
Thus the procedure `fun rev xs = foldl op:: nil xs` reverses a list

In Example 3.7.18, we reverse a list by folding the list constructor (which duly constructs the reversed list in the process) over the input list; here the empty list is the right start value.

Folding Procedures (`foldr`)

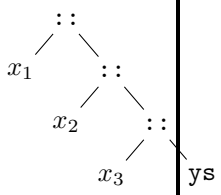
▷ **Definition 3.7.19** The right folding operator `foldr` is a variant of `foldl` that processes the list elements in reverse order.

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldr f s [x1,x2,x3] = f(x1,f(x2,f(x3,s)))
```



▷ **Example 3.7.20 (Appending Lists)**

```
foldr op:: ys [x1,x2,x3] = x1 :: (x2 :: (x3 :: ys))
```




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


In Example 3.7.20 we fold with the list constructor again, but as we are using `foldr` the list is not reversed. To get the append operation, we use the list in the second argument as a base case of the iteration.

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)

3.8 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.

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 3.8.1** $+ \subseteq (\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1$ *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

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 $\mathbb{N}_1$. With this, the proof of functionhood is a straightforward induction over the second argument.

Addition is a total Function

▷ **Lemma 3.8.2** *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 3.8.3** $+ : \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ *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 $s : \mathbb{N}_1 \to \mathbb{N}_1$

▷ 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 3.8.4** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$

▷ **Example 3.8.5** 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.

This observation directly leads us to a very important concept in computing.

Inductively Defined Sets

▷ **Definition 3.8.6** An inductively defined set $\langle S, C\rangle$ is a set S together with a finite set $C := \{c_i \mid 1 \leq i \leq n\}$ of k_i-ary constructors $c_i \colon S^{k_i} \to S$ with $k_i \geq 0$, such that

▷ if $s_i \in S$ for all $1 \leq i \leq k_i$, then $c_i(s_1, \ldots, s_{k_i}) \in S$ (generated by constructors)

▷ all constructors are injective, (no internal confusion)

▷ $\mathbf{Im}(c_i) \cap \mathbf{Im}(c_j) = \emptyset$ for $i \neq j$, and (no confusion between constructors)

▷ for every $s \in S$ there is a constructor $c \in C$ with $s \in \mathbf{Im}(c)$. (no junk)

▷ Note that we also allow nullary constructors here.

▷ **Example 3.8.7** $\langle \mathbb{N}_1, \{s, o\}\rangle$ is an inductively defined set.

▷ Proof: We check the three conditions in Definition 3.8.6 using the Peano Axioms

P.1 Generation is guaranteed by **P**1 and **P**2

P.2 Internal confusion is prevented **P**4

P.3 Inter-constructor confusion is averted by **P**3

P.4 Junk is prohibited by **P**5. □

This proof shows that the Peano Axioms are exactly what we need to establish that $\langle \mathbb{N}_1, \{s, o\}\rangle$ is an inductively defined set.

Now that we have invested so much elbow grease into specifying the concept of an inductively defined set, it is natural to ask whether there are more examples. We will look at a particularly important one next.

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: nil $\in \mathcal{L}[\mathbb{N}]$ and cons: $\mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$
- ▷ **Example 3.8.8** e.g. $[3,2,1] \mathrel{\hat{=}} \mathsf{cons}(3,\mathsf{cons}(2,\mathsf{cons}(1,\mathsf{nil})))$ and $[] \mathrel{\hat{=}} \mathsf{nil}$
- ▷ **Definition 3.8.9** 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 3.1.5
- ▷ **Axiom 3.8.10 (LP1)** nil $\in \mathcal{L}[\mathbb{N}]$ (generation axiom (nil))
- ▷ **Axiom 3.8.11 (LP2)** cons: $\mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$(generation axiom (cons))
- ▷ **Axiom 3.8.12 (LP3)** nil is not a cons-value
- ▷ **Axiom 3.8.13 (LP4)** cons is injective
- ▷ **Axiom 3.8.14 (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.

 ©: Michael Kohlhase 81 JACOBS UNIVERSITY

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 furthermore: that the $\mathbf{P}i$ and the $\mathbf{LP}i$ 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 $\mathcal{L}[\mathbb{N}]$.

A direct consequence of this observation is that

Corollary 3.8.15 *The set $\langle \mathbb{N}_1 \cup \mathcal{L}[\mathbb{N}], \{o, s, nil, cons\}\rangle$ is an inductively defined set in the sense of Definition 3.8.6.*

Operations on Lists: Append

▷ The append function @: $\mathcal{L}[\mathbb{N}] \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ concatenates lists
Defining equations: $\mathsf{nil}@l = l$ and $\mathsf{cons}(n,l)@r = \mathsf{cons}(n, l@r)$

▷ **Example 3.8.16** $[3,2,1]@[1,2] = [3,2,1,1,2]$ and $[]@[1,2,3] = [1,2,3] = [1,2,3]@[]$

▷ **Lemma 3.8.17** *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: $l = \mathsf{nil}$: must have $s = r$.

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

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

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

□

▷ **Corollary 3.8.18** *Append is a function* *(see, this just worked fine!)*

You should have noticed that this proof looks exactly like the one for addition. In fact, wherever we have used an axiom $\mathbf{P}i$ there, we have used an axiom $\mathbf{LP}i$ 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 3.8.19** $\lambda(\mathsf{nil}) = o$ and $\lambda(\mathsf{cons}(n,l)) = s(\lambda(l))$

▷ **Definition 3.8.20** $\rho(\mathsf{nil}) = \mathsf{nil}$ and $\rho(\mathsf{cons}(n,l)) = \rho(l)@\mathsf{cons}(n, \mathsf{nil})$.

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.

3.9 Inductively Defined Sets in SML

We are about to introduce one of the most powerful aspects of SML, its ability to let the user define 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

▷ **Definition 3.9.1** SML data type provide concrete syntax for inductively defined sets via the keyword `datatype` followed by a list of constructor declarations.

▷ **Example 3.9.2** We can declare a data type for unary natural numbers by

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

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

▷ **Observation 3.9.3** *We can define functions by (complete) case analysis over the constructors*

▷ **Example 3.9.4 (Converting `mynat` to `int`)**

```
fun num (zero) = 0 | num (suc(n)) = num(n) + 1;
val num = fn : mynat -> int
```

▷ **Example 3.9.5 (Missing Constructor Cases)**

```
fun incomplete (zero) = 0;
stdIn:10.1-10.25 Warning: match non-exhaustive
      zero => ...
val incomplete = fn : mynat -> int
```

▷ **Example 3.9.6 (Inconsistency)**

```
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 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: We try out our new data type, and indeed we can construct objects with the new constructors.

```
- 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
```

The beauty of the representation in user-defined types is that this affords powerful abstractions that allow to structure data (and consequently program functionality).

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

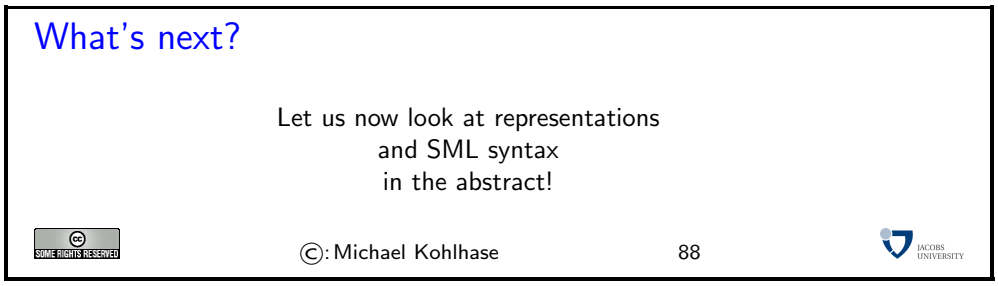
```
- area (Square 3.0)
9.0 : real
- area (Triangle(6.0, 6.0, Math.sqrt 72.0))
18.0 : real
```


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.

[Abstract Data Types and Term Languages]A Theory of SML: Abstract Data Types and Term Languages

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



In this chapter, we will study computation in functional languages in the abstract by building mathematical models for them.

Warning: The chapter on abstract data types we are about to start is probably the most daunting of the whole course (to first-year students), even though the concepts are very simple[2]. The reason for this seems to be that the models we create are so abstract, and that we are modeling language constructs (SML expressions and types) with mathematical objects (terms and sorts) that look very similar. The crucial step here for understanding is that sorts and terms are *similar to, but not equal* to SML types and expressions. The former are abstract mathematical objects, whereas the latter are concrete objects we write down for computing on a concrete machine. Indeed, the similarity in spirit is because we use sorts and terms to *model* SML types and expressions, i.e. to understand what the SML type checker and evaluators do and make predictions about their behavior.

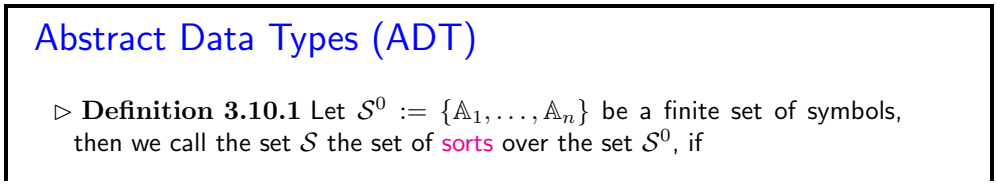
The idea of building representations (abstract mathematical objects or expressions) that model other objects is a recurring theme in the GenCS course; here we get a first glimpse of it: we use sorts terms to model SML types and expressions.

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.

3.10 Abstract Data Types and Ground Constructor Terms

Abstract data types are abstract objects that specify inductively defined sets by declaring a set of constructors with their sorts (which we need to introduce first).



[2] . . . in retrospect: second-year students report that they cannot imagine how they found the material so difficult to understand once they look at it again from the perspective of having survived GenCS.

▷ $\mathcal{S}^0 \subseteq \mathcal{S}$ (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 3.10.2** 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 3.10.3** 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 3.10.4** $\langle \{\mathbb{B}\}, \{[T\colon \mathbb{B}], [F\colon \mathbb{B}]\} \rangle$ is an abstract data for truth values.

▷ **Example 3.10.5** $\langle \{\mathbb{N}\}, \{[o\colon \mathbb{N}], [s\colon \mathbb{N} \to \mathbb{N}]\} \rangle$ represents unary natural numbers.

▷ **Example 3.10.6** $\langle \{\mathbb{N}, \mathcal{L}(\mathbb{N})\}, \{[o\colon \mathbb{N}], [s\colon \mathbb{N} \to \mathbb{N}], [\mathsf{nil}\colon \mathcal{L}(\mathbb{N})], [\mathsf{cons}\colon \mathbb{N} \times \mathcal{L}(\mathbb{N}) \to \mathcal{L}(\mathbb{N})]\} \rangle$ In particular, the term $\mathsf{cons}(s(o), \mathsf{cons}(o, \mathsf{nil}))$ represents the list $[1, 0]$

▷ **Example 3.10.7** $\langle \{\mathcal{S}\}, \{[\iota\colon \mathcal{S}], [\to\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}], [\times\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]\} \rangle$(what is this?)

The abstract data types in Example 3.10.4 and Example 3.10.5 are good old friends, the first models the build-in SML type `bool` and the second the unary natural numbers we started the course with.

In contrast to SML `datatype` declarations we allow more than one sort to be declared at one time (see Example 3.10.6). So abstract data types correspond to a group of `datatype` declarations.

The last example is more enigmatic. It shows how we can represent the set of sorts defined above as an abstract data type itself. Here, things become a bit mind-boggling, but it is useful to think this through and pay very close attention what the symbols mean.

First it is useful to note that the abstract data type declares the base sort $\mathcal{S}$, which is just an abstract mathematical object that models the set of sorts from Definition 3.10.1– without being the same, even though the symbols look the same. The declaration $[\iota\colon \mathcal{S}]$ just introduces a single base sort – if we have more base sorts to model, we need more declarations, and finally the two remaining declarations $[\to\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]$ and $[\times\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]$ introduce the sort constructors (see where the name comes from?) in Definition 3.10.1.

With Definition 3.10.3, 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 3.10.8** 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 3.10.9** 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 3.10.10** 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 3.10.5 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 3.10.6, 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 3.10.11** if t is a ground constructor term of sort $\mathbb{T}$, then $t \in \mathbb{T}$

▷ **Axiom 3.10.12** equality on ground constructor terms is trivial

▷ **Axiom 3.10.13** only ground constructor terms of sort $\mathbb{T}$ are in $\mathbb{T}$(induction axioms)

Note that these Peano axioms are left implicit – we never write them down explicitly, but they are there if we want to reason about or compute with expressions in the abstract data types.

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.

But before we embark on this, we build up our intuition with an extended example

Towards Understanding Computation on ADTs

▷ Aim: We want to understand computation with data from ADTs

▷ Idea: Let's look at a concrete example: abstract data type $\mathcal{B} := \langle\{\mathbb{B}\}, \{[T\colon \mathbb{B}], [F\colon \mathbb{B}]\}\rangle$ and the operations we know from mathtalk: $\wedge$, $\vee$, $\neg$, for "and", "or", and "not".

▷ Idea: think of these operations as functions on $\mathbb{B}$ that can be defined by "defining equations" e.g. $\neg(T) = F$, which we represent as $\neg(T) \rightsquigarrow F$ to stress the direction of computation.

▷ **Example 3.10.14** We represent the operations by declaring sort and equations.

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

Idea: Computation is just replacing equals by equals

$$\vee(T, \wedge(F, \neg(F))) \rightsquigarrow \vee(T, \wedge(F,T)) \rightsquigarrow \vee(T,F) \rightsquigarrow T$$

▷▷ Next Step: Define all the necessary notions, so that we can make this work mathematically.

3.11 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 3.11.1** 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 3.11.2** turn $\lambda(\mathsf{nil}) = o$ and $\lambda(\mathsf{cons}(n,l)) = s(\lambda(l))$ to $\lambda(\mathsf{nil}) \rightsquigarrow o$ and $\lambda(\mathsf{cons}(n,l)) \rightsquigarrow s(\lambda(l))$

▷ **Definition 3.11.3** 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 3.11.4** 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 3.11.5** An abstract computation is a computation that we can perform in our heads.(no real world constraints like memory size, time limits)

▷ **Definition 3.11.6** 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 3.10.6 (only that we abbreviate unary natural numbers).

Example: the functions ρ and @ on lists

▷ Consider the abstract procedures $\langle \rho::\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N}) ; \{\rho(\mathsf{cons}(n,l))\rightsquigarrow @(\rho(l),\mathsf{cons}(n,\mathsf{nil})),\rho(\mathsf{nil})\rightsquigarrow \mathsf{ni}\}\rangle$ and $\langle @::\mathcal{L}(\mathbb{N})\times\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N}) ; \{@(\mathsf{cons}(n,l),r)\rightsquigarrow\mathsf{cons}(n,@(l,r)),@(\mathsf{nil},l)\rightsquigarrow l\}\rangle$

▷ Then we have the following abstract computation

- ▷ $\rho(\mathsf{cons}(2,\mathsf{cons}(1,\mathsf{nil})))\rightsquigarrow @(\rho(\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))$ ($\rho(\mathsf{cons}(n,l))\rightsquigarrow @(\rho(l),\mathsf{cons}(n,\mathsf{nil}))$ with $n=2$ and
- ▷ $@(\rho(\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))\rightsquigarrow @(@(\rho(\mathsf{nil}),\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))$ ($\rho(\mathsf{cons}(n,l))\rightsquigarrow @(\rho(l),\mathsf{cons}(n,\mathsf{nil}))$ with $n=1$ and $l=\mathsf{nil}$)
- ▷ $@(@(\rho(\mathsf{nil}),\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))\rightsquigarrow @(@(\mathsf{nil},\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))$ ($\rho(\mathsf{nil})\rightsquigarrow\mathsf{nil}$)
- ▷ $@(@(\mathsf{nil},\mathsf{cons}(1,\mathsf{nil})),\mathsf{cons}(2,\mathsf{nil}))\rightsquigarrow @(\mathsf{cons}(1,\mathsf{nil}),\mathsf{cons}(2,\mathsf{nil}))$ ($@(\mathsf{nil},l)\rightsquigarrow l$ with $l=\mathsf{cons}(1,\mathsf{nil})$)
- ▷ $@(\mathsf{cons}(1,\mathsf{nil}),\mathsf{cons}(2,\mathsf{nil}))\rightsquigarrow \mathsf{cons}(1,@(\mathsf{nil},\mathsf{cons}(2,\mathsf{nil})))$ ($@(\mathsf{cons}(n,l),r)\rightsquigarrow \mathsf{cons}(n,@(l,r))$ with $n=1$, $l=$
- ▷ $\mathsf{cons}(1,@(\mathsf{nil},\mathsf{cons}(2,\mathsf{nil})))\rightsquigarrow \mathsf{cons}(1,\mathsf{cons}(2,\mathsf{nil}))$ ($@(\mathsf{nil},l)\rightsquigarrow l$ with $l=\mathsf{cons}(2,\mathsf{nil})$)
- ▷ Aha: ρ terminates on the argument $\mathsf{cons}(2,\mathsf{cons}(1,\mathsf{nil}))$

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 3.11.7 (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 3.11.8** 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 3.11.9** 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 3.11.10** 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 3.11.11** 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.

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

Constr. Terms with Variables: The Complete Definition

▷ **Definition 3.11.12** 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 $\mathcal{T}(\mathcal{A};\mathcal{V}) := \bigcup_{\mathbb{A}\in\mathcal{S}} \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$.

▷ **Definition 3.11.13** We define the depth of a term t by the way it is constructed.

$$\mathsf{dp}(t) := \begin{cases} 1 & \text{if } ([t\colon \mathbb{T}] \in \Sigma) \\ \max(a,b) & \text{if } t = \langle a, b\rangle \\ \mathsf{dp}(a)+1 & \text{if } t = (f(a)) \end{cases}$$

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.

3.12 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 3.12.1** 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 $\mathbf{supp}(\sigma) := \{x_{\mathbb{A}} \in \mathcal{V}_{\mathbb{A}} \,|\, \sigma(x_{\mathbb{A}}) \neq x_{\mathbb{A}}\}$ 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.

If $\sigma(\mathbf{A}) = \mathbf{B}$, then we say that σ instantiates $\mathbf{A}$ to $\mathbf{B}$ or that $\mathbf{B}$ is an instance of $\mathbf{A}$.

▷ **Notation 3.12.2** We denote the substitution σ with $\mathbf{supp}(\sigma) = \{x^i_{\mathbb{A}_i} \,|\, 1 \leq i \leq n\}$ 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 3.12.3 (Substitution Extension)** Let σ be a substitution, then we denote with $\sigma, [t/x_{\mathbb{A}}]$ the function $\{\langle y_{\mathbb{B}}, t\rangle \in \sigma \,|\, y_{\mathbb{B}} \neq x_{\mathbb{A}}\} \cup \{\langle x_{\mathbb{A}}, t\rangle\}$.
($\sigma, [t/x_{\mathbb{A}}]$ 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 3.12.2 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.

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

Substitution Application

▷ **Definition 3.12.4 (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 3.12.5** 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 3.12.6** $[a/x], [(f(b))/y], [a/z]$ instantiates $g(x, y, h(z))$ to $g(a, f(b), h(a))$. (sorts irrelevant here)

▷ **Definition 3.12.7** We give the defining equations for substitution application

- ▷ $\sigma(x_{\mathbb{A}}) = t$ if $[t/x_{\mathbb{A}}] \in \sigma$.
- ▷ $\sigma(y_{\mathbb{B}}) = y_{\mathbb{B}}$ if $y_{\mathbb{B}} \notin \mathbf{supp}(\sigma)$.
- ▷ $\sigma(\langle a, b\rangle) = \langle \sigma(a), \sigma(b)\rangle$.
- ▷ $\sigma(f(a)) = f(\sigma(a))$.

▷ this definition uses the inductive structure of the terms.

Note that a substitution in and of itself is a total function on variables. A substitution can be *extended* to a function on terms for substitution application. But we do not have a notation for this function. In particular, we may not write $[t/s]$, unless s ia a variable. And that would not make sense, since substitution application is completely determined by the behavior on variables. For instance if we have $\sigma(c) = t$ for a constant c, then the value t must be c itself by the definition below.

We now come to a very important property of substitution application: it conserves sorts, i.e. instances have the same sort.

Substitution Application conserves Sorts

▷ **Theorem 3.12.8** *Let $\mathcal{A}$ be an abstract data type, $t \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$, and σ a substitution on $\mathcal{A}$, then $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$.*

▷ Proof: by induction on $\mathsf{dp}(t)$ using Definition 3.11.12 and Definition 3.12.4

P.1 By Definition 3.11.12 we have to consider four cases

P.1.1 $[t\colon \mathbb{T}] \in \mathcal{D}$: $\sigma(t) = t$ by Definition 3.12.4, so $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ by construction.

P.1.2 $t \in \mathcal{V}_{\mathbb{T}}$: If $t \in \mathbf{supp}(\sigma)$, then $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ by Definition 3.12.1, otherwise $\sigma(t) = t$ and we get the assertion as in P,1,1.

P.1.3 $t = \langle a, b\rangle$ and $\mathbb{T} = \mathbb{A} \times \mathbb{B}$, where $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ and $b \in \mathcal{T}_{\mathbb{B}}(\mathcal{A};\mathcal{V})$: We have $\sigma(t) = \sigma(\langle a, b\rangle) = \langle \sigma(a), \sigma(b)\rangle$. By inductive hypothesis we have $\sigma(a) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ and $\sigma(b) \in \mathcal{T}_{\mathbb{B}}(\mathcal{A};\mathcal{V})$ and therefore $\langle \sigma(a), \sigma(b)\rangle \in \mathcal{T}_{(\mathbb{A}\times\mathbb{B})}(\mathcal{A};\mathcal{V})$ which gives the assertion.

P.1.4 $t = c(a)$, where $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ and $[c\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$: We have $\sigma(t) = \sigma(c(a)) = c(\sigma(a))$. By IH we have $\sigma(a) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ therefore $(c(\sigma(a))) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A};\mathcal{V})$. □

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.

3.13 Terms in Abstract Data Types

Unfortunately, constructor terms are still not enough to write down rules, as rules also contain the symbols from the abstract procedures. So we have to extend our notion of expressions yet again.

Are Constructor Terms Really Enough for Rules?

▷ **Example 3.13.1** $\rho(\mathsf{cons}(n, l)) \rightsquigarrow @(\rho(l), \mathsf{cons}(n, \mathsf{nil}))$.($\rho$ is not a constructor)

▷ Idea: need to include symbols for the defined procedures.(provide declarations)

▷ **Definition 3.13.2** 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 (for unique sorts).

▷ Idea: 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 $[f\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.

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 3.13.3** 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 $\mathcal{A}$, Σ, and $\mathcal{V}$), 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 $[c\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.

We denote the set of terms of sort $\mathbb{A}$ over $\mathcal{A}$, Σ, and $\mathcal{V}$ with $\mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$ and the set of all terms with $\mathcal{T}(\mathcal{A}, \Sigma; \mathcal{V})$.

Now that we have defined the concept of terms, we can ask ourselves which parts of terms are terms themselves. This results in the subterm relation, shich is surprisingly intricate to define: we need an intermediate relation, the immediate subterm relation. The subterm relation is the transitive-reflexive closure of that.

Subterms

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

▷ **Definition 3.13.4** 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 3.13.5** 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 3.13.6** $f(a)$ is a subterm of the terms $f(a)$ and $h(g(f(a), f(b)))$, and an immediate subterm of $h(f(a))$.

3.14 A Second Abstract Interpreter

Now that we have extended the notion of terms we can rethink the definition of abstract procedures and define an abstract notion of programs (coherent collections of abstract procedures).

For the final definition of abstract procedures we have to solve a tricky problem, which we have avoided in the treatment of terms above: To allow recursive procedures, we have to make sure that the (function) symbol that is introduced in the abstract procedure can already be used in the procedure body (the right hand side of the rules). So we have to be very careful with the signatures we use.

Abstract Procedures, Final Version

▷ **Definition 3.14.1 (Rules, final version)** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D} \rangle$ be an ab-

stract data type, Σ a signature over $\mathcal{A}$, and $f \notin (\mathbf{dom}(\mathcal{D}) \cup \mathbf{dom}(\Sigma))$ a symbol, then we call $f(s) \rightsquigarrow r$ a rule for $[f\colon \mathbb{A} \to \mathbb{B}]$ over Σ, if $s \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$ has no duplicate variables, constructors, or defined functions and $r \in \mathcal{T}_{\mathbb{B}}(\mathcal{A}, \Sigma, [f\colon \mathbb{A} \to \mathbb{B}]; \mathcal{V})$.

▷ Note: Rules are *well-sorted*, i.e. both sides have the same sort and *recursive*, i.e. rule heads may occur on the right hand side.

▷ **Definition 3.14.2 (Abstract Procedures, final version)**

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 $[f\colon \mathbb{A} \to \mathbb{R}]$.

▷ **Example 3.14.3** Let $\mathcal{A}$ be the union of the abstract data types from Example 3.10.6 and Example 3.10.14, then

$$\langle \mu{::}\mathbb{N} \times \mathcal{L}(\mathbb{N}) \to \mathbb{B}\,;\, \{\mu(\langle x_{\mathbb{N}}, \mathsf{nil}\rangle) \rightsquigarrow F, \mu(\langle x_{\mathbb{N}}, \mathsf{cons}(h_{\mathbb{N}}, t_{\mathcal{L}(\mathbb{N})})\rangle) \rightsquigarrow \vee(x = h, \mu(\langle y, t\rangle))\}\rangle$$

is an abstract procedure that induces the procedure declaration $[\mu\colon \mathbb{N} \times \mathcal{L}(\mathbb{N}) \to \mathbb{B}]$

Note that we strengthened the restrictions on what we allow as rules in Definition 3.14.2, so that matching of rule heads becomes unique (remember that we want to take the choice out of interpretation).

But there is another namespacing problem we have to solve. 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 3.14.4 (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 $[f\colon \mathbb{A}]$.

▷ **Example 3.14.5**

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

An Abstract Interpreter (second version)

▷ **Definition 3.14.6 (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 3.14.7** 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?

Note: that we have taken care in the definition of the concept of abstract procedures in Definition 3.14.1 that computation (replacement of equals by equals by rule application) is well-sorted. Moreover, the fact that we always apply instances of rules yields the analogy that rules are "functions" whose input/output pairs are the instances of the rules.

3.15 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.

⋙ **Example 3.15.1** 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 3.15.2 (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 3.15.3** 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 3.15.4 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 3.15.4** $\tau\colon \mathbb{N}_1 \to \mathbb{N}_1$, where $\tau(n) \rightsquigarrow \tau(3n+1)$ for n odd and $\tau(n) \rightsquigarrow \tau(n/2)$ for n even. (does this procedure terminate?)

▷ **Definition 3.15.5** 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 3.15.6** $\langle 4, 3\rangle$ 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 3.15.7** 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 3.15.8** 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 3.15.9** *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 3.15.10** 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 3.15.11** *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 3.15.11 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

Chapter 4

More SML

4.1 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.

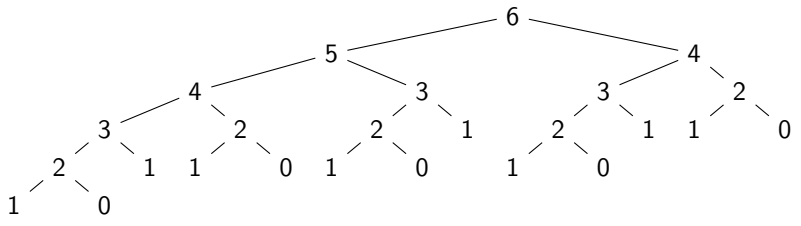
This makes the function extremely simple and straightforward to write down in SML. If we look at the recursion relation of this procedure, then we see that it can be visualized a tree, as each natural number has two successors (as the the function `fib` has two recursive calls in the step case).

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`)

```
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;
```


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`.

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.

4.2 Programming with Effects: Imperative Features in SML

So far, we have been programming in the "purely functional" fragment of SML. This will change now, indeed as we will see, purely functional programming languages are pointless, since they do not have any effects (such as altering the state of the screen or the disk). The advantages of functional languages like SML is that they limit effects (which pose difficulties for understanding the behavior of programs) to very special occasions

The hallmark of purely functional languages is that programs can be executed without changing the machine state, i.e. the values of variables. Indeed one of the first things we learnt was that variables are bound by declarations, and re-binding only shadows previous variable bindings. In the same way, the execution of functional programs is not affected by machine state, since the value of a function is fully determined by it arguments.

Note that while functions may be purely functional, the already SML interpreter cannot be, since it would be pointless otherwise: we *do want it to change the state of the machine*, if only to observe the computed values as changes of the (state of the) screen, similarly, we want it to be affected by the effects of typing with the keyboard.

But effects on the machine state can be useful in other situations, not just for input and output.

▷ Programming with Effects on the Machine State

▷ 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!

We will now look at the basics of input/output behavior and exception handling. They are state-changing and state-sensitive in completely different ways.

4.2.1 Input and Output

LIke in many programming languages Input/Output are handled via "streams" in SML. Streams are special, system-level data structures that generalize files, data sources, and periphery systems like printers or network connections. In SML, they are supplied by module `TextIO` from the the SML basic library [SML10]. Note that as SML is typed, streams have types as well.

Input and Output in SML

- ▷ Input and Output is handled via "streams" (think of possibly infinite strings)
- ▷ there are two predefined streams `TextIO.stdIn` and `TextIO.stdOut` (≙ keyboard input and screen)
- ▷ **Example 4.2.1 (Input)** via `TextIO.inputLine : TextIO.instream -> string`

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

- ▷ **Example 4.2.2 (Printing to Standard Output)** `print "sdfsfsdf''`

  `TextIO.print` prints its argument to the screen

  The user can also create streams as files: `TextIO.openIn` and `TextIO.openOut`.

⋙ Streams should be closed when no longer needed: `TextIO.closeIn` and `TextIO.closeOut`.

Input and Output in SML

- ▷ Problem: How to handle the end of input files?

  `TextIO.input1 : instream -> char option`

  attempts to read one char from an input stream (may fail)

▷ The SML basis library supplies the datatype

```
datatype 'a option = NONE | SOME of 'a
```

which can be used in such cases together with lots of useful functions.

IO Example: Copying a File – Char by Char

▷ **Example 4.2.3** The following function copies the contents of from one text file, `infile`, to another, `outfile` character by character:

```
fun copyTextFile(infile: string, outfile: string) =
  let
    val ins = TextIO.openIn infile
    val outs = TextIO.openOut outfile
    fun helper(copt: char option) =
      case copt of
           NONE => (TextIO.closeIn ins; TextIO.closeOut outs)
         | SOME(c) => (TextIO.output1(outs,c);
                       helper(TextIO.input1 ins))
  in
    helper(TextIO.input1 ins)
  end
```

Note the use of the `char option` to model the fact that reading may fail (EOF)

4.2.2 Programming with Exceptions

The first kind of stateful functionality is a generalization of error handling: when an error occurs, we do not want to to continue computation in the usual way, and we do not want to return regular values of a function. Therefore SML introduces the concept of "raising an exception". When an exception is raised, functional computation aborts and the exception object is passed up to the next level, until it is handled (usually) by the interpreter.

Raising Exceptions

▷ Idea: Exceptions are generalized error codes

▷ **Definition 4.2.4** An exception is a special SML object. Raising an exception e in a function aborts functional computation and returns e to the next level.

▷ **Example 4.2.5** 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>
```

Exceptions are first-class citizens in SML, in particular they

▷ ▷ have types, and

▷ can be defined by the user.

▷ **Example 4.2.6** user-defined exceptions(exceptions are first-class objects)

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

▷ **Example 4.2.7** exception constructors(exceptions are just like any other value)

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


Let us fortify our intuition with a simple example: a factorial function. As SML does not have a type of natural numbers, we have to give the `factorial` function the type `int -> int`; in particular, we cannot use the SML type checker to reject arguments where the factorial function is not defined. But we can guard the function by raising an custom exception when the argument is negative.

Programming with Exceptions

▷ **Example 4.2.8** 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

Note that this function is inefficient, as it checks the guard on every call, even though we can see that in the recursive calls the argument must be non-negative by construction. The solution is to use two functions, an interface function which guards the argument with an exception and a recursive function that does not check.

Programming with Exceptions (next attempt)

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

```
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
```

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

```
- safe_factorial(~1);
uncaught exception Factorial
raised at: stdIn:28.31-28.40
```


In the improved implementation, we see another new SML construct. `local` acts just like a `let`, only that it also that the body (the part between `in` and `end`) contains sequence of declarations and not an expression whose value is returned as the value of the overall expression. Here the identifier `fact` is bound to the recursive function in the definition of `safe_factorial` but unbound outside. This way we avoid polluting the name space with functions that are only used locally.

There is more to exceptions than just raising them to all the way the SML interpreter, which then shows them to the user. We can extend functions by exception handler that deal with any exceptions that are raised during their execution.

Handling Exceptions

▷ **Definition 4.2.9 (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 4.2.10 (SML Construct)** exception handler: `exp handle rules`

▷ **Example 4.2.11** 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!"
```

▷ **Example 4.2.12** the `read_integer` function (just to be complete)

```
exception NaN; (* Not a Number *)
fun read_integer () =
    let
        val instring = TextIO.inputLine(TextIO.stdIn);
    in
```


The function `factorial_driver` in Example 4.2.11 uses two functions that can raise exceptions: `safe_factorial` defined in Example 4.2.8 and the function `read_integer` in Example 4.2.12. Both are handled to give a nicer error message to the user.

Note that the exception handler returns situations of exceptional control to normal (functional)

computations. In particular, the results of exception handling have to of the same type as the ones from the functional part of the function. In Example 4.2.11, both the body of the function as well as the handers print the result, which makes them type-compatible and the function `factorial_driver` well-typed.

The previous example showed how to make use of the fact that exceptions are objects that are different from values. The next example shows how to take advantage of the fact that raising exceptions alters the flow of computation.

Using Exceptions for Optimizing Computation

▷ **Example 4.2.13 (Nonlocal Exit)** If we multiply a list of integers, we can stop when we see the first zero. So

```
local
  exception Zero
  fun p [] = 1
     | p (0::_) = raise Zero
     | p (h::t) = h * p t
in
  fun listProdZero ns = p ns
        handle Zero => 0
end
```

is more efficient than just

```
fun listProd ns = fold op* ns 1
```

and the more clever

```
fun listProd ns = if member 0 ns then 0 else fold op* ns 1
```

The optimization in Example 4.2.13 works as follows: If we call `listProd` on a list l of length n, then SML will carry out $n+1$ multiplications, even though we (as humans) know that the product will be zero as soon as we see that the k^{th} element of l is zero. So the last $n-k+1$ multiplications are useless.

In `listProdZero` we use the local function `p` which raises the exception `Zero` in the head 0 case. Raising the exception allows control to leapfrog directly over the entire rest of the computation and directly allow the handler of `listProdZero` to return the correct value (zero).

For more information on SML

RTFM ($\hat{=}$ "read the fine manuals")

Chapter 5

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.

5.1 Formal Languages

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

The Mathematics of Strings

▷ **Definition 5.1.1** 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 5.1.2** 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 5.1.3** We will often write a string $\langle c_1, \ldots, c_n\rangle$ as "$c_1 \ldots c_n$", for instance "abc" for $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}\rangle$

▷ **Example 5.1.4** Take $A = \{\mathtt{h}, 1, /\}$ as an alphabet. Each of the symbols h, 1, and / is a character. The vector $\langle /, /, 1, \mathtt{h}, 1\rangle$ is a string of length 5 over A.

▷ **Definition 5.1.5 (String Length)** Given a string s we denote its length with $|s|$.

▷ **Definition 5.1.6** The concatenation $\mathsf{conc}(s,t)$ 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 $\mathsf{conc}(s,t)$ as $s + t$ or simply st(e.g. $\mathsf{conc}(\texttt{"text"},\texttt{"book"}) = \texttt{"text"} + \texttt{"book"} = \texttt{"textbook"}$)

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 5.1.7** 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 5.1.8** If $A = \{a, b, c\}$, then $A^* = \{\epsilon, a, b, c, aa, ab, ac, ba, \ldots, aaa, \ldots\}$.

▷ **Definition 5.1.9** A set $L \subseteq A^*$ is called a formal language in A.

▷ **Definition 5.1.10** We use $c^{[n]}$ for the string that consists of n times c.

▷ **Example 5.1.11** $\#^{[5]} = \langle \#, \#, \#, \#, \#\rangle$

▷ **Example 5.1.12** 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 5.1.13** The concatenation $\mathsf{conc}(L_1, L_2)$ of two languages L_1 and L_2 over the same alphabet is defined as $\mathsf{conc}(L_1, L_2) := \{s_1s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$.

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.

The next step will be to define some operations and relations on these new objects: strings. As always, we are interested in well-formed sub-objects and ordering relations.

Substrings and Prefixes of Strings

▷ **Definition 5.1.14** 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 5.1.15** $\mathsf{conc}(/, 1, \mathtt{h})$ is a substring of $\mathsf{conc}(/, /, 1, \mathtt{h}, 1)$, whereas

conc$(/, 1, 1)$ is not.

▷ **Definition 5.1.16** A string p is a called a prefix of s (write $p \trianglelefteq s$), iff there is a string t, such that $s = \mathsf{conc}(p, t)$. p is a proper prefix of s (write $p \triangleleft s$), iff $t \neq \epsilon$.

▷ **Example 5.1.17** $text$ is a prefix of $textbook = \mathsf{conc}(text, book)$.

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

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 5.1.18** Let A be an alphabet and $<_A$ a strict 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 5.1.19** *$<_{lex}$ is a strict partial order on A^*. Moreover, if $<_A$ is linear on A, then $<_{lex}$ is linear on A^*.*

▷ **Example 5.1.20** Roman alphabet with a<b<c$\cdots$<z $\rightsquigarrow$ telephone book order ($computer <_{\mathsf{lex}} text, text <_{\mathsf{lex}} textbook$)

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.

5.2 Elementary Codes

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.

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 5.2.1** Let A and B be alphabets, then we call an injective function $c\colon A \to B^+$ a character code. A string $c(w) \in \{c(a) \mid a \in A\}$ is called a codeword.

▷ **Definition 5.2.2** A code is a called binary iff $B = \{0, 1\}$.

▷ **Example 5.2.3** 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 5.2.4** 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 5.2.5** The extension c' of c from the above example on the string "bbabc"

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

▷ **Definition 5.2.6** 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.,

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

Character codes in and of themselves are not that interesting: we want to encode strings in another alphabet. But they can be turned into mappings from strings to strings by extension: strings are sequences of characters, so we can encode them by concatenating the codewords.

We will now come to a paradigmatic example of an encoding function: the Morse code, which was used for transmitting text information as standardized sequences of short and long signals called "dots" and "dashes".

Morse Code

▷ In the early days of telecommunication the "Morse Code" was used to transmit texts, using long and short pulses of electricity.

▷ **Definition 5.2.7 (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 5.2.8** The Morse Code in the table above induces a character code $\mu\colon \mathcal{R} \to \{., -\}$.

Note: The Morse code is a character code, but its extension (which was actually used for transmission of texts) is not an injective mapping, as e.g. $\mu'(\texttt{AT}) = \mu'(\texttt{W})$. While this is mathematically a problem for decoding texts encoded by Morse code, for humans (who were actually decoding it) this ist not, since they understand the meaning of the texts and can thus eliminate nonsensical possibilities, preferring the string "ARRIVE AT 6" over "ARRIVE W 6".

The Morse code example already suggests the next topic of investigation: When are the extensions of character codes injective mappings (that we can decode automatically, without taking other information into account).

Codes on Strings

▷ **Definition 5.2.9** A function $c' : A^* \to B^*$ is called a code on strings or short string code if c' is an injective function.

▷ **Theorem 5.2.10** (⚠2) *There are character codes whose extensions are not string codes.*

▷ Proof: we give an example

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

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

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

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

Note that in contrast to checking for injectivity on character codes – where we have to do $n^2/2$ comparisons for a source alphabet A of size n, we are faced with an infinite problem, since A^* is infinite. Therefore we look for sufficient conditions for injectivity that can be decided by a finite process.

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 5.2.11** *The extension $c' : A^* \to B^*$ of a prefix code $c : A \to B^+$ 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. □

Even armed with Theorem 5.2.11, 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 5.2.12** *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 5.2.13** *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 $c(b) \in B^+$ □

▷ **Definition 5.2.14** The new character that makes an arbitrary code a prefix code in the construction of Theorem 5.2.13 is often called a stop character.

Theorem 5.2.13 allows another interpretation of the decodeability of the Morse code: it can be made into a prefix code by adding a stop character (in Morse code a little pause). In reality, pauses (as stop characters) were only used where needed (e.g. when transmitting otherwise meaningless character sequences).

5.3 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 capabilities in the time of punch cards (see below).

The ASCII Character Code

▷ **Definition 5.3.1** 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	{	|	}	∼	DEL

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

▷▷ Motivated by punchcards: The character 0 (binary 0000000) carries no information NUL, (used as dividers)
Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)

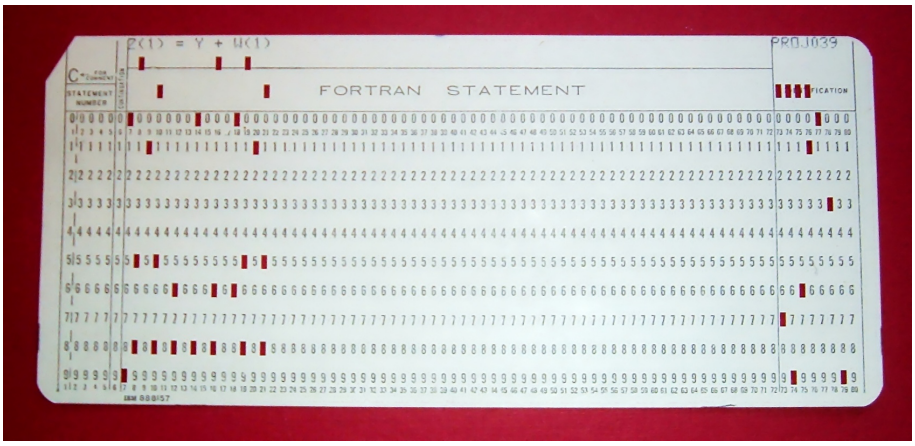
▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)

Punch cards were the the preferred medium for long-term storage of programs up to the late 1970s, since they could directly be produced by card punchers and automatically read by computers.

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 5.3.2** This punch card encoded the FORTRAN statement `Z(1) = Y + W(1)`



Up to the 1970s, computers were batch machines, where the programmer delivered the program to the operator (a person behind a counter who fed the programs to the computer) and collected the printouts the next morning. Essentially, each punch card represented a single line (80 characters) of program code. Direct interaction with a computer is a relatively young mode of operation.

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 5.3.3 (German `ASCII`)** remap e.g. [↦ Ä,] ↦ Ü in German `ASCII` ("`Apple ][`" comes out as "`Apple ÜÄ`")
- ▷ **Definition 5.3.4 (ISO-Latin (ISO/IEC 8859))** 16 Extensions of ASCII to 8-bit (256 characters) ISO-Latin 1 ≙ "Western European", ISO-Latin 6 ≙ "Arabic",ISO-Latin 7 ≙ "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)

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 5.3.5 (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 5.3.6** 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 5.3.7** Most (non-Chinese) characters have code points in $[1, 65536]$ (the basic multilingual plane).

▷ **Notation 5.3.8** For code points in the Basic Multilingual Plane (BMP), four digits are used, e.g. U+0058 for the character LATIN CAPITAL LETTER X;

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 5.3.9** 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 5.3.10 (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 5.3.11** 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 5.3.12** $ = U+0024 is encoded as 00100100 (1 byte)

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

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

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..

5.4 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 i

▷ E_{un} is based on the alphabet $\Sigma_{\mathsf{un}} := (C_{\mathsf{un}} \cup V \cup F^2_{\mathsf{un}} \cup B)$, where

- ▷ $C_{\mathsf{un}} := \{/\}^*$ is a set of constant names,
- ▷ $V := (\{\mathtt{x}\} \times \{1, \ldots, 9\} \times \{0, \ldots, 9\}^*)$ is a set of variable names,
- ▷ $F^2_{\mathsf{un}} := \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{mod}\}$ is a set of (binary) function names, and
- ▷ $B := (\{(,)\} \cup \{,\})$ is a set of structural characters.(⚠ ",","(",")" characters!)

▷ define strings in stages: $E_{\mathsf{un}} := \bigcup_{i \in \mathbb{N}} E^i_{\mathsf{un}}$, where

- ▷ $E^1_{\mathsf{un}} := (C_{\mathsf{un}} \cup V)$
- ▷ $E^{i+1}_{\mathsf{un}} := \{a, \mathsf{add}(a,b), \mathsf{sub}(a,b), \mathsf{mul}(a,b), \mathsf{div}(a,b), \mathsf{mod}(a,b) \mid a, b \in E^i_{\mathsf{un}}\}$

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

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^1_{un}, 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 $E^i_{\text{un}} \subseteq E^{i+1}_{\text{un}}$.

A formal Language for Unary Arithmetics (Examples)

▷ **Example 5.4.1** 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 ////// $\in E^1_{\text{un}}$, and x1902 $\in E^1_{\text{un}}$ and /// $\in E^1_{\text{un}}$,

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

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

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!)

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^i_{un}, the string s must already be a member of a set E^j_{un} 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 5.4.2** 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,///)) $\mapsto 6 + (x_{1902} \cdot 3)$(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,///)) $\mapsto 6 + (3 \cdot 3) = 15$

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.

Chapter 6

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[1] derive new information and knowledge from existing one.

6.1 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 6.1.1 (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 6.1.2 (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 6.1.3** Let $a \in E_{\mathsf{bool}}$. The minimal i, such that $a \in E^i_{\mathsf{bool}}$ is called the depth of a.

▷ $e_1 := ((-\mathtt{x1})+\mathtt{x3})$ (depth 3)

▷ $e_2 := ((-(\mathtt{x1}*\mathtt{x2}))+(\mathtt{x3}*\mathtt{x4}))$ (depth 4)

▷ $e_3 := ((\mathtt{x1}+\mathtt{x2})+((-((-\mathtt{x1})*\mathtt{x2}))+(\mathtt{x3}*\mathtt{x4})))$ (depth 6)

Boolean Expressions as Structured Objects.

▷ Idea: As strings in in E_{bool} are built up via the "union-principle", we can

[1] 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.

think of them as constructor terms with variables

▷ **Definition 6.1.4** 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 6.1.5** $\sigma\colon E_{\mathsf{bool}} \to \mathcal{T}_{\mathbb{B}}(\mathcal{B};\mathcal{V})$ 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 treat the strings and constructor terms synonymously. (σ is a (hidden) isomorphism)

▷ **Definition 6.1.6** 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 6.1.7** $\sigma(((-(\mathtt{x1}{*}\mathtt{x2})){+}(\mathtt{x3}{*}\mathtt{x4}))) = \overline{x_1 * x_2} + x_3 * x_4$

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

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 6.1.8** 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 6.1.9** A function $\varphi\colon V \to \mathcal{U}$ is called a variable assignment.

▷ **Definition 6.1.10** 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 $0 \mapsto 0, 1 \mapsto 1, + \mapsto \mathsf{min}, * \mapsto \mathsf{max}, - \mapsto \lambda x.1 - x$.

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 subexpressions and then combining them via the function that is the interpretation of the constructor at the head of the expression.

Evaluating Boolean Expressions

▷ **Example 6.1.11** 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

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

▷ What a mess!

Boolean Algebra

▷ **Definition 6.1.12** 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 6.1.13** We will use $\mathbb{B}$ for the universe, which can be either $\{0, 1\}$ or $\{\mathsf{T}, \mathsf{F}\}$

▷ **Definition 6.1.14** 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 6.1.15** $e_1 \equiv e_2$, *iff* $\mathcal{I}_\varphi((\overline{e1} + e_2) * (e_1 + \overline{e_2})) = \mathsf{T}$ *for all variable assignments* φ.

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.

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 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)

6.2 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 in Chapter8.

Boolean Functions

▷ **Definition 6.2.1** A Boolean function is a function from $\mathbb{B}^n$ to $\mathbb{B}$.

▷ **Definition 6.2.2** 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 6.2.3** 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 6.2.4** 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 $\varphi_{\langle c_1, \ldots, c_n \rangle}\colon x_i \mapsto c_i$. Dually, we say that e realizes f_e.

▷ **Theorem 6.2.5** $e_1 \equiv e_2$, *iff* $f_{e_1} = f_{e_2}$.

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 producttерm is a literal or the product of literals
- ▷ a clause or sumterm 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 6.2.6** 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 Boolean functions.

Normal Forms of Boolean Functions

▷ **Definition 6.2.7** 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 $S_c := \sum_{j=1}^n x_j^{1-c_j}$

▷ **Definition 6.2.8** The disjunctive normalform (DNF) of f is $\sum_{c \in f^{-1}(1)} M_c$ (also called the canonical sum (written as DNF(f)))

▷ **Definition 6.2.9** The conjunctive normal form (CNF) of f is $\prod_{c \in f^{-1}(0)} S_c$ (also called the canonical product (written as 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: $(x_1 + \overline{x_2} + x_3)\,(x_1 + \overline{x_2} + \overline{x_3})\,(\overline{x_1} + \overline{x_2} + x_3)$

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 6.2.10** The cost $C(e)$ of $e \in E_{\mathsf{bool}}$ is the number of operators in e.

▷ **Example 6.2.11** $C(\overline{x_1} + x_3) = 2$, $C(\overline{x_1 * x_2} + x_3 * x_4) = 4$, $C((x_1 + x_2) + (\overline{\overline{x_1} * x_2} + x_3 * x_4)) = 7$

▷ **Definition 6.2.12** 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?

6.3 Complexity Analysis for Boolean Expressions

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

▷ **Definition 6.3.1** 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 6.3.2** 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")

▷ that was easy wasn't it?

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 6.3.3** *These Ω-classes establish a ranking(increasing rank $\rightsquigarrow$ 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 6.3.4** $(\lambda n.263748) \in O(1)$

▷ **Example 6.3.5** $(\lambda n.26n + 372) \in O(n)$

▷ **Example 6.3.6** $(\lambda n.7n^2 - 372n + 92) \in O(n^2)$

▷ **Example 6.3.7** $(\lambda n.857n^{10} + 7342n^7 + 26n^2 + 902) \in O(n^{10})$

▷ **Example 6.3.8** $(\lambda n.3 \cdot 2^n + 72) \in O(2^n)$

▷ **Example 6.3.9** $(\lambda n.3 \cdot 2^n + 7342n^7 + 26n^2 + 722) \in O(2^n)$

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 6.3.10** 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 6.3.11** *Any $f\colon \mathbb{B}^n \to \mathbb{B}$ 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 $e_n := \mathsf{CNF}(f)$ has $\frac{2^n}{2}$ clauses or less or $\mathsf{DNF}(f)$ 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" $\kappa\colon \mathbb{N} \to \mathbb{N}$ 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.

P.2 P.3 P.4

We can do better (if we accept complicated witness)

▷ **Theorem 6.3.12** *Let $\kappa(n) := \mathsf{max}(\{C(f) \mid f\colon \mathbb{B}^n \to \mathbb{B}\})$, then $\kappa \in O(2^n)$.*

▷ Proof: we show that $\kappa(n) \leq 2^{n+1}$ by induction on n

P.1.1 base case ($n = 1$): 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 $\kappa(1) \leq 2^2$.

P.1.2 step case ($n > 1$):

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 $e_i \in E_{\mathsf{bool}}$, such that $f_i = f_{e_i}$ and $C(e_i) = 2^n$.(IH)

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

□

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

▷ 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 6.3.14** $\mathcal{E}_n := \{e \in E_{\mathsf{bool}} \mid e \text{ has } n \text{ variables and } C(e) \leq \kappa(n)\}$

▷ **Lemma 6.3.15** $\#(\mathcal{E}_n) \geq \#(\mathbb{B}^n \to \mathbb{B})$

▷ Proof:

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

P.2 $C(f_n) = \mathsf{min}(\{C(e) \mid f_e = f_n\})$ 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 6.3.16** $\#(\mathcal{E}_n) \geq 2^{2^n}$

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

P.2 An Upper Bound For $\kappa(n)$-cost Expressions

▷ 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 6.3.17** Given a cost c let $\Lambda(e)$ be the length of e considering variables as single characters. We define

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

▷ **Lemma 6.3.18** $\sigma(n) \leq 5n$ *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 6.3.19** *max*$(\{\Lambda(e) \,|\, e \in \mathcal{E}_n\}) \leq 5 \cdot \kappa(n)$

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 6.3.20** $\mathcal{E}_n \subseteq \bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i$ *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

$$\#\left(\left(\bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i\right)\right) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} (n+7)^i = \frac{(n+7)^{5\kappa(n)+1}-1}{n+6}$$

□

Solving for $\kappa(n)$

▷ $\frac{(n+7)^{5\kappa(n)+1}-1}{n+6} \geq 2^{2^n}$

▷ $(n+7)^{5\kappa(n)+1} \geq 2^{2^n}$ (as $(n+7)^{5\kappa(n)+1} \geq \frac{(n+7)^{5\kappa(n)+1}-1}{n+6}$)

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

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

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

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

 ©: Michael Kohlhase 164

6.4 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 6.4.1** We will use the following ordering on $\mathbb{B}$: F $\leq$ 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 6.4.2** A monomial M implies a Boolean function $f: \mathbb{B}^n \to \mathbb{B}$ (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 6.4.3** 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'.

 ©: Michael Kohlhase 165

With these definitions, we can convince ourselves that sub-monomials dominate 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 f becomes "less true". In particular, if we have too few 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 6.4.4** *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 6.4.5** An implicant M of f is a prime implicant of f iff no sub-monomial of M is an implicant of f.

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 6.4.6** *Given a Boolean function $f \neq \lambda x.\mathsf{F}$ 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 $f = \lambda x.\mathsf{T}$.

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 Now, 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 6.4.4.

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 6.4.7** Let M be a set of monomials, then
  - ▹ $\mathcal{R}(M) := \{m \,|\, (m\,x) \in M \wedge (m\,\overline{x}) \in M\}$ is called the set of resolvents of M
  - ▹ $\widehat{\mathcal{R}}(M) := \{m \in M \,|\, m \text{ has a partner in } M\}$ ($n\,\overline{x_i}$ and $n\,x_i$ are partners)
- ▷ **Definition 6.4.8 (Algorithm)** Given $f\colon \mathbb{B}^n \to \mathbb{B}$
  - ▹ let $M_0 := \mathsf{DNF}(f)$ 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 $P_{\mathsf{prime}} := \bigcup_{j=1}^{n} P_j$

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	$x1^1\,x2^1\,x3^1$

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

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

▷ 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 6.4.9** *(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 6.4.10** *QMC_1 terminates after at most n rounds.*

▷ **Corollary 6.4.11** *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 6.4.12 (Algorithm)** Generate and test!

▷ enumerate $S_p \subseteq P_{prime}$, i.e., all possible combinations of prime impli-

cants 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 6.4.13** $P_{prime} = \{x1\,x3, \overline{x2}\}$, 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 6.4.14** 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 6.4.15**

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

▷ **Definition 6.4.16** 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 6.4.17** *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 □

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	$x1^1\,x2^1\,x3^1\,x4^1$

A complex Example for QMC (QMC$_1$)

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

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

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 6.4.18** $\langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F}\rangle$ and $\langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T}\rangle$ resolve to $\langle \mathsf{F},\mathsf{F},\mathsf{F},X\rangle$.

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	$x1^1\,x2^1\,x3^1\,x4^1$

$\rightsquigarrow$

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

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
$\overline{x1}\,\overline{x2}\,x4$	T	F	T	F	F	F	F	F
$\overline{x1}\,x3\,x4$	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
$\overline{x1}\,x3\,x4$	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
$\overline{x1}\,x3\,x4$	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 ⚠

6.5 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 6.5.1** 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 6.5.2 (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

	$\overline{AB}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{CD}$	m_0	m_4	m_{12}	m_8
$\overline{C}D$	m_1	m_5	m_{13}	m_9
CD	m_3	m_7	m_{15}	m_{11}
$C\overline{D}$	m_2	m_6	m_{14}	m_{10}

torus
2/4/8/16-groups

▷ 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 6.5.3

#	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:

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

▷ in the red/magenta 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 $A\,\overline{B} + A\,\overline{C} + B\,C\,\overline{D}$

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!

Chapter 7

Propositional Logic

7.1 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 7.1.1** 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 7.1.2** 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 $\widehat{\ }$ 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[1].

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 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 7.1.3** 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 7.1.4** $x \vee x$ is satisfiable and falsifiable.
- ▷ **Example 7.1.5** $x \vee \neg x$ is valid and $x \wedge \neg x$ is unsatisfiable.
- ▷ **Notation 7.1.6** (alternative) Write $[\![e]\!]_\varphi^{\mathcal{M}}$ 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 c
- ▷ **Definition 7.1.7 (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

[1] 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.

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 7.1.8** 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 7.1.9** $\mathcal{I}_\varphi(\mathsf{love}(\mathsf{bill},\mathsf{mary}) \wedge (\mathsf{love}(\mathsf{mary},\mathsf{john}) \Rightarrow \mathsf{hate}(\mathsf{bill},\mathsf{john}))) = \mathsf{T}$ 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 7.1.10** $g_1 \wedge g_3 \wedge \mathsf{love}(\mathsf{mary},\mathsf{john}) \models \mathsf{hate}(\mathsf{bill},\mathsf{john})$

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 $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.

▷ $\mathbf{K} := (P \Rightarrow Q \Rightarrow P)$, $\mathbf{S} := ((P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R)$

▷ $\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}\mathsf{MP}$ $\qquad$ $\dfrac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\mathsf{Subst}$

▷ Let us look at a $\mathcal{H}^0$ theorem (with a proof)

▷ $\mathbf{C} \Rightarrow \mathbf{C}$ *(Tertium non datur)*

▷ Proof:

P.1 $(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (**S** with $[\mathbf{C}/P], [\mathbf{C} \Rightarrow \mathbf{C}/Q], [\mathbf{C}/R]$)
P.2 $\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}$ (**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 $\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (**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?)
□

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

$$(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R \tag{7.1}$$

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

$$(\mathbf{C} \Rightarrow Q \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow Q) \Rightarrow \mathbf{C} \Rightarrow R \tag{7.2}$$

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

$$(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow R \tag{7.3}$$

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.

7.2 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 **K** and **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 **K** and **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 3.7.11

```
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 **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.

7.3 Calculi for Propositional Logic

Let us now turn to the syntactical counterpart of the entailment relation: derivability in a calculus. Again, we take care to define the concepts at the general level of logical systems.

The intuition of a calculus is that it provides a set of syntactic rules that allow to reason by considering the form of propositions alone. Such rules are called inference rules, and they can be strung together to derivations — which can alternatively be viewed either as sequences of formulae where all formulae are justified by prior formulae or as trees of inference rule applications. But we can also define a calculus in the more general setting of logical systems as an arbitrary relation on formulae with some general properties. That allows us to abstract away from the homomorphic setup of logics and calculi and concentrate on the basics.

Derivation Systems and Inference Rules

▷ **Definition 7.3.1** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a relation $\vdash \subseteq \mathcal{P}(\mathcal{L}) \times \mathcal{L}$ a derivation relation for $\mathcal{S}$, if it

- ▷ is proof-reflexive, i.e. $\mathcal{H} \vdash \mathbf{A}$, if $\mathbf{A} \in \mathcal{H}$;
- ▷ is proof-transitive, i.e. if $\mathcal{H} \vdash \mathbf{A}$ and $\mathcal{H}' \cup \{\mathbf{A}\} \vdash \mathbf{B}$, then $\mathcal{H} \cup \mathcal{H}' \vdash \mathbf{B}$;
- ▷ admits weakening, i.e. $\mathcal{H} \vdash \mathbf{A}$ and $\mathcal{H} \subseteq \mathcal{H}'$ imply $\mathcal{H}' \vdash \mathbf{A}$.

▷ **Definition 7.3.2** We call $\langle \mathcal{L}, \mathcal{K}, \models, \vdash \rangle$ a formal system, iff $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ is a logical system, and $\vdash$ a derivation relation for $\mathcal{S}$.

▷ **Definition 7.3.3** Let $\mathcal{L}$ be a formal language, then an inference rule over $\mathcal{L}$

$$\frac{\mathbf{A}_1 \quad \cdots \quad \mathbf{A}_n}{\mathbf{C}}\mathcal{N}$$

where $\mathbf{A}_1, \ldots, \mathbf{A}_n$ and $\mathbf{C}$ are formula schemata for $\mathcal{L}$ and $\mathcal{N}$ is a name. The $\mathbf{A}_i$ are called assumptions, and $\mathbf{C}$ is called conclusion.

▷ **Definition 7.3.4** An inference rule without assumptions is called an axiom (schema).

▷ **Definition 7.3.5** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a set $\mathcal{C}$ of inference rules over $\mathcal{L}$ a calculus for $\mathcal{S}$.

With formula schemata we mean representations of sets of formulae, we use boldface uppercase letters as (meta)-variables for formulae, for instance the formula schema $\mathbf{A} \Rightarrow \mathbf{B}$ represents the set of formulae whose head is $\Rightarrow$.

Derivations and Proofs

▷ **Definition 7.3.6** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system and $\mathcal{C}$ a calculus for $\mathcal{S}$, then a $\mathcal{C}$-derivation of a formula $\mathbf{C} \in \mathcal{L}$ from a set $\mathcal{H} \subseteq \mathcal{L}$ of hypotheses (write $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{C}$) is a sequence $\mathbf{A}_1, \ldots, \mathbf{A}_m$ of $\mathcal{L}$-formulae, such that

- $\mathbf{A}_m = \mathbf{C}$, (derivation culminates in $\mathbf{C}$)
- for all $1 \leq i \leq m$, either $\mathbf{A}_i \in \mathcal{H}$, or (hypothesis)
- there is an inference rule $\dfrac{\mathbf{A}_{l_1} \quad \cdots \quad \mathbf{A}_{l_k}}{\mathbf{A}_i}$ in $\mathcal{C}$ with $l_j < i$ for all $j \leq k$. (rule application)

Observation: We can also see a derivation as a tree, where the $\mathbf{A}_{l_j}$ are the children of the node $\mathbf{A}_k$.

⋙ **Example 7.3.7** In the propositional Hilbert calculus $\mathcal{H}^0$ we have the derivation $P \vdash_{\mathcal{H}^0} Q \Rightarrow P$: the sequence is $P \Rightarrow Q \Rightarrow P, P, Q \Rightarrow P$ and the corresponding tree on the right.

$$\dfrac{\dfrac{}{P \Rightarrow Q \Rightarrow P}\,K \qquad P}{Q \Rightarrow P}\,MP$$

▷ **Observation 7.3.8** *Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system and $\mathcal{C}$ a calculus for $\mathcal{S}$, then the $\mathcal{C}$-derivation relation $\vdash_{\mathcal{D}}$ defined in Definition 7.3.6 is a derivation relation in the sense of Definition 7.3.1.*[2]

▷ **Definition 7.3.9** Correspondingly, we call $\langle \mathcal{L}, \mathcal{K}, \models, \mathcal{C} \rangle$ a formal system, iff $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ is a logical system, and $\mathcal{C}$ a calculus for $\mathcal{S}$.

▷ **Definition 7.3.10** A derivation $\emptyset \vdash_{\mathcal{C}} \mathbf{A}$ is called a proof of $\mathbf{A}$ and if one exists (write $\vdash_{\mathcal{C}} \mathbf{A}$) then $\mathbf{A}$ is called a $\mathcal{C}$-theorem.

▷ **Definition 7.3.11** 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.

[b] EDNOTE: MK: this should become a view!

Inference rules are relations on formulae represented by formula schemata (where boldface, uppercase letters are used as meta-variables for formulae). For instance, in Example 7.3.7 the inference rule $\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}$ was applied in a situation, where the meta-variables $\mathbf{A}$ and $\mathbf{B}$ were instantiated by the formulae P and $Q \Rightarrow P$.

As axioms do not have assumptions, they can be added to a derivation at any time. This is just what we did with the axioms in Example 7.3.7.

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?

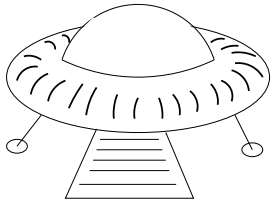
Soundness and Completeness

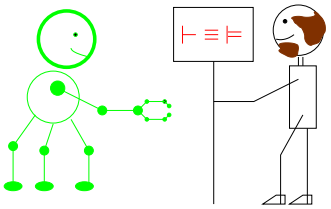
▷ **Definition 7.3.12** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a calculus $\mathcal{C}$ for $\mathcal{S}$

- ▷ sound (or correct), iff $\mathcal{H} \models \mathbf{A}$, whenever $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{A}$, and
- ▷ complete, iff $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{A}$, whenever $\mathcal{H} \models \mathbf{A}$.

▷ Goal: $\vdash \mathbf{A}$ iff $\models \mathbf{A}$ (provability and validity coincide)

- ▷ To TRUTH through PROOF (CALCULEMUS [Leibniz ∼1680])





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 be quite difficult to establish in general), then the solutions of the program will be correct, and we will find all possible ones.

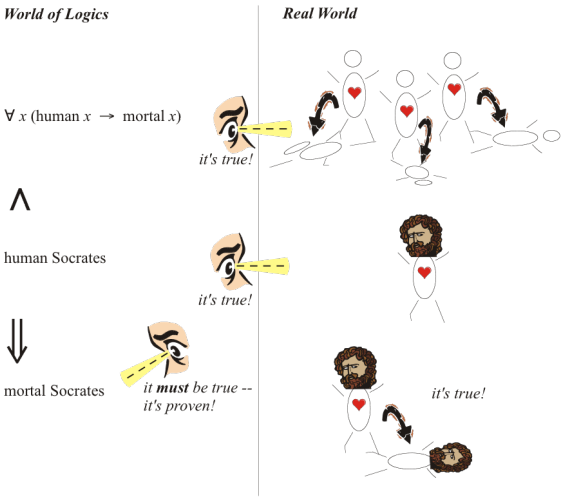
Of course, the logics we have studied so far are very simple, and not able to express interesting facts about the world, but we will study them as a simple example of the fundamental problem of Computer Science: How do the formal representations correlate with the real world.

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*.

In particular, 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.

The miracle of logics

▷ Purely formal derivations are true in the real world!



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!)

7.4 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 7.4.1** $\vdash \mathbf{A}$ *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 $\varphi: \mathcal{V}_o \to \{\mathsf{T}, \mathsf{F}\}$ 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 7.4.2** *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

□

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 7.4.3** *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. □

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 7.4.4** *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 7.4.5** *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 7.4.6** $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, *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 7.4.7** *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)

$\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}$, $(\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}$

Only one inference rule: MP.

▷ **Definition 7.4.8** $\mathcal{H}^1$ 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 7.4.9** *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 7.4.10** $\vdash_{\mathcal{H}^1} \; = \; \vdash_{\mathcal{H}^0}$

▷ 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 7.4.11** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, *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 7.4.12** *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 $\mathbf{A} \models \mathbf{C}$. □

Now, we can look at all the results so far in a single overview slide:

Properties of Calculi (Theoretical Logic)

▷ Correctness: (provable implies valid)

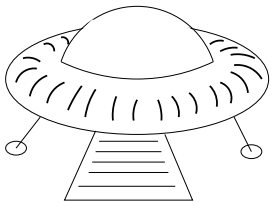
▷ $\mathcal{H} \vdash \mathbf{B}$ implies $\mathcal{H} \models \mathbf{B}$ (equivalent: $\vdash \mathbf{A}$ implies $\models \mathbf{A}$)

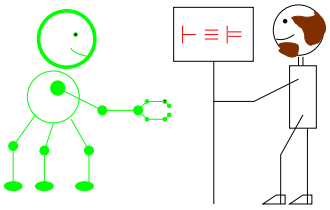
▷ Completeness: (valid implies provable)

▷ $\mathcal{H} \models \mathbf{B}$ 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])





7.5 A Calculus for Mathtalk

In our introduction to Section7.0 we have positioned Boolean expressions (and proposition logic) as a system for understanding the mathematical language "mathtalk" introduced in Section3.3. 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".

7.5.1 Propositional Natural Deduction Calculus

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 [Gen35])

▷ Idea: $\mathcal{ND}^0$ tries to mimic human theorem proving behavior (non-minimal)

▷ **Definition 7.5.1** The propositional natural deduction calculus $\mathcal{ND}^0$ has rules for the introduction and elimination of connectives

Introduction | Elimination | Axiom

$$\frac{\mathbf{A}\ \ \mathbf{B}}{\mathbf{A}\wedge\mathbf{B}}\wedge I \qquad \frac{\mathbf{A}\wedge\mathbf{B}}{\mathbf{A}}\wedge E_l \qquad \frac{\mathbf{A}\wedge\mathbf{B}}{\mathbf{B}}\wedge E_r \qquad \frac{}{\mathbf{A}\vee\neg\mathbf{A}}\mathsf{TND}$$

$$\frac{\begin{array}{c}[\mathbf{A}]^1\\ \hline\hline \mathbf{B}\end{array}}{\mathbf{A}\Rightarrow\mathbf{B}}\Rightarrow I^1 \qquad \frac{\mathbf{A}\Rightarrow\mathbf{B}\ \ \mathbf{A}}{\mathbf{B}}\Rightarrow E$$

▷ 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

$$\dfrac{\dfrac{\dfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{B}}\wedge E_r \qquad \dfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{A}}\wedge E_l}{\mathbf{B}\wedge\mathbf{A}}\wedge I}{\mathbf{A}\wedge\mathbf{B}\Rightarrow\mathbf{B}\wedge\mathbf{A}}\Rightarrow I^1$$

$$\dfrac{\dfrac{\begin{array}{c}[A]^1\\ [B]^2\end{array}}{A}}{\dfrac{A}{B\Rightarrow A}\Rightarrow I^2}\Rightarrow I^1$$

One of the nice things about the natural deduction calculus is that the deduction theorem is almost trivial to prove. In a sense, the triviality of the deduction theorem is the central idea of the calculus and the feature that makes it so natural.

A Deduction Theorem for $\mathcal{ND}^0$

▷ **Theorem 7.5.2** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$, *iff* $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

▷ Proof: We show the two directions separately

P.1 If $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$, then $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$ by $\Rightarrow I$, and

P.2 If $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$, then $\mathcal{H}, \mathcal{A} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$ by weakening and $\mathcal{H}, \mathcal{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$ by $\Rightarrow E$. □

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 7.5.1 for disjunction, negation and falsity.

More Rules for Natural Deduction

▷ **Definition 7.5.3** $\mathcal{ND}^0$ 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".[3] EdN:3

Building on the calculus $\mathcal{ND}^0$, we define a first-order calculus for "mathtalk" by providing introduction and elimination rules for the quantifiers.

[3] EdNote: give a forward reference

To obtain a first-order calculus, we have to extend $\mathcal{ND}^0$ with (introduction and elimination) rules for the quantifiers.

First-Order Natural Deduction ($\mathcal{ND}^1$; Gentzen [Gen35])

▷ Rules for propositional connectives just as always

▷ **Definition 7.5.4 (New Quantifier Rules)** The first-order natural deduction calculus $\mathcal{ND}^1$ extends $\mathcal{ND}^0$ by the following four rules

$$\frac{\mathbf{A}}{\forall X.\mathbf{A}}\forall I^* \qquad \frac{\forall X.\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\forall E$$

$$\frac{[\mathbf{B}/X](\mathbf{A})}{\exists X.\mathbf{A}}\exists I \qquad \frac{\exists X.\mathbf{A} \quad \begin{array}{c}[[c/X](\mathbf{A})]^1\\ \vdots\\ \mathbf{C}\end{array}}{\mathbf{C}}\exists E^1$$

* 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.

The only part of MathTalk we have not treated yet is equality. This comes now.

Again, we have two rules that follow the introduction/elimination pattern of natural deduction calculi.

Natural Deduction with Equality

▷ **Definition 7.5.5** We add the following two equality rules to $\mathcal{ND}^1$ to deal with equality:

$$\frac{}{\mathbf{A}=\mathbf{A}}=I \qquad \frac{\mathbf{A}=\mathbf{B} \quad \mathbf{C}\,[\mathbf{A}]_p}{[\mathbf{B}/p]\mathbf{C}}=E$$

where $\mathbf{C}\,[\mathbf{A}]_p$ if the formula $\mathbf{C}$ has a subterm $\mathbf{A}$ at position p and $[\mathbf{B}/p]\mathbf{C}$ is the result of replacing that subterm with $\mathbf{B}$.

With the $\mathcal{ND}^1$ 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) $\mathcal{ND}^1$ 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.

We will now show this on an example: the proof of the irrationality of the square root of two.

Theorem 7.5.6 $\sqrt{2}$ *is irrational*

Proof: We prove the assertion by contradiction

P.1 Assume that $\sqrt{2}$ is rational.

P.2 Then there are numbers p and q such that $\sqrt{2} = p/q$.

P.3 So we know $2s^2 = r^2$.

P.4 But $2s^2$ has an odd number of prime factors while r^2 an even number.

P.5 This is a contradiction (since they are equal), so we have proven the assertion □

If we want to formalize this into $\mathcal{ND}^1$, we have to write down all the assertions in the proof steps in MathTalk and come up with justifications for them in terms of $\mathcal{ND}^1$ inference rules. Figure 7.1 shows such a proof, where we write prime(n) is prime, use $\#(n)$ for the number of prime factors of a number n, and write irr(r) if r is irrational. Each line in Figure 7.1 represents one "step" in the proof. It consists of line number (for referencing), a formula for the asserted property, a justification via a $\mathcal{ND}^1$ rules (and the lines this one is derived from), and finally a list of line numbers of proof steps that are local hypotheses in effect for the current line. Lines 6 and 9 have the pseudo-justification "local hyp" that indicates that they are local hypotheses for the proof (they only have an implicit counterpart in the inference rules as defined above). Finally we have abbreviated the arithmetic simplification of line 9 with the justification "arith" to avoid having to formalize elementary arithmetic.

We observe that the $\mathcal{ND}^1$ proof is much more detailed, and needs quite a few Lemmata about # to go through. Furthermore, we have added a MathTalk version of the definition of irrationality (and treat definitional equality via the equality rules). Apart from these artefacts of formalization, the two representations of proofs correspond to each other very directly.

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 Section7.2 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.

#	hyp	formula	NDjust
1		$\forall n, m.\neg(2n+1) = (2m)$	lemma
2		$\forall n, m.\#(n^m) = m\#(n)$	lemma
3		$\forall n, p.\text{prime}(p) \Rightarrow \#(pn) = \#(n) + 1$	lemma
4		$\forall x.\text{irr}(x) := \neg(\exists p, q.x = p/q)$	definition
5		$\text{irr}(\sqrt{2}) = \neg(\exists p, q.\sqrt{2} = p/q)$	$\forall E(4)$
6	6	$\neg\text{irr}(\sqrt{2})$	local hyp
7	6	$\neg\neg(\exists p, q.\sqrt{2} = p/q)$	$=E(5,4)$
8	6	$\exists p, q.\sqrt{2} = p/q$	$\neg E(7)$
9	6,9	$\sqrt{2} = r/s$	local hyp
10	6,9	$2s^2 = r^2$	arith(9)
11	6,9	$\#(r^2) = 2\#(r)$	$\forall E^2(2)$
12	6,9	$\text{prime}(2) \Rightarrow \#(2s^2) = \#(s^2) + 1$	$\forall E^2(1)$
13		prime(2)	lemma
14	6,9	$\#(2s^2) = \#(s^2) + 1$	$\Rightarrow E(13,12)$
15	6,9	$\#(s^2) = 2\#(s)$	$\forall E^2(2)$
16	6,9	$\#(2s^2) = 2\#(s) + 1$	$=E(14,15)$
17		$\#(r^2) = \#(r^2)$	$=I$
18	6,9	$\#(2s^2) = \#(r^2)$	$=E(17,10)$
19	6.9	$2\#(s) + 1 = \#(r^2)$	$=E(18,16)$
20	6.9	$2\#(s) + 1 = 2\#(r)$	$=E(19,11)$
21	6.9	$\neg(2\#(s) + 1) = (2\#(r))$	$\forall E^2(1)$
22	6,9	F	$FI(20,21)$
23	6	F	$\exists E^6(22)$
24		$\neg\neg\text{irr}(\sqrt{2})$	$\neg I^6(23)$
25		$\text{irr}(\sqrt{2})$	$\neg E^2(23)$

Figure 7.1: A $\mathcal{ND}^1$ proof of the irrationality of $\sqrt{2}$

Part II

Interlude for the Semester Change

Chapter 8

Welcome Back and Administrativa

Happy new year! and Welcome Back!

- ▷ I hope you have recovered over the last 6 weeks (slept a lot)
- ▷ I hope that those of you who had problems last semester have caught up on the material (We will need much of it this year)
- ▷ I hope that you are eager to learn more about Computer Science(I certainly am!)

Your Evaluations

- ▷ Thanks: for filling out the forms (to all $14/44 \approx \frac{1}{3}$ of you!) Evaluations are a good tool for optimizing teaching/learning
- ▷ What you wrote: I have read all, will take action on some (paraphrased)
  - ▷ *Change the instructor next year!* (not your call)
  - ▷ *nice course. SML rulez! I really learned recursion* (thanks)
  - ▷ *To improve this course, I would remove its "ML part"*(let me explain,...)
  - ▷ *He doesnt' care about teaching. He simply comes unprepared to the lectures* (have you ever attended?)
  - ▷ *the slides tell simple things in very complicated ways*(this is a problem)
  - ▷ *The problem is with the workload, it is too much*(I agree, but we want to give you a chance to become Compu
  - ▷ *The Prof. is an elitist who tries to scare off all students who do not make this course their first priority* (There is General ICT I/II)
  - ▷ *More examples should be provided,* (will try do to this, you can help)
  - ▷ *give the quizzes 20% and the hw 39%,* (is this consensus?)

8.1 Recap from General CS I

Recap from GenCSI: Discrete Math and SML

- ▷ MathTalk (Rigorous communication about sets, relations,functions)
- ▷ unary natural numbers. (we have to start with something)
  - ▷ Axiomatic foundation, in particular induction (Peano Axioms)
  - ▷ constructors s, o, defined functions like $+$
- ▷ Abstract Data Types (ADT) (generalize natural numbers)
  - ▷ sorts, constructors, (defined) parameters, variables, terms, substitutions
  - ▷ define parameters by (sets of) recursive equations (rules)
  - ▷ abstract interpretation, termination,
- ▷ Programming in SML (ADT on real machines)
  - ▷ strong types, recursive functions, higher-order syntax, exceptions, ...
  - ▷ basic data types/algorithms: numbers, lists, strings,

Recap from GenCSI: Formal Languages and Boolean Algebra

- ▷ Formal Languages and Codes (models of "real" programming languages)
  - ▷ string codes, prefix codes, uniform length codes
  - ▷ formal language for unary arithmetics (onion architecture)
  - ▷ syntax and semantics (... by mapping to something we understand)
- ▷ Boolean Algebra (special syntax, semantics, ...)
  - ▷ Boolean functions vs. expressions (syntax vs. semantics again)
  - ▷ Normal forms (Boolean polynomials, clauses, CNF, DNF)
- ▷ Complexity analysis (what does it cost in the limit?)
  - ▷ Landau Notations (aka. "big-O") (function classes)
  - ▷ upper/lower bounds on costs for Boolean functions (all exponential)
- ▷ Constructing Minimal Polynomials(simpler than general minimal expressions)
  - ▷ Prime implicants, Quine McCluskey (you really liked that...)
- ▷ Propositional Logic and Theorem Proving (A simple Meta-Mathematics)

▷ Models, Calculi (Hilbert,Tableau,Resolution,ND), Soundness, Completeness

Part III

How to build Computers and the Internet (in principle)

In this part, 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 part by an investigation into the fundamental properties and limitations of computation.

Chapter 9

Combinational Circuits

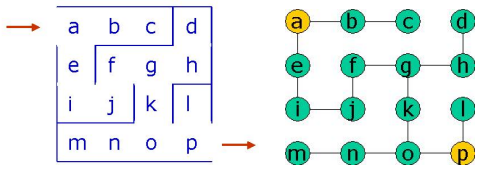
We will now study a new model of computation that comes quite close to the circuits that execute computation on today's computers. Since the course studies computation in the context of computer science, we will abstract away from all physical issues of circuits, in particular the construction of gats 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 Section9.0, and then build a simple theory of combinational circuits in Section9.1 and study their time and space complexity in Section9.2. We introduce combinational circuits for computing with numbers, by introducing positional number systems and addition in Section10.0 and covering 2s-complement numbers and subtraction in Section10.1. A basic introduction to sequential logic circuits and memory elements in Chapter10 concludes our study of circuits.

9.1 Graphs and Trees

Some more Discrete Math: Graphs and Trees

▷ Remember our Maze Example from the Intro? (long time ago)



▷ 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 combinational 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 9.1.1** 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 9.1.2** 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 9.1.3** 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.

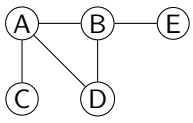
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for indeg and outdeg are the obvious variants: $\mathrm{indeg}(v) = \#(\{w \mid \{w, v\} \in E\})$ 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.

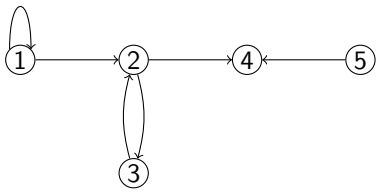
This is a very abstract yet elementary definition. We only need very basic concepts like sets and ordered pairs to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

▷ Examples

▷ **Example 9.1.4** 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 9.1.5** A directed graph $G_2 = \langle V_2, E_2\rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 2, 4\rangle, \langle 5, 4\rangle\}$



 ©: Michael Kohlhase 218

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!)

 ©: Michael Kohlhase 219

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

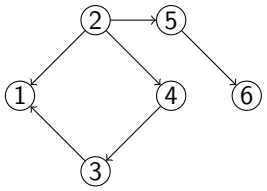
▷ **Definition 9.1.6** Let $G = \langle V, E \rangle$ be a directedgraph, then we call a node $v \in V$

  ▷ 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 9.1.7** 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.

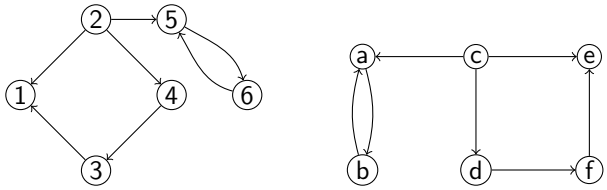
Graph Isomorphisms

▷ **Definition 9.1.8** 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
$\langle a, b\rangle \in E \Leftrightarrow \langle \psi(a), \psi(b)\rangle \in E'$	$\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in E'$

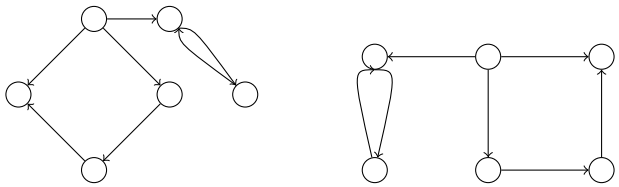
▷ **Definition 9.1.9** Two graphs G and G' are equivalent iff there is a graph-isomorphism ψ between G and G'.

▷ **Example 9.1.10** 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.

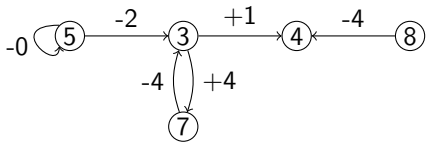
Labeled Graphs

▷ **Definition 9.1.11** A labeled graph G is a triple $\langle V, E, f\rangle$ where $\langle V, E\rangle$ is a graph and $f\colon V \cup E \to R$ is a partial function into a set R of labels.

▷ **Notation 9.1.12** 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 9.1.13** $G = \langle V, E, f\rangle$ with $V = \{A, B, C, D, E\}$, where

- ▹ $E = \{\langle A, A\rangle, \langle A, B\rangle, \langle B, C\rangle, \langle C, B\rangle, \langle B, D\rangle, \langle E, D\rangle\}$
- ▹ $f\colon V \cup E \to \{+, -, \emptyset\} \times \{1, \ldots, 9\}$ 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 9.1.14** 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 9.1.15** 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 9.1.16** 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

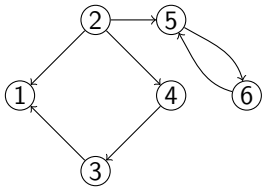
▷ **Definition 9.1.17** Given a graph $G = \langle V, E\rangle$, then

▷ 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 9.1.18** $\langle 2, 4, 3\rangle$ 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 9.1.19** 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.

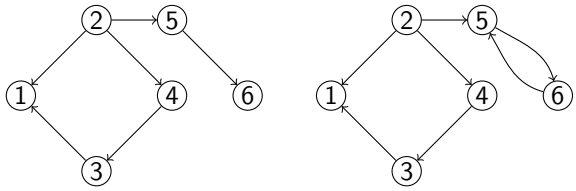
Graph Depth

▷ **Definition 9.1.20** 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 $\sup(\{\mathsf{len}(p) \mid \mathsf{indeg}(\mathsf{start}(p)) = 0 \wedge \mathsf{end}(p) = v\})$ otherwise, i.e. the length of the longest path from a source of G to v.
(⚠ can be infinite)

▷ **Definition 9.1.21** Given a digraph $G = \langle V, E\rangle$. The depth ($\mathsf{dp}(G)$) of G is defined as $\sup(\{\mathsf{len}(p) \mid p \in \Pi(G)\})$, i.e. the maximal path length in G.

▷ **Example 9.1.22** 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.

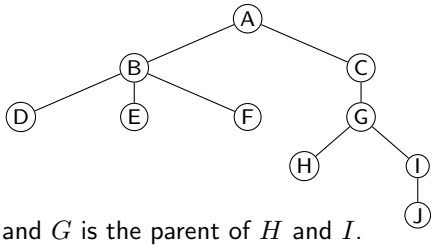
Trees

▷ **Definition 9.1.23** 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 9.1.24** 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 9.1.25** *For any node $v \in V$ except the root v_r, there is exactly one path $p \in \Pi(G)$ with $\mathsf{start}(p) = v_r$ and $\mathsf{end}(p) = v$.(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.

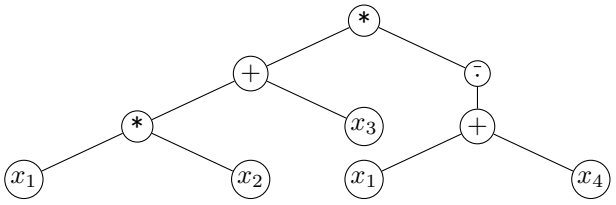
This allows us to view Boolean expressions as trees and apply all the mathematics (nomenclature and results) we will develop for them.

The Parse-Tree of a Boolean Expression

▷ **Definition 9.1.26** 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 9.1.27** the parse tree of $(x_1 * x_2 + x_3) * \overline{x_1 + x_4}$ is



9.2 Introduction to Combinatorial Circuits

We will now come to another model of computation: combinational circuits. These are models of logic circuits (physical objects made of transistors (or cathodbe 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.

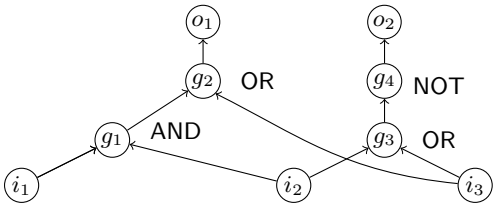
Combinational Circuits as Graphs

▷ **Definition 9.2.1** 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

- ▹ $\mathsf{indeg}(v) = 2$ and $\mathsf{outdeg}(v) = 1$ for all nodes $v \in f_g^{-1}(\{\mathsf{AND}, \mathsf{OR}\})$
- ▹ $\mathsf{indeg}(v) = \mathsf{outdeg}(v) = 1$ 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 9.2.2** The following graph $G_{cir1} = \langle V, E \rangle$ is a combinational circuit



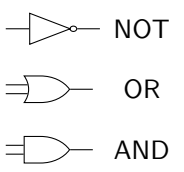
▷ **Definition 9.2.3** Add two special input nodes 0, 1 to a combinational circuit G to form a combinational circuit with constants. (will use this from now on)

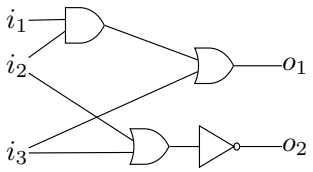
So combinational circuits are simply a class of specialized labeled directed graphs. As such, they inherit the nomenclature and equality conditions we introduced for graphs. The motivation for the restrictions is simple, we want to model computing devices based on gates, i.e. simple computational devices 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.

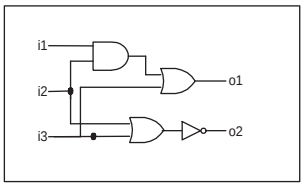
Using Special Symbols to Draw Combinational Circuits

▷ Conventional (US) Notation: The symbols for the logic gates AND, OR, and NOT.





▷ Another Visual Convention: edges with right angles (like wire wraps)



In particular, the diagram on the lower right is a visualization for the combinatoinal 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

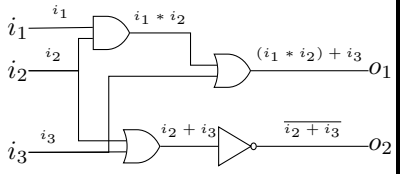
▷ Combinational Circuits and parse trees for Boolean expressions look similar

▷ Idea: Let's annotate edges in combinational circuit with Boolean Expressions!

▷ **Definition 9.2.4** 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	$v \in I(G)$
$\overline{f_L(\langle u, v\rangle)}$	$f_g(v) = \mathsf{NOT}$
$f_L(\langle u, v\rangle) * f_L(\langle u', v\rangle)$	$f_g(v) = \mathsf{AND}$
$f_L(\langle u, v\rangle) + f_L(\langle u', v\rangle)$	$f_g(v) = \mathsf{OR}$

▷ **Example 9.2.5**



Armed with the expression label of edges we can now make the computational behavior of combinational 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 9.2.6** 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 9.2.7** The circuit in Example 9.2.5 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 9.2.8** 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.

9.3 Realizing Complex Gates Efficiently

The main properties of combinational 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 combinational 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 combinational circuits that have good depth and cost. These will become important, when we build actual combinational circuits with given input/output behavior. We will now study the case for n-ary gates, since they are important building blocks for combinational circuits, see e.g. the DNF circuit on slide 238.

9.3.1 Balanced Binary Trees

We study an important class of trees now: binary trees, where all internal nodes have two children. These trees occur for instance in combinational circuits or in tree search algorithms. In many cases, we are interested in the depth of such trees, and in particular in minimizing the depth (e.g. for minimizing computation time). As we will see below, binary trees of minimal depth can be described quite easily.

Balanced Binary Trees

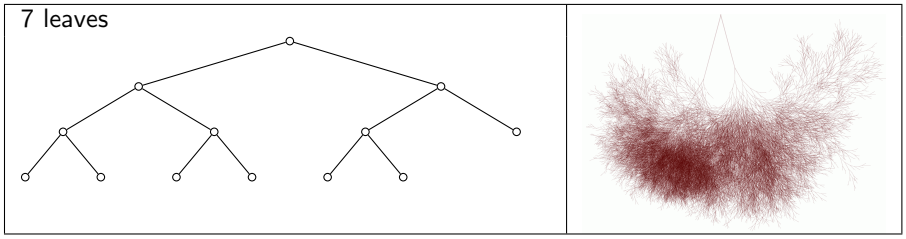
▷ **Definition 9.3.1 (Binary Tree)** A binary tree is a tree where all nodes have out-degree 2 or 0.

▷ **Definition 9.3.2** 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 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 9.3.3** Balanced binary trees become quite bushy as they grow:



We will now establish a few properties of these balanced binary trees that show that they are good building blocks for combinational circuits. In particular, we want to get at handle on the dependencies of depth, cost, and number of leaves.

Size Lemma for Balanced Trees

▷ **Lemma 9.3.4** *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 $\mathsf{dp}(G) = n > i$, 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 9.3.5** *A fully balanced tree of depth d has $2^{d+1} - 1$ nodes.*

▷ Proof Sketch: If $G := \langle V, E\rangle$ is a fully balanced tree, then $\#(V) = \sum_{i=1}^{d} 2^i = 2^{d+1} - 1$. □

Note that there is a slight subtlety in the formulation of the theorem: to get the result that the breadth of the i^{th} "row" of nodes is *exactly* 2^i, we have to restrict i by excluding the deepest "row" (which might be only partially filled in trees that are not fully balanced).

Lemma 9.3.4 shows that balanced binary trees grow in breadth very quickly, a consequence of this is that they are very shallow (and thus compute very fast), which is the essence of the next result. For an intuition, see the shape of the right tree in Example 9.3.3.

Depth Lemma for Balanced Trees

▷ **Lemma 9.3.6** *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 $d = \mathsf{dp}(G)$

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 $2 \leq c < 2$, 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 $d = \log_2(\#(V)) - \log_2(c) = \lfloor \log_2(\#(V)) \rfloor$. □

In many cases, we are interested in the ratio of leaves to nodes of balanced binary trees (e.g. when realizing n-ary gates). This result is a simple induction away.

Leaves of Binary Trees

▷ **Lemma 9.3.7** *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 combinational circuits.

9.3.2 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

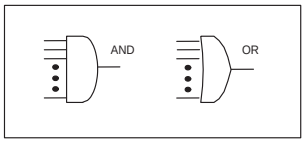
▷ Idea: Identify (and abbreviate) frequently occurring subgraphs

▷ **Definition 9.3.8** We define the n-ary gates by Boolean equivalence.[4] $\mathsf{AND}(x_1, \ldots, x_n) := 1 * \prod_{i=1}^{n} x_i$ and $\mathsf{OR}(x_1, \ldots, x_n) := 0 + \sum_{i=1}^{n} x_i$

▷ Note: These can be realized as balanced binary trees G_n

▷ **Corollary 9.3.9** $C(G_n) = n - 1$ *and* $\mathit{dp}(G_n) = \lfloor \log_2(n) \rfloor$.

▷ **Notation 9.3.10**



[d]EDNOTE: MK: what is the right way to define them? Also: we should really not use regular products here! but define special ones for boolean circuits.

Using these building blocks, we can establish a worst-case result for the depth of a combinational circuit computing a given Boolean function.

Worst Case Depth Theorem for Combinational Circuits

▷ **Theorem 9.3.11** *The worst case depth $dp(G)$ of a combinational circuit G which realizes an $k \times n$-dimensional boolean function is bounded by $dp(G) \leq n + \lfloor log_2(n) \rfloor + 1$.*

▷ 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

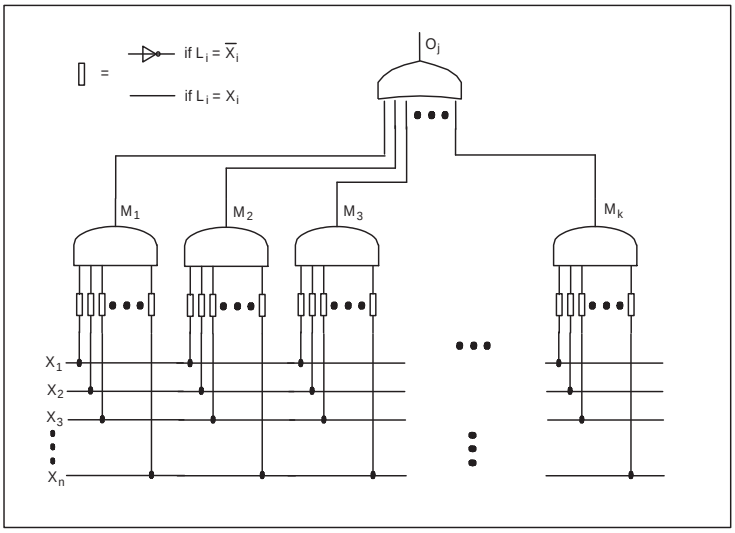
P.3 the possible negation of inputs for some literals can be done in depth 1

P.4 for each monomial the ANDs in the related circuit can be arranged in a balanced binary tree of depth $\lfloor \log_2(n) \rfloor$

P.5 there are at most 2^n monomials which can be ORed together in a balanced binary tree of depth $\lfloor \log_2(2^n) \rfloor = n$. □

Of course, the depth result is related to the first worst-case complexity result for Boolean expressions (Theorem 6.3.13); 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.

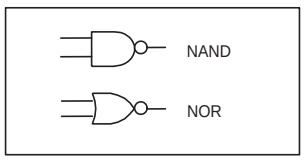
Of course, by now, we know how to do better in practice. Instead of the DNF, we can always compute the minimal polynomial for a given Boolean function using the Quine-McCluskey algorithm and derive a combinational circuit from this. While this does not give us any theoretical mileage (there are Boolean 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

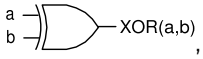
and

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 significance of the NAND and NOR gates is that we can build combinational circuite with just one type of gate, which is useful practically. On a theoretical side, we can show that both NAND and NOR are universal, i.e. they can be used to express everything we can also express with AND, OR, and NOT respectively

The Universality of NAND and NOR

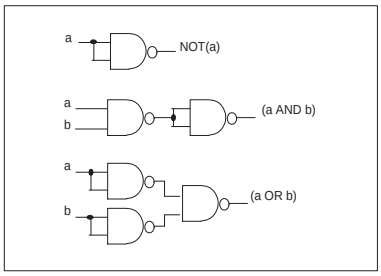
▷ **Theorem 9.3.12** *NAND and NOR are universal; i.e. any Boolean function can be expressed in terms of them.*

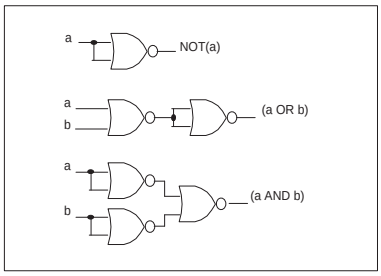
▷ Proof Sketch: Express AND, OR, and NOT via NAND and NOR respectively:

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.

Chapter 10

Arithmetic Circuits

10.1 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.

10.1.1 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 10.1.1** 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)
- ▷ $\psi_b \colon {D_b}^+ \to \{/\}^*; \langle n_k, \ldots, n_1 \rangle \mapsto \bigoplus_{i=1}^{k} \varphi_b(n_i) \odot \exp(/^{[b]}, /^{[i-1]})$ (extends φ_b to string code)

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 (cf. ?id=PNS-arithm-as?) that this give us a bijection between ${D_b}^+$ and the unary natural numbers.

Commonly Used Positional Number Systems

▷ **Example 10.1.2** 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 10.1.3** 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)

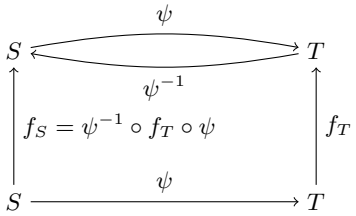
▷ ▷ $110001101011100_2 = \underbrace{0110_2}_{6_{16}}\underbrace{0011_2}_{3_{16}}\underbrace{0101_2}_{5_{16}}\underbrace{1100_2}_{C_{16}} = 635C_{16}$

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

▷ $F3A_{16} = \underbrace{F_{16}}_{1111_2}\underbrace{3_{16}}_{0011_2}\underbrace{A_{16}}_{1010_2} = 111100111010_2$, $4721_8 = \underbrace{4_8}_{100_2}\underbrace{7_8}_{111_2}\underbrace{2_8}_{010_2}\underbrace{1_8}_{001_2} = 100111010001_2$

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.



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 10.1.4** *Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS, then ψ_b is bijective.*

▷ Proof Sketch: Construct ψ_b^{-1} by successive division modulo the base of $\mathcal{N}$. □

▷ Idea: use this to define arithmetics on $\mathcal{N}$.

▷ **Definition 10.1.5** 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)

10.1.2 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 10.1.6** 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 10.1.7** An n-bit adder is a circuit computing the function $f^n_{+_2} : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^{n+1}$ 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 10.1.8** A half adder is a circuit HA implementing the function f_{HA} in the truth table on the right.

$$f_{\mathsf{HA}} : \mathbb{B}^2 \to \mathbb{B}^2 \quad \langle a, b \rangle \mapsto \langle c, s \rangle$$

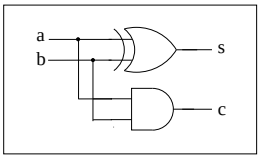
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

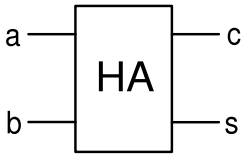
s is called the sum bit and c the carry bit.

▷ 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.

As always we are interested in the cost and depth of the circuits.

Building and Evaluating the Half-Adder





▷ So, the half-adder corresponds to the Boolean function $f_{\mathsf{HA}}: \mathbb{B}^2 \to \mathbb{B}^2; \langle a, b\rangle \mapsto \langle a \oplus b, a \wedge b\rangle$

▷ 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 $\mathsf{dp}(\mathsf{HA}) = 1$.

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

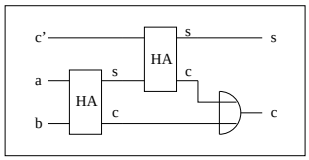
▷ **Definition 10.1.9** The 1-bit full adder is a circuit FA^1 that implements the function $f^1_{\mathsf{FA}}: \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}^2$ with $\mathsf{FA}^1(a, b, c') = B(\langle\!\langle a\rangle\!\rangle + \langle\!\langle b\rangle\!\rangle + \langle\!\langle c'\rangle\!\rangle)$

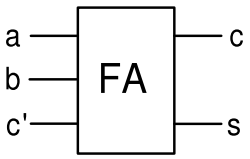
a	b	c'	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

▷ 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 10.1.10 (Cost and Depth)** $C(FA^1) = 2C(HA) + 1 = 5$ *and* $dp(FA^1) = 2dp(HA) + 1 = 3$





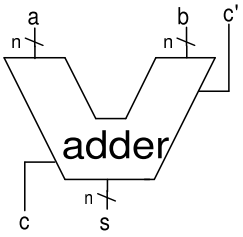
Of course adding single digits is a rather simple task, and hardly worth the effort, if this is all we can do. What we are really after, are circuits that will add n-bit binary natural numbers, so that we arrive at computer chips that can add long numbers for us.

Full n-bit Adder

▷ **Definition 10.1.11** 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 10.1.12** 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

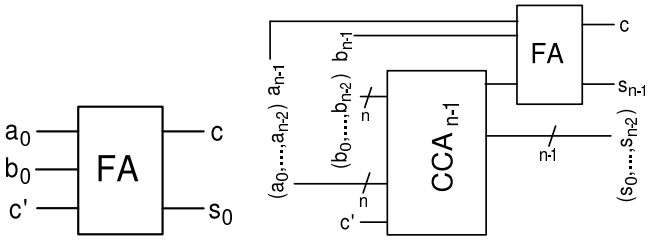
This implementation follows the intuition behind elementary school addition (only for binary numbers): we write the numbers below each other in a tabulated fashion, and from the least significant digit, we follow the process of

- 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 (Idea)

▷ 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

The Carry Chain Adder (Definition)

▷ **Definition 10.1.13** 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

▷ $\langle c, s_{n-1}\rangle = \mathsf{FA}^{n-1}(a_{n-1}, b_{n-1}, c_{n-1})$

▷ $\langle s_{n-1}, \ldots, s_0\rangle = f^{n-1}_{\mathsf{CCA}}(\langle a_{n-2}, \ldots, a_0\rangle, \langle b_{n-2}, \ldots, b_0\rangle, c')$

▷ **Lemma 10.1.14 (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 10.1.15 (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)

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					

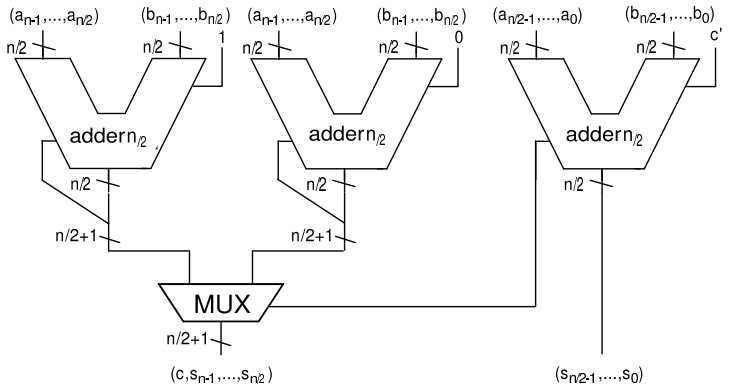
Here we start at column 10 to compute the lower sum, and at column 6 to compute two upper sums, one with carry, and one without. Once we have fully computed the lower sum, we will know about the carry in column 6, so we can simply choose which upper sum was the correct one and combine lower and upper 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 10.1.16** 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
  - ▷ $\langle c_{n/2},s_{n/2-1},\ldots,s_0\rangle = f^{n/2}_{\mathsf{CSA}}(\langle a_{n/2-1},\ldots,a_0\rangle,\langle b_{n/2-1},\ldots,b_0\rangle,c')$
  - ▷ $\langle c,s_{n-1},\ldots,s_{n/2}\rangle = \begin{cases} f^{n/2}_{\mathsf{CSA}}(\langle a_{n-1},\ldots,a_{n/2}\rangle,\langle b_{n-1},\ldots,b_{n/2}\rangle,0) & \text{if } c_{n/2}=0 \\ f^{n/2}_{\mathsf{CSA}}(\langle a_{n-1},\ldots,a_{n/2}\rangle,\langle b_{n-1},\ldots,b_{n/2}\rangle,1) & \text{if } c_{n/2}=1 \end{cases}$
  - ▷ $f^1_{\mathsf{CSA}}(a_0,b_0,c) = \mathsf{FA}^1(a_0,b_0,c)$

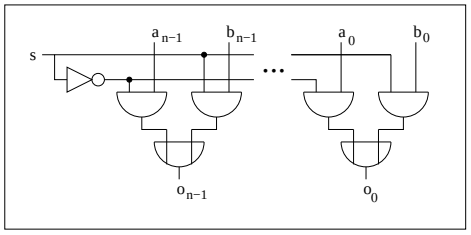
The only circuit that we still have to look at is the one that chooses the correct upper sums. Fortunately, this is a rather simple design that makes use of the classical trick that "if C, then A, else B" can be expressed as "(C and A) or (not C and B)".

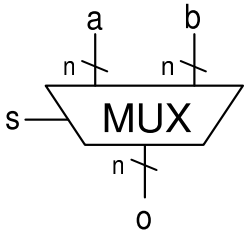
The Multiplexer

▷ **Definition 10.1.17** 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

$$f^n_{\mathsf{MUX}}(a_{n-1},\ldots,a_0,b_{n-1},\ldots,b_0,s) = \begin{cases} \langle a_{n-1},\ldots,a_0\rangle & \text{if } s=0 \\ \langle b_{n-1},\ldots,b_0\rangle & \text{if } s=1 \end{cases}$$

▷ 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

▷ $dp(CSA^n) \leq dp(CSA^{n/2}) + dp(MUX^{n/2+1})$

▷ solve the recursive equation:

$$\begin{aligned}
\mathsf{dp}(\mathsf{CSA}^n) &\leq \mathsf{dp}(\mathsf{CSA}^{n/2}) + \mathsf{dp}(\mathsf{MUX}^{n/2+1}) \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/2}) + 3 \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/4}) + 3 + 3 \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n/8}) + 3 + 3 + 3 \\
&\ldots \\
&\leq \mathsf{dp}(\mathsf{CSA}^{n2^{-i}}) + 3i \\
&\leq \mathsf{dp}(\mathsf{CSA}^1) + 3\log_2(n) \\
&\leq 3\log_2(n) + 3
\end{aligned}$$

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

▷ $C(CSA^n) = 3C(CSA^{n/2}) + C(MUX^{n/2+1})$.

▷ 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 10.1.18 (Master's Theorem for Recursive Equations)** *Given the recursively defined function $f: \mathbb{N} \to \mathbb{R}$, 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 $g: \mathbb{N} \to \mathbb{R}$, then $f(b^k) = ca^k + \sum_{i=0}^{k-1} a^i g(b^{k-i})$*

▷ We have

$$\begin{aligned} C(\mathsf{CSA}^n) &= 3C(\mathsf{CSA}^{n/2}) + C(\mathsf{MUX}^{n/2+1}) \\ &= 3C(\mathsf{CSA}^{n/2}) + 3(n/2+1) + 1 \\ &= 3C(\mathsf{CSA}^{n/2}) + \frac{3}{2}n + 4 \end{aligned}$$

▷ 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: $a^{\log_2(n)} = 2^{\log_2(a)^{\log_2(n)}} = 2^{\log_2(a)\cdot\log_2(n)} = 2^{\log_2(n)^{\log_2(a)}} = n^{\log_2(a)}$

$$\begin{aligned} C(\mathsf{CSA}^n) &= 5 \cdot 3^{\log_2(n)} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4 \\ &= 5n^{\log_2(3)} + \sum_{i=1}^{\log_2(n)} n\frac{3}{2}^i n + 4 \\ &= 5n^{\log_2(3)} + n \cdot \sum_{i=1}^{\log_2(n)} \frac{3}{2}^i + 4\log_2(n) \\ &= 5n^{\log_2(3)} + 2n \cdot \frac{3}{2}^{\log_2(n)+1} - 1 + 4\log_2(n) \\ &= 5n^{\log_2(3)} + 3n \cdot n^{\log_2(\frac{3}{2})} - 2n + 4\log_2(n) \\ &= 8n^{\log_2(3)} - 2n + 4\log_2(n) \in O(n^{\log_2(3)}) \end{aligned}$$

▷ Compare with: $C(\mathsf{CCA}^n) \in O(n)$ $\quad$ $\mathsf{dp}(\mathsf{CCA}^n) \in O(n)$

▷ So, the conditional sum adder has a smaller depth than the carry chain adder. This smaller depth is paid with higher cost.

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.

10.2 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 10.2.1 ($(n+1)$-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 10.2.2** 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!

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 10.2.3** 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 10.2.4** 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

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., $b_n = 0 \Leftrightarrow (\langle\!\langle b \rangle\!\rangle^{2s}_n \geq 0)$.

▷ Negative numbers have a sign bit 1, i.e., $b_n = 1 \Leftrightarrow \langle\!\langle b \rangle\!\rangle^{2s}_n < 0$.

▷ For positive numbers, the two's complement representation corresponds to the normal binary number representation, i.e., $b_n = 0 \Leftrightarrow \langle\!\langle b \rangle\!\rangle^{2s}_n = \langle\!\langle b \rangle\!\rangle$

▷ There is a unique representation of the number zero in the n-bit two's complement system, namely $B^{2s}_n(0) = \langle 0, \ldots, 0 \rangle$.

▷ This number system has an asymmetric range $\mathcal{R}^{2s}{}_n := \{-2^n, \ldots, 2^n - 1\}$.

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 10.2.5** *Let $a \in \mathbb{B}^{n+1}$ be a binary string, then $-\langle\!\langle a \rangle\!\rangle^{2s}_n = \langle\!\langle \overline{a} \rangle\!\rangle^{2s}_n + 1$, where $\overline{a}$ is the pointwise bit complement of a.*

▷ Proof Sketch: By calculation using the definitions:

$$\begin{aligned}
\langle\!\langle \overline{a_n}, \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle^{2s}_n &= -\overline{a_n} \cdot 2^n + \langle\!\langle \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle \\
&= \overline{a_n} \cdot -2^n + \sum_{i=0}^{n-1} \overline{a_i} \cdot 2^i \\
&= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 1 - a_i \cdot 2^i \\
&= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= -2^n + a_n \cdot 2^n + 2^{n-1} - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle \\
&= (-2^n + 2^n) + a_n \cdot 2^n - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle - 1 \\
&= -(a_n \cdot -2^n + \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle) - 1 \\
&= -\langle\!\langle a \rangle\!\rangle^{2s}_n - 1
\end{aligned}$$

□

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 10.2.6** 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

$$f^n_{\mathsf{SUB}}(a, b, b') = B^{2s}_n(\langle\!\langle a \rangle\!\rangle^{2s}_n - \langle\!\langle b \rangle\!\rangle^{2s}_n - b')$$

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 $\langle\!\langle a \rangle\!\rangle^{2s}_n - \langle\!\langle b \rangle\!\rangle^{2s}_n = \langle\!\langle a \rangle\!\rangle^{2s}_n + (-\langle\!\langle b \rangle\!\rangle^{2s}_n) = \langle\!\langle a \rangle\!\rangle^{2s}_n + \langle\!\langle \overline{b} \rangle\!\rangle^{2s}_n + 1$

▷ 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 10.2.7** 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 $f^n_{\mathsf{TCA}}(a, b, c') = B^{2s}_n(\langle\!\langle a \rangle\!\rangle^{2s}_n + \langle\!\langle b \rangle\!\rangle^{2s}_n + c')$ for all $a, b \in \mathbb{B}^n$ and $c' \in \mathbb{B}$.

▷ **Theorem 10.2.8** $f^n_{TCA} = f^n_{FA}$ *(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 10.2.9** *Let $a = \langle a_n, \ldots, a_0\rangle \in \mathbb{B}^{n+1}$ be a binary string, then $\langle\!\langle a_n, \ldots, a_0\rangle\!\rangle^{2s}_{n+1} = \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle^{2s}_{n}$.*

▷ Proof Sketch: By calculation:

$$\begin{aligned}
\langle\!\langle a_n, \ldots, a_0\rangle\!\rangle^{2s}_{n+1} &= -a_n \cdot 2^{n+1} + \langle\!\langle a_n, \ldots, a_0\rangle\!\rangle \\
&= -a_n \cdot 2^{n+1} + a_n \cdot 2^n + \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle \\
&= a_n \cdot (-2^{n+1} + 2^n) + \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle \\
&= a_n \cdot (-2 \cdot 2^n + 2^n) + \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle \\
&= -a_n \cdot 2^n + \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle \\
&= \langle\!\langle a_{n-1}, \ldots, a_0\rangle\!\rangle^{2s}_{n}
\end{aligned}$$

□

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 10.2.10** 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 $(\mathsf{ic}_k(a,b,c))$, the k-th intermediate carry of a, b, and c, iff

$$\langle\!\langle \mathsf{ic}_k(a,b,c), s_{k-1}, \ldots, s_0\rangle\!\rangle = \langle\!\langle a_{k-1}, \ldots, a_0\rangle\!\rangle + \langle\!\langle b_{k-1}, \ldots, b_0\rangle\!\rangle + c$$

for some $s_i \in \mathbb{B}$.

▷ **Theorem 10.2.11** *Let $a, b \in \mathbb{B}^n$ and $c \in \mathbb{B}$, then*

1. *$\langle\!\langle a\rangle\!\rangle^{2s}_n + \langle\!\langle b\rangle\!\rangle^{2s}_n + c \in \mathcal{R}^{2s}_n$, iff $(ic_{n+1}(a,b,c)) = (ic_n(a,b,c))$.*
2. *If $(ic_{n+1}(a,b,c)) = (ic_n(a,b,c))$, then $\langle\!\langle a\rangle\!\rangle^{2s}_n + \langle\!\langle b\rangle\!\rangle^{2s}_n + c = \langle\!\langle s\rangle\!\rangle^{2s}_n$, where $\langle\!\langle ic_{n+1}(a,b,c), s_n, \ldots, s_0\rangle\!\rangle = \langle\!\langle a\rangle\!\rangle + \langle\!\langle b\rangle\!\rangle + c$.*

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

$$\begin{aligned} \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c &= \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle + \langle\!\langle b_{n-1}, \ldots, b_0 \rangle\!\rangle + c \\ &= \langle\!\langle \mathsf{ic}_n(a,b,c), s_{n-1}, \ldots, s_0 \rangle\!\rangle \end{aligned}$$

and $a_n + b_n + (\mathsf{ic}_n(a,b,c)) = \langle\!\langle \mathsf{ic}_{n+1}(a,b,c), s_n \rangle\!\rangle$.

P.2 We have to consider three cases

P.2.1 $a_n = b_n = 0$:

P.2.1.1 $\langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}}$ and $\langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}}$ are both positive, so $(\mathsf{ic}_{n+1}(a,b,c)) = 0$ and furthermore

$$\begin{aligned} (\mathsf{ic}_n(a,b,c)) = 0 &\Leftrightarrow \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow \langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}} + \langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}} + c \leq 2^n - 1 \end{aligned}$$

P.2.1.2 Hence,

$$\begin{aligned} \langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}} + \langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}} + c &= \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \\ &= \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= \langle\!\langle 0, s_{n-1}, \ldots, s_0 \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle_n^{2\mathsf{s}} \end{aligned}$$

□

P.2.2 $a_n = b_n = 1$:

P.2.2.1 $\langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}}$ and $\langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}}$ 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 $\langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}} + \langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}} + c = -2^{n+1} + \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \geq -2^n$

P.2.2.2 Hence,

$$\begin{aligned} \langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}} + \langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}} + c &= -2^n + \langle\!\langle a' \rangle\!\rangle + -2^n + \langle\!\langle b' \rangle\!\rangle + c \\ &= -2^{n+1} + \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c \\ &= -2^{n+1} + \langle\!\langle 1, s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= -2^n + \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle \\ &= \langle\!\langle s \rangle\!\rangle_n^{2\mathsf{s}} \end{aligned}$$

□

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 $\langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}}$ and $\langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}}$ is in the admissible range $\mathcal{R}^{2\mathsf{s}}{}_n$ as

$$\langle\!\langle a \rangle\!\rangle_n^{2\mathsf{s}} + \langle\!\langle b \rangle\!\rangle_n^{2\mathsf{s}} + c = \langle\!\langle a' \rangle\!\rangle + \langle\!\langle b' \rangle\!\rangle + c - 2^n$$

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 $s_n = a_n \oplus b_n \oplus (\mathsf{ic}_n(a,b,c)) = 1 \oplus (\mathsf{ic}_n(a,b,c)) = \overline{\mathsf{ic}_n(a,b,c)}$. □

P.3 Thus we have considered all the cases and completed the proof. □

The Main Theorem for TCN again

▷ 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.

10.3 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

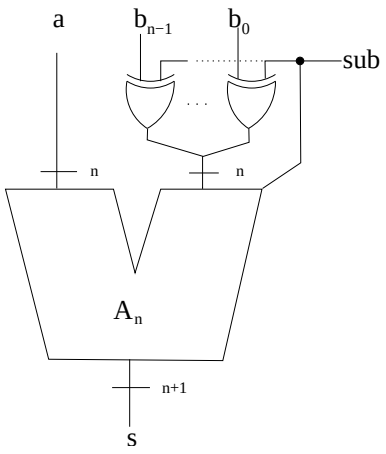
▷ Idea: Build a Combinational Circuit that can add and subtract ($\mathrm{sub} = 1 \leadsto$ subtract)

▷ If $\mathrm{sub} = 0$, then the circuit acts like an adder ($a \oplus 0 = a$)

▷ If $\mathrm{sub} = 1$, let $S := \langle\!\langle a \rangle\!\rangle^{2s}_n + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle^{2s}_n + 1$ ($a \oplus 0 = 1 - a$)

▷ For $s \in \mathcal{R}^{2s}{}_n$ the TCN main theorem and the TCN structure theorem together guarantee

$$\begin{aligned} s &= \langle\!\langle a \rangle\!\rangle^{2s}_n + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle^{2s}_n + 1 \\ &= \langle\!\langle a \rangle\!\rangle^{2s}_n - \langle\!\langle b \rangle\!\rangle^{2s}_n - 1 + 1 \end{aligned}$$



▷ 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,...)

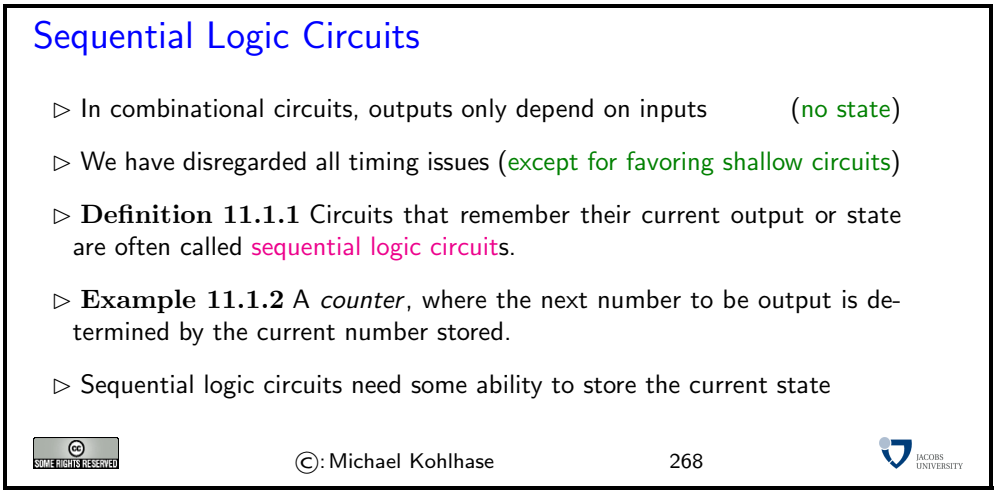
In fact extended variants of the very simple Add/Subtract unit are at the heart of any computer. These are called arithmetic logic units.

Chapter 11

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 chapter we 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.

11.1 Sequential Logic Circuits

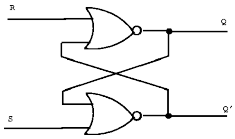


Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits. We will investigate basic circuits that have the ability to store bits of data. We will start with the simplest possible memory element, and develop more elaborate 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 11.1.3** 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.

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:

↓	T	F
0	1	0
1	0	0

$S = 1$ **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.

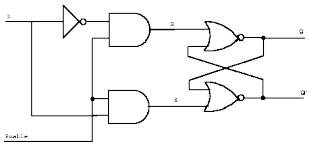
The D-Flipflop: the simplest memory device

▷ 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 11.1.4** 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.

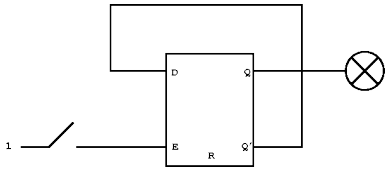
▷ When $E = 1$ the value of D determines the value of the output Q, when E returns to 0, the most recent input D is "remembered."

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 11.1.5 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.

11.2 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.

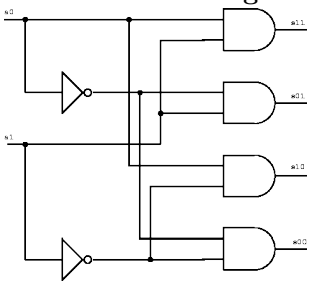
Address Decoder Logic

▷ 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 11.2.1** 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 11.2.2 (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

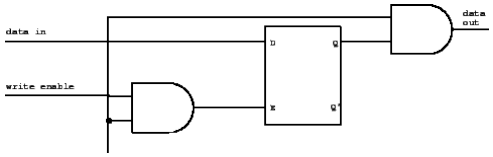
▷ Idea (Input): Use a D-flipflop connect its E input to the ADL output. Connect the D-input to the common RAM data input line.(input only if addressed)

▷ 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 11.2.3** 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

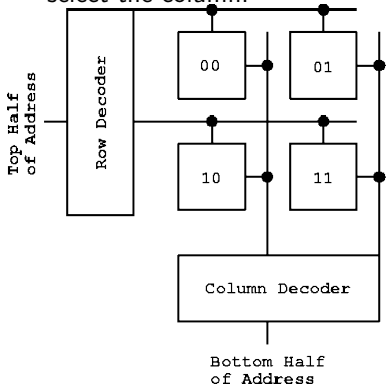
▷ To take advantage of the two-dimensional nature of the chip, storage elements are arranged on a square grid.(columns and rows of storage elements)

▷ 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.



Now that we have seen how to build memory chips, we should learn how to talk about their sizes.

11.3 Units of Information

We introduce the units for information and try to get an intuition about what we can store in memory.

Units for Information

▷ Observation: The smallest unit of information is knowing the state of a system with only two states.

▷ **Definition 11.3.1** A bit (a contraction of "binary digit") is the basic unit of capacity of a data storage device or communication communication channel. The capacity of a system which can exist in only two states, is one bit (written as 1 b)

▷ Note: In the ASCII encoding, one character is encoded as 8 b, so we introduce another basic unit:

▷ **Definition 11.3.2** The byte is a derived unit for information capacity: 1 B = 8 b.

From the basic units of information, we can make prefixed units for prefixed units for larger chunks of information. But note that the usual SI unit prefixes are inconvenient for application to information measures, since powers of two are much more natural to realize (recall the discussion on balanceds).

Larger Units of Information via Binary Prefixes

▷ We will see that memory comes naturally in powers to 2, as we address memory cells by binary numbers, therefore the derived information units are prefixed by special prefixes that are based on powers of 2.

▷ **Definition 11.3.3 (Binary Prefixes)** The following binary unit prefixes are used for information units because they are similar to the SI unit prefixes.

prefix	symbol	2^n	decimal	˜SI prefix	Symbol
kibi	Ki	2^{10}	1024	kilo	k
mebi	Mi	2^{20}	1048576	mega	M
gibi	Gi	2^{30}	1.074×10^9	giga	G
tebi	Ti	2^{40}	1.1×10^{12}	tera	T
pebi	Pi	2^{50}	1.125×10^{15}	peta	P
exbi	Ei	2^{60}	1.153×10^{18}	exa	E
zebi	Zi	2^{70}	1.181×10^{21}	zetta	Z
yobi	Yi	2^{80}	1.209×10^{24}	yotta	Y

Note: The correspondence works better on the smaller prefixes; for yobi vs. yotta there is a 20% difference in magnitude.

▷▷ The SI unit prefixes (and their operators) are often used instead of the correct binary ones defined here.

▷ **Example 11.3.4** You can buy hard-disks that say that their capacity is "one tera-byte", but they actually have a capacity of one tebibyte.

Let us now look at some information quantities and their real-world counterparts to get an intuition for the information content.

How much Information?

Bit (b)	*binary digit 0/1*
Byte (B)	*8 bit*
2 Bytes	A Unicode character in UTF.
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 Climate 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 (ZB)	*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.

The information in this table is compiled from various studies, most recently [HL11].

Note: Information content of real-world artifacts can be assessed differently, depending on the view. Consider for instance a text typewritten on a single page. According to our definition, this has ca. 2 kB, but if we fax it, the image of the page has 2 MB or more, and a recording of a text read out loud is ca. 50 MB. Whether this is a terrible waste of bandwidth depends on the application. On a fax, we can use the shape of the signature for identification (here we actually care more about the shape of the ink mark than the letters it encodes) or can see the shape of a coffee stain. In the audio recording we can hear the inflections and sentence melodies to gain an impression on the emotions that come with text.

Chapter 12

Computing Devices and Programming Languages

The main focus of this chapter is 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 $\mathcal{L}(\texttt{VM})$ and write a $\mathcal{L}(\texttt{VM})$-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 $\mathcal{L}(\texttt{VM})$ 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 $\mathcal{L}(\texttt{VM})$ for it.

We conclude this chapter by 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 chapter is inspired by and adapted from Gert Smolka excellent introduction to Computer Science based on SML [Smo11].

12.1 How to Build and Program a Computer (in Principle)

In this section, we will combine the arithmetic/logical units from Chapter9 with the storage elements (RAM) from Section11.1 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) (We can get by with $n = 4$)
- ▷ 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 $D\colon \mathbb{N} \to \mathbb{B}^n$)

▷ add a memory management unit (MMU)(move values between RAM and registers)

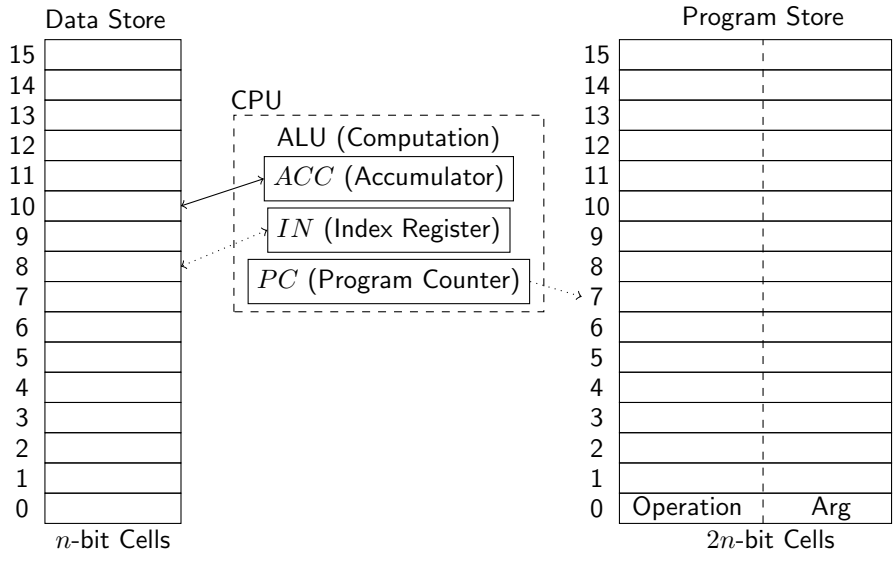
▷ program it in assembler language (lowest level of programming)

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 program store 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 $D\colon \mathbb{N} \to \mathbb{B}^n$.

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.

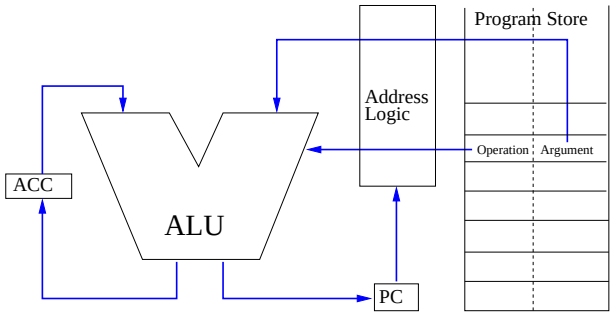
Memory Plan of a Register Machine



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



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 15 operations, so we need to set $n \geq 4$. Note that $n = 4$ also means that we only have $2^4 = 16$ cells in the program and data stores. Which is a serious restriction. Realistic 4-bit architectures therefore make the registers $2n = 8$-bit wide. For the purposes of this course, we will gloss over this.

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 12.1.1** assembler language (ASM)as mnemonic encoding of n-bit binary codes.

instruction	effect	PC	comment
`LOAD` i	$ACC := D(i)$	$PC := PC+1$	load data
`STORE` i	$D(i) := ACC$	$PC := PC+1$	store data
`ADD` i	$ACC := ACC+D(i)$	$PC := PC+1$	add to ACC
`SUB` i	$ACC := ACC-D(i)$	$PC := PC+1$	subtract from ACC
`LOADI` i	$ACC := i$	$PC := PC+1$	load number
`ADDI` i	$ACC := ACC+i$	$PC := PC+1$	add number

Definition 12.1.2 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 12.1.3 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 12.1.4** Exchange the values of cells 0 and 1 in the data store

P	instruction	comment
0	`LOAD` 0	$ACC := D(0) = x$
1	`STORE` 2	$D(2) := ACC = x$
2	`LOAD` 1	$ACC := D(1) = y$
3	`STORE` 0	$D(0) := ACC = y$
4	`LOAD` 2	$ACC := D(2) = x$
5	`STORE` 1	$D(1) := ACC = x$

▷ **Example 12.1.5** Let $D(1) = a$, $D(2) = b$, and $D(3) = c$, store $a+b+c$ in data cell 4

P	instruction	comment
0	`LOAD` 1	$ACC := D(1) = a$
1	`ADD` 2	$ACC := ACC+D(2) = a+b$
2	`ADD` 3	$ACC := ACC+D(3) = a+b+c$
3	`STORE` 4	$D(4) := ACC = a+b+c$

▷ use `LOADI` i, `ADDI` i, and `ADDI` $-i$ 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 $D(0) = x$ and $D(1) = y$, how to we store y into cell x of the data store? (impossible, as we have only absolute addressing)

▷ **Definition 12.1.6** Introduce an index register IN and register instructions

instruction	effect	PC	comment
`RLOAD` i	$ACC := D(IN)$	$PC := PC+1$	relative load
`RSTORE` i	$D(IN) := ACC$	$PC := PC+1$	relative store
`MVAI` i	$IN := ACC$	$PC := PC+1$	move value
`MVIA` i	$ACC := IN$	$PC := PC+1$	move value

Problem Solution:

	P	instruction	comment
▷	0	`LOAD` 0	$ACC := D(0) = x$
	1	`MVAI` 0	$IN := ACC = x$
	2	`LOAD` 1	$ACC := D(1) = y$
	3	`RSTORE` 0	$D(x) = D(IN) := ACC = y$

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 on slide 283, but we will not worry about this).

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). Real assembler languages give themselves a set of comparison relations (e.g. $=$ and $<$) they 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 12.1.7**

instruction	effect	PC	comment
`JUMP` i		$PC := PC + i$	jump forward i steps
`JUMP`$_=$ i		$PC := \begin{cases} PC+i & \text{if } ACC = 0 \\ PC+1 & \text{else} \end{cases}$	conditional jump

▷ **Definition 12.1.8 (Two more)**

instruction	effect	PC	comment
`NOP` i		$PC := PC + 1$	no operation
`STOP` i			stop computation

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 12.1.9** Let $D(0) = n$, $D(1) = a$, and $D(2) = b$, copy the values of cells $a, \ldots, a+n-1$ to cells $b, \ldots, b+n-1$, while $a, b \geq 4$ and $|a-b| \geq n$.

P	instruction	comment	P	instruction	comment
0	`LOAD` 0	$ACC := i$ $(i = n)$	9	`MVAI` 0	$IN := b + i$
1	`JUMP`$_=$ 15	if $i = 0$ then stop	10	`LOAD` 4	$ACC := D(a+i)$
2	`LOAD` 1	$ACC := a$	11	`RSTORE` 0	$D(b+i) := D(a+i)$
3	`ADD` 0	$ACC := a + i$	12	`LOAD` 0	$ACC := i$
4	`MVAI` 0	$IN := a + i$	13	`ADDI` -1	$ACC := ACC - 1$
5	`RLOAD` 0	$ACC := D(a+i)$	14	`STORE` 0	$D(0) := i - 1$
6	`STORE` 4	$D(4) := D(a+i)$	15	`JUMP` -14	goto step 1
7	`LOAD` 2	$ACC := b$	16	`STOP` 0	Stop
8	`ADD` 0	$ACC := b + i$			

▷ **Lemma 12.1.10** *We have* $D(0) = n - (i-1)$, $IN = a + i - 1$, *and* $IN = b + i - 1$ *for all* $1 \leq i \leq n+1$. *(the program does what we want)*

▷ proof by induction on n.

▷ **Definition 12.1.11** The induction hypotheses are called loop invariants.

12.2 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 12.2.1 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 12.2.2** An ASM program VM is called a virtual machine for $\mathcal{L}(\texttt{VM})$, iff VM inputs a $\mathcal{L}(\texttt{VM})$ 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 $\mathcal{L}(\texttt{VM})$.(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.

12.2.1 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 12.2.3** 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 12.2.4** The $\mathcal{L}(\mathtt{VM})$ program "`con` 4 `con` 7 `add`" pushes $7+4=11$ to the stack.

▷ **Example 12.2.5** 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 12.2.6** 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 12.2.7** 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 12.2.8** 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 12.2.9** 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 12.2.10** Consider the following program that computes (12)! and the corresponding $\mathcal{L}(\mathtt{VM})$ program:

```
var n := 12; var a := 1;
while 2 <= n do (
  a := a * n;
  n := n - 1;
)
return a;
```

```
con 12 con 1
peek 0 con 2 leq cjp 18
peek 0 peek 1 mul poke 1
con 1 peek 0 sub poke 0
jp −21
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 $\mathcal{L}(\texttt{VM})$). We will use the example above as guiding intuition for a compiler from a simple imperative language to $\mathcal{L}(\texttt{VM})$ byte code below. But first we build a virtual machine for $\mathcal{L}(\texttt{VM})$.

12.3 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 language in the next step. We will write this compiler in SML, since we are most familiar with this[1]

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 12.3.1** 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⟫

To make the concepts involved concrete, we look at a concrete example.

Example: An SW Program for 12 Factorial

▷ **Example 12.3.2 (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
```

[1] Note that this is a standard procedure called cross-compiling. We just assume that we already have a computer that runs SML. Note that only if we are creating the first compiler for the first computer, we have to write it in assembler language.

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.

Recall that we want to builds a compiler for SW in SML. To simplify this task, we skip the lexical analysis phase of a compiler that converts a SW string into an abstract syntax tree, and start directly with an SML expression.

The following slide presents the SML data types for SW programs.

Abstract Syntax of SW

▷ **Definition 12.3.3** `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
```


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 12.3.4 (Abstract SW Syntax)** We apply the abstract syntax

to the SW program from Example 12.3.2:

```
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.

Similarly, we do not want to get into the code generation stage of a compiler that generates a $\mathcal{L}(\texttt{VM})$ program in concrete syntax. Instead we directly generate abstract SML expression.

We need an SML data type for $\mathcal{L}(\texttt{VM})$ programs, and an auxiliary function `wlen` that counts the numbers of instructions in a $\mathcal{L}(\texttt{VM})$ program (a list of instructions $\mathcal{L}(\texttt{VM})$ instructions). Fortunately, this is very simple.

An SML Data Type for $\mathcal{L}(\texttt{VM})$ Programs

```
type index = int          (* index in the environment *)
type noi   = int          (* number of instructions *)

datatype instruction =
    con     of int
  | add | sub | mul     (* addition, subtraction, ... *)
  | 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
```

We have introduced a couple of types only for documentation purposes. We want to keep apart the integers we use as index in the environment to those we use as numbers of instructions to jump over in the jump instructions.

Before we can come to the implementation of the compiler, we will more infrastructure. Recall that we needed to keep track of which variable names corresponded to which stack position in slide 299, in our SW compiler, we will have to do the same. There is a standard data structure for this.

Needed Infrastructure: Environments

▷ Need a structure to keep track of the values of declared identifiers.(take shadowing into account)

▷ **Definition 12.3.5** 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 ($\rightsquigarrow$ the empty function)
- ▹ insertion of a key/value pair $\langle k, v\rangle$ into an environment φ:($\rightsquigarrow \varphi, [v/k]$)
- ▹ lookup of the value v for a key k in φ ($\rightsquigarrow \varphi(k)$)

▷ 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
```


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 $\mathcal{L}(\mathtt{VM})$ instructions. From there, we only need to apply a simple conversion (which we omit) to numbers to obtain $\mathcal{L}(\mathtt{VM})$ byte code.

Compiling SW programs

▷ SML function from SW programs (type program) to $\mathcal{L}(\mathtt{VM})$ 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
```


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]
```


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 $\mathcal{L}(\mathtt{VM})$ 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
```


As we anticipated above, the `compileD` function is more complex than the other two. It gives $\mathcal{L}(\mathtt{VM})$ 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
```


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 $\mathcal{L}(\mathtt{VM})$ we discussed above, we can run SW programs on the register machine REMA.

If we had a REMA simulator, then we could run SW programs on our computers outright.

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 $\mathcal{L}(\mathtt{VM})$ with new operations next.

Note that the compiler we have seen above produces $\mathcal{L}(\mathtt{VM})$ 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 12.3.6** 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)

12.4 Basic Functional Programs

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.

12.4.1 A Bare-Bones Language

We will now define a minimal core of the functional programming language SML, which we will call μML. It has all the characteristics of a functional programming language: functional variables and named functions, and lacks imperative features like variable assignment or statements.

As a bare-bone functional programming language, μML is not far off from the first incarnations of LISP – we would only need to add lists as primary data type; which we will not to keep things simple.

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 12.4.1** 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)

To make the concepts involved concrete, we look at a concrete example: the procedure on the next slide computes 10^2.

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 12.4.2 (Computing Ten Squared)**

```
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(y,2)                   (* return value *)
end                          (* end program *)
```


Note that in our example, we have declared two values: a function `exp` (with arguments `x` and `n`) and `y` (without arguments). Both are used in the return expression.

12.4.2 A Virtual Machine with Procedures

We will now extend the virtual machine by four instructions that allow to represent procedures with arbitrary numbers of arguments.

Adding Instructions for Procedures to $\mathcal{L}(\texttt{VM})$

▷ **Definition 12.4.3** We obtain the language $\mathcal{L}(\texttt{VMP})$ by adding the following four commands to $\mathcal{L}(\texttt{VM})$:

- ▷ `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 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 12.4.2 to $\mathcal{L}(\texttt{VMP})$. We have indicated corresponding parts of the programs by putting them on the same line.

A μML Program and its $\mathcal{L}(\texttt{VMP})$ Translation

▷ **Example 12.4.4 (A μML Program and its $\mathcal{L}(\texttt{VMP})$ Translation)**

```
[                              let
 proc 2 26,                     fun exp(x,n) =
 con 0, arg 2, leq, cjp 5,        if n<=0
 con 1, return,                   then 1
 con 1, arg 2, sub, arg 1,        else x*exp(x,n-1)
 call 0, arg 1, mul, return,
 proc 0 6, con 10, return,       val y 10
                                in
 con 2, call 26, call 0,          exp(y,2)
 halt]                          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 12.4.5 We go through the $\mathcal{L}(\texttt{VMP})$ code step by step and trace the

state of 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,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```

empty stack

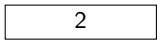
`proc` jumps over the body of the procedure declaration(with the help of its second argument)

```
▷ [proc 2 26,
   con 0, arg 2, leq, cjp 5,
   con 1, return,
   con 1, arg 2, sub, arg 1,
   call 0, arg 1, mul, return,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```

empty stack

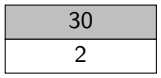
again, `proc` jumps over the body of the procedure declaration

```
▷ [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,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```



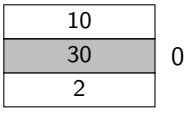
We push the argument onto the stack

```
▷ [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,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```



- ▷ `call` pushes the return address (of the `call` statement in the $\mathcal{L}(\texttt{VM})$ program)
- ▷ then it jumps to the first body instruction.

```
[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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```



push the arguent to the stack

```
[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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

10
2

- ▷ `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.

```
[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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

2	
0	
38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

0	
38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

2	
1	
38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

1	
38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

10	
1	
38	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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

22	0
10	-1
1	-2
38	
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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

1	
0	
22	0
10	-1
1	-2
38	
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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

22	0
10	-1
1	-2
38	
10	
2	

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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

1	
1	
22	0
10	-1
1	-2
38	
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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

10	
0	
22	0
10	-1
1	-2
38	
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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

22	0
10	-1
0	-2
22	
10	
1	
38	
10	
2	

`call` pushes the return address and moves the current frame up

```
[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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

0	
0	
22	0
10	-1
0	-2
22	
10	
1	
38	
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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

22	0
10	-1
0	-2
22	
10	
1	
38	
10	
2	

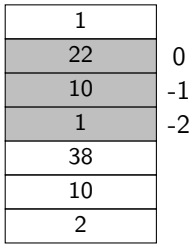
`leq` compares the top two numbers, `cjp` pops the result and does not jump.

```
[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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```

1	
22	0
10	-1
0	-2
22	
10	
1	
38	
10	
2	

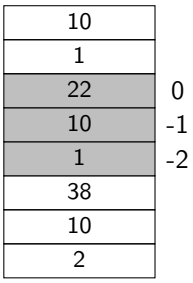
we push the result value 1

```
▷ [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,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```



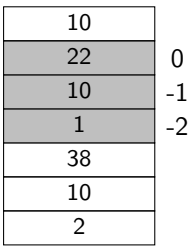
- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address memorized right below the top of the stack,
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack.

```
  [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,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```



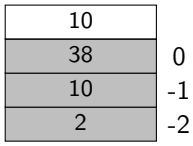
`arg` pushes the first argument from the (new) current frame

```
▷▷ [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,
    proc 0 6, con 10, return,
    con 2, call 26, call 0,
    halt]
```



`mul` multiplies, pops the arguments and pushes the result.

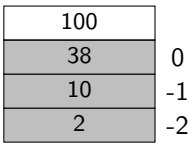
```
▷ [proc 2 26,
   con 0, arg 2, leq, cjp 5,
   con 1, return,
   con 1, arg 2, sub, arg 1,
   call 0, arg 1, mul, return,
   proc 0 6, con 10, return,
   con 2, call 26, call 0,
   halt]
```



- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address,
- ▷ deletes the current frame

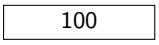
▷ and puts the result back on top of the remaining stack.

```
[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, call 26, call 0,
halt]
```



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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```



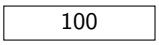
▷ `return` interprets the top of the stack as the result,

▷ it jumps to the return address (38 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,
 proc 0 6, con 10, return,
 con 2, call 26, call 0,
 halt]
```



we are finally done; the result is on the top of the stack. Note that the stack below has not changed.

Time for a recap, to see what we have learned from the example.

▷ What have we seen?

▷ The four new $\mathcal{L}(\texttt{VMP})$ instructions 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

`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.

▷ the `VMP` stack is dual-purpose: it stores (very elegant design)

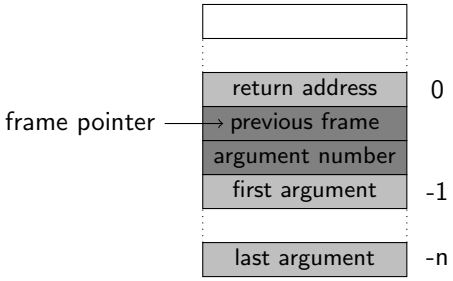
- ▷ intermediate results of (arithmetic) computation (as in `VM`)
- ▷ frames for the arguments of (static) procedures(e.g. for recursive computation)

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.

▷ **Definition 12.4.6** We obtain the virtual machine with procedures `VMP` by extending `VM` by `ASM` implementations for the new $\mathcal{L}(\texttt{VMP})$ instructions. (on the next slides)

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	$D(0) := ACC$	cache VPC
	LOADIN 1 2	$ACC := D(\text{VPC} + 2)$	load length
	ADD 0	$ACC := ACC + D(0)$	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	$ACC := D(\text{VPC} + 1)$	load i
	STORE 0	$D(0) := ACC$	cache i
	MOVE $IN3$ ACC		
	STORE 1	$D(1) :=$ FP	cache FP
	ADDI -1		
	SUB 0	$ACC :=$ FP $- 1 - i$	load argument position
	MOVE ACC $IN3$	FP $:= ACC$	move it to FP
	inc $IN2$	SP $:=$ SP $+ 1$	prepare push
	LOADIN 3 0	$ACC := D(\text{FP})$	load arg i
	STOREIN 2 0	$D(\text{SP}) := ACC$	push arg i
	LOAD 1	$ACC := D(1)$	load FP
	MOVE ACC $IN3$	FP $:= ACC$	recover FP
	MOVE $IN1$ ACC		
	ADDI 2		
	MOVE ACC $IN1$	VPC $:=$ 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	$D(0) := IN1$	cache current VPC
	inc $IN2$	SP $:=$ SP $+ 1$	prepare push for later
	LOADIN 1 1	$ACC := D(\text{VPC} + 1)$	load argument
	ADDI $2^{24} + 3$	$ACC := ACC + 2^{24} + 3$	add displacement and skip proc a l
	MOVE ACC $IN1$	VPC $:= ACC$	point to the first instruction
	LOADIN 1 -2	$ACC := D(\text{VPC} - 2)$	stealing a from proc a l
	STOREIN 2 0	$D(\text{SP}) := ACC$	push the number of arguments
	inc $IN2$	SP $:=$ SP $+ 1$	prepare push
	MOVE $IN3$ ACC	$ACC := IN3$	load FP
	STOREIN 2 0	$D(\text{SP}) := ACC$	create anchor cell
	MOVE $IN2$ $IN3$	FP $:=$ SP	update FP
	inc $IN2$	SP $:=$ SP $+ 1$	prepare push
	LOAD 0	$ACC := D(0)$	load VPC
	ADDI 2	$ACC := ACC + 2$	point to next instruction
	STOREIN 2 0	$D(\text{SP}) := ACC$	push the return address
	JUMP ⟨jt⟩		jump back

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	$ACC := D(\mathsf{SP})$	load top value
	`STORE` 0	$D(0) := ACC$	cache it
	`LOADIN` 2 −1	$ACC := D(\mathsf{SP}-1)$	load return address
	`MOVE` ACC $IN1$	$IN1 := ACC$	set VPC to it
	`LOADIN` 3 −1	$ACC := D(\mathsf{FP}-1)$	load the number n of arguments
	`STORE` 1	$D(1) := D(\mathsf{FP}-1)$	cache it
	`MOVE` $IN3$ ACC	$ACC := \mathsf{FP}$	$ACC = \mathsf{FP}$
	`ADDI` −1	$ACC := ACC - 1$	$ACC = \mathsf{FP}-1$
	`SUB` 1	$ACC := ACC - D(1)$	$ACC = \mathsf{FP}-1-n$
	`MOVE` ACC $IN2$	$IN2 := ACC$	$\mathsf{SP} = ACC$
	`LOADIN` 3 0	$ACC := D(\mathsf{FP})$	load anchor value
	`MOVE` ACC $IN3$	$IN3 := ACC$	point to previous frame
	`LOAD` 0	$ACC := D(0)$	load cached return value
	`STOREIN` 2 0	$D(IN2) := ACC$	pop return value
	`JUMP` ⟨jt⟩		jump back

Note that all the realizations of the $\mathcal{L}(\texttt{VMP})$ 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 $\mathcal{L}(\texttt{VMP})$ programs. Just as above, we will write this compiler in SML.

12.4.3 Compiling Basic Functional Programs

For the μML compiler we will proceed as above: we first introduce SML data types for the abstract syntax of $\mathcal{L}(\texttt{VMP})$ and μML and then we define a SML function that converts abstract μML programs to abstract $\mathcal{L}(\texttt{VMP})$ 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
```


This is a very direct realization of the idea that a μML program consists of a list of (function and value) declarations and a return expression, where a function declaration introduces new names for the function and its (formal) arguments. The realization of expressions is just as it was in the abstract syntax for `SW`, only that conditionals that were statements there are now expressions. The only real novelty is the `Id` constructor that represents identifiers (function or argument names).

As always, we fortify our intution by looking at a concrete example: the μML function from Example 12.4.2.

Concrete vs. Abstract Syntax of μML

▷ A μML program first declares procedures, then evaluates expression for the return value.

```
let
  fun exp(x,n) =
    if n<=0
    then 1
    else x*exp(x,n−1)
 val y 10
in
  exp(y,2)
end
```

```
([
  ("exp", ["x", "n"],
   If(Leq(Id"n", Con 0),
      Con 1,
      Mul(Id"x", App("exp", [Id"x", Sub(Id"n", Con 1)])))),
  ("y", [], Con 10)
 ],
 App("exp", [Id "y", Con 2])
)
```


We define the abstract syntax for $\mathcal{L}(\texttt{VMP})$ as an extension of the one for $\mathcal{L}(\texttt{VM})$. Again we introduce type names for documentation. We want to keep apart the integers we use as numbers of arguments and instructions for `proc`, and distinguish them from the code addresses used `call`.

Abstract Syntax for $\mathcal{L}(\texttt{VMP})$

▷ Extensions to the abstract data types for $\mathcal{L}(\texttt{VM})$ ...

```
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 proc. call
  *)

  type code = instruction list                (* recap *)
```

▷ . . . and the auxiliary length function

```
fun wln ···
  | wln (proc _ _)=3 | wln (arg _)=2
  | wln (call _)=2 | wln (return)=1
```

For our μML compiler, we first need to define some infrastructure needed for functional programs: we need to use the environment for both procedure names and argument names (and we do not assume separate namespaces for arguments and procedures). So we use different constructors for the values of the identifiers in the environment and provide two separate lookup functions that also do some error reporting.

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)
```

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)
```

Next we define the μML expression compiler: a function `compileE` that compiles abstract μML expressions into lists of abstract $\mathcal{L}$(`VMP`) instructions. As expressions also appear in argument

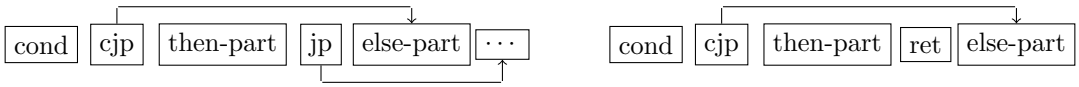
sequences, it is convenient (and indeed necessary since we do not know the arities of functions) to define an expression list compiler, – a function `compileEs` that compiles μML expression lists via left folding. Note that the two expression compilers are very naturally mutually recursive.

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` statement needed to return from recursive functions.

Compiling μML: Expressions

```
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
```


Observe the use of the `tail` argument for the `If` case in `compileE`; the declaration compiler will use it to pass the `return` command to `complileE`. For all expressions, we can just append the `return` (`tail` will always be `return` or the empty list) to the end of the generated code. The only exception is the `If` expressions, where we need to return from the "then-part" and the "else-part" separately. Moreover, we can optimize the generated code by realizing that we do not the traditional two-jump pattern for if-then-else (on the left of the image below),



but only a pattern without the second jump (the pattern on the right), since the "then-part" already returns on its own. These two patterns are realized in the `If` case of `compileE`.

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.

The declaration compiler recurses over the list of declarations, and for each declaration, it first inserts the procedure name into the environment `env`, yielding a new environment `env'`, into which we then insert the argument list via an auxiliary function `insertArgs` we have defined before. In the resulting environment `env''` we compile the body of the function (which may contain the formal arguments). Note that we compile the rest of the declarations in the environment `env'` that contains the function name, but not the function arguments. Indeed it is an error to use the procedure arguments outside the procedure body; using the environment `env'` which does not contain them leads to an exception if they are.

Compiling μML Declarations

```
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
```


Note that in the $\mathcal{L}(\texttt{VMP})$ code we generate from μML programs we do not use `peek` and `poke` instructions. This is in keeping with the functional nature of the μML language, which does not have variable assignment (we used `poke` for that in the imperative programming language `SW`), and (global) variable declarations (we used `peek` to access them). In stead of variable declarations, we have function and value declarations which are accessed by `call` in $\mathcal{L}(\texttt{VMP})$ compiled from μML.

Note that even if we do not use them in the μML compiler, `peek` and `poke` are still available in $\mathcal{L}(\texttt{VMP})$ (which is defined to be a superset of $\mathcal{L}(\texttt{VM})$). And indeed this is a good thing, as we would need them hem if we were to build an imperative programming language with procedures – e.g. building on `SW`.

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 $\mathcal{L}(\texttt{VMP})$ code from μML programs.

▷ μML is minimal, but Turing-Complete (has conditionals and procedures)

12.5 Turing Machines: A theoretical View on Computation

In this section, 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 (combinational 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 12.5.1** *[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.

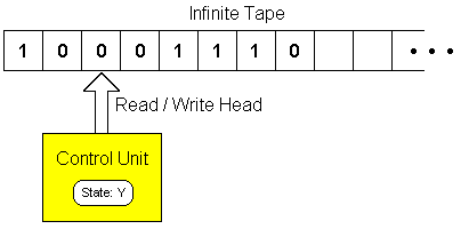
Turing Machines: The idea

▷ 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."

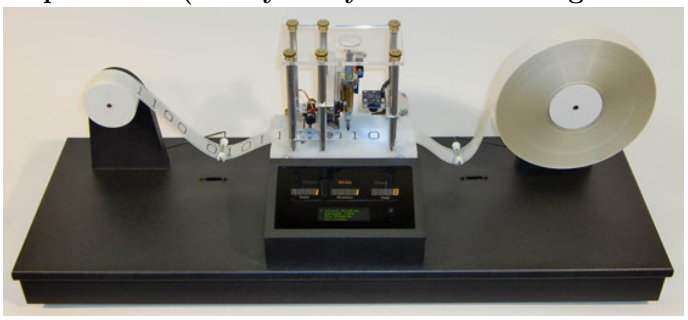


Note that the physical realization of the machine as a box with a (paper) tape is immaterial, it is inspired by the technology at the time of its inception (in the late 1940ies; the age of ticker-tape commuincation).

A Physical Realization of a Turing Machine

▷ Note: Turing machine can be built, but that is not the important aspect

▷ **Example 12.5.2 (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 12.5.3** 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 specia
- ▷ 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 12.5.4** 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 12.5.3 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 12.5.5 with Alphabet $\{0, 1\}$

▷ Given: a series of 1s on the tape (with head initially on the leftmost)

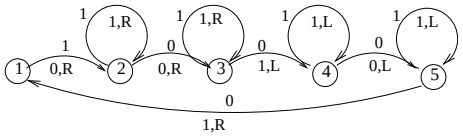
▷ Computation: doubles the 1's with a 0 in between, i.e., "111" becomes "1110111".

▷ 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	0			f

▷ State Machine:



The computation of the turing machine is driven by the transition function: 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.

#	St	Tape	#	St	Tape
1	s_1	[1] 1	9	s_2	10 [0] 1
2	s_2	0 [1]	10	s_3	100 [1]
3	s_2	01 [0]	11	s_3	1001 [0]
4	s_3	010 [0]	12	s_4	100 [1] 1
5	s_4	01 [0] 1	13	s_4	10 [0] 11
6	s_5	0 [1] 01	14	s_5	1 [0] 011
7	s_5	[0] 101	15	s_1	11 [0] 11
8	s_1	1 [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 12.5.6 ([Church/Turing])** *(unprovable, but accepted)*
  Anything that can be computed at all, can be computed by a Turing Machine
- ▷ **Definition 12.5.7** 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 Section12.0.

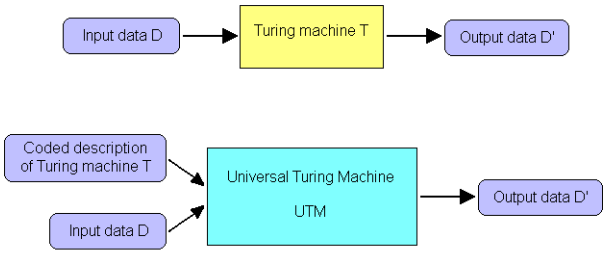
Universal Turing machines

- ▷ 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 12.5.8** *Such a Turing machine is indeed possible(e.g. with 2 states, 18 symbols)*

▷ **Definition 12.5.9** 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.

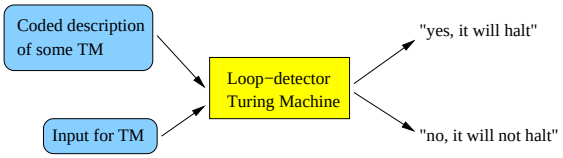
Question: So, if all non-trivial programming languages can compute the same, are there things that none of them can compute? This is what we will have a look at next.

▷ Is there anything that cannot be computed by a TM

▷ **Theorem 12.5.10 (Halting Problem [Tur36])** *No Turing machine can infallibly tell if another Turing machine will get stuck in an infinite loop on some given input.*

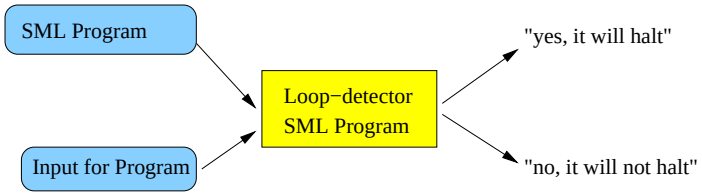
▷ The problem of determining whether a Turing machine will halt on an input is called the halting problem.

▷



▷ 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`

P.2 Consider the following program

```
fun turing (prog) =
   if will_halt(eval(prog),prog) then looper(1) else 1;
turing : string -> string
```

P.3 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.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.

P.2 But if the function happens to be the `turing` function itself, then

- ▷ 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)

P.3 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.

Bibliography

[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.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.

[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.

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

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

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

[Koh13] Michael Kohlhase. sTeX: Semantic markup in TeX/LaTeX. Technical report, Comprehensive TeX Archive Network (CTAN), 2013.

[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/`.

[Pau91] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[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.

[Zus36] Konrad Zuse. Verfahren zur selbsttätigen durchführung von rechnungen mit hilfe von rechenmaschinen. Patent Application Z23139/GMD Nr. 005/021, 1936.